LATERAL SHEAR INTERFEROMETRY

TESIS QUE COMO REQUISITO PARA OBTENER EL GRADO DE MAESTRO EN CIENCIAS (ÓPTICA) PRESENTA

ING. MIGUEL ANGEL CASILLAS ARAIZA

ASESOR: DR. MANUEL SERVÍN GUIRADO

LEÓN, GTO. **FEBRERO 2004**

Interferometry
Using
The Lateral Shear Interferometer.

by

Miguel Ángel Casillas Araiza.

B.S. Universidad Iberoamericana
(Plantel León, Gto.; México)

(1995)

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
at the Center of Research in Optics

February 2004.

Signature of the author_____

Accepted by _____
Dr. Manuel Servin Guirado.
M.S. Adviser of the Work.

Dedicatory.

To my wife and sons, Rosario, Ángel Roberto and Obed Salomón.

To my parents, Ma. Teresa and Miguel.

To my sister, Elizabeth and brothers, Luis Gerardo, Francisco Javier.

Acknowledges

I want to acknowledge to the CoNaCyT, for support me during the development of this work. Also, I want to thank to Dr. Manuel Servin G. for his valuable teaching during my instruction in the interesting area of the optics, the Interferometry.

Foreword

General aspects

This work has a CD where complete thesis is available. There is an electronic address where the routines (developed in Java Language) presented in Appendix E and the code in Appendix F can be used (http://www.cio.mx/mcasillas). With the disposition for use the technology, i.e. the use of code, we pretend share information (code) for fast evolution finding better ways of do the things. We pretend that the people can read the papers, thesis and other documents in order to understand the concepts of an interactive way, even improve them (do them better). These ideas are related with the fact that soon the new scientific magazines will be published through the Internet. This allows the interaction between different groups working in an environment that work in multi-platform.

We pretend to breakdown with some paradigms, first, the thesis was written in two languages, the motivation is that in own real life as scientist and Mexican we must talk and write in both languages (Spanish and English). Because of this all the chapters are presented in English and the Appendices are written in Spanish. Appendix C is written in Spanish but the titles of each section are written in English because the techniques are better known with its original names in English.

Second, the presentation is also changed, we think that the rules are not agree with an ecologist point of view, because the rules were establish long time ago when personal computer were not available. We use both sides of pages (fewer trees will be used in many copies of my thesis work). Also change the interlined to a single line. We think that the presentation is clear enough and is not necessary more space on each written line.

As will be observed, many equations were demonstrated from obvious precepts, making the understanding of this thesis in a pedagogic current called constructivism, we pretend that all the necessary information is self-contained in the thesis (we try this as much as were possible).

Content and organization

In chapter 1 we present some historical review of the interferometry in general and in particular an introduction to the Lateral Shear Interferometry (LSItry). Some shearograms produced by aberrations of third order in the Lateral Shear Interferometer (LSI) are exhibited. A little information about the LSItry is presented in an introductory way. The chapter 2 is concern with the basis of the interference, aspects of coherence of electromagnetic waves that superpose in space. An example of two plane waves that interfere in space is developed using a vectorial description. Chapter 3 studies the Lateral Shear Interferometer (LSI), third-order primary aberrations are developed for the LSI; this set of equations can be used to calculate the shearogram produced by this kind of

interferometer. A complete study of the Murty's interferometer is developed. Regarding about its parameter to be used to design are showed and demonstrated. Some characteristics of the Lateral Shear Interferometer like the frequency response are shown; the LSI like a linear system and the sensitivity of the LSI respect to the displacement is also calculated. In chapter 4 we study the estimation of the wave-front from maps of phase lateral sheared unwrapped, this chapter pretend to explain the effects of try to recover the wave-front from simple inverse transfer function. The least squares integration for lateral displacements greater than one CCD pixel produce a matrix that can't be inverted, in this case, to yield a stable solution the regularizing potentials are a must. A particular case is studied, i.e. with little shearing, the method of Fried-Hudgin for integrating field of gradient, has a well-defined and unique minimum. Therefore it is not necessary to regularize the problem for one CCD pixel lateral shearing. Introducing higher lateral shearing is better to increase the sensitivity of the measurement. In chapter 5 we review the loss of information due to a large lateral shearing in the LSI. The effect of the pupils is that there is loss of information due to exist points on the wave-front that don't interfere with another point of the wave-front, so on that point is truncated from the observations and must be interpolated by a priory knowledge. This information is reconstructed from the regularizing potentials acting like a low-pass filter, i.e. smoothing the data observed. In chapter 6 are shown several results in one dimension of the recovery of the wave-front. The effects of large lateral shearing on the matrix used to recover the wave-front are exhibit. The conclusion about the work is presented in this chapter and the future work.

In Appendix A is presented a synthesis and a classification of signals. In Appendix B are developed the relationships between Fourier series and Fourier transform, form continuous domain to discrete domain (all the connections to go forward and backward between them). In Appendix C is presented a very synthetic explanation and classification of several techniques to estimate the wrapped or unwrapped phase from an interferogram. Appendix D shows an Elementary manual of Java for beginners. Finally the Appendix E exhibits the algorithms in mathematical logic for solving a linear system of equations, the methods showed are: Gradient, Conjugate Gradient, and Preconditioned Conjugate Gradient. A Glossary of term pretends to clarify some of the most usual concepts used frequently in this work.

General Index

Glossary.

Digital image is a two-dimensional digital signal, since the intensity or brightness $I[n_x, n_y]$ at each point is a function of two spatial independent discrete-variables.

Diffraction is the deviation of light from rectilinear propagation, that is neither refraction or reflection. The effect is a general characteristic of wave phenomena occurring whenever a portion of a wave-front, be it sound, a matter wave, or light, is obstructed in some way. There is no significant physical distinction between *interference* and *diffraction*.

Discrete-variable is a sequence of real or complex numbers.

Discrete-variable signals are defined only at certain specific values of the discrete-variable. These specific values need not be equidistant, but in practice they are usually taken at equally spaced intervals for computational convenience and mathematical tractability.

Dispersion is the dependence of index refraction on the wavelength (or color) of light.

Dissipative absorption is the process of removal of energy from an incident wave, the amount of energy removed from the incident wave, increases the frequency of the wave approaches a natural frequency of the atom. If the photon's energy matches that of one of the excited states, the atom will "absorb" the light, making a quantum jump to that higher energy level, this energy will rapidly be transferred, via collisions into thermal energy.

Index refraction is a quantity that express the ratio between speed o light on vaccum and the velocity of light in a material. Each material have different index refraction upon the wavelength.

Image is the reproduction of the appearance of an object due to the convergence real or virtual of the ray of light that come form every point on the object, go through the optical system, or reflect on it.

Interference Phenomenon that is produced by the superposition on a point in the space of two or more waves. This phenomenon requires spatial and temporal coherence of such waves. [See chapter 2].

Interferogram is the image produced by an interferometer usually this image on a plane.

Interferometer Instrument that is used for make measures in the interference field or their applications.

Interferometry Branch of physics that study the interference phenomenon.

Laser Acronym (Light Amplification by Stimulated Emission of Radiations). Device that produce coherent ray of light specially intense.

Lens Glass or plastic dish, or whatever material that produce refraction, whose faces generally have spherical profile.

Light is a part of the electromagnetic spectra that allow reconstruct the images from the objects, on the eye, on a photographic plate, or on other photosensitive surfaces.

Light ray Related to wave-front is the normal on an infinitesimal section of the wave-front. Describe de direction in which every infinitesimal section of wave-front propagates.

Reflection is the rejection of the waves of light by the interface between two medium. The light propagates through one of this media and the interface is rejected by the other media with a different index refraction.

Refraction is the change in direction that suffer a light ray when it propagates from one material to other with different index refraction.

Scattering is the removal of energy from an incident wave and the subsequent reemission of some portion of that energy.

Sequence of numbers is a set of numbers in an specific order.

Signal is defined as any physical quantity that varies with time, space, or any other independent variable or variables.

Optics is the science that study the origin, propagation, and detection of light. Including visible, infrared, and ultraviolet electromagnetic radiation; and all their phenomena involved like interaction radiation-matter, non-linear phenomena, etc...

Wave-front is the locus of the points on the space where the phase of the wave is constant. In aberration theory means an error between an ideal wave-front and the actual wave-front.

Wavelength is the spatial length of periodicity of a wave. Its units are of length.

Note:
All this definitions intend an optical purpose.

Figures List

Footprints

Chapter 1.

Chapter 2.

Chapter 3.

Chapter 4.

Chapter 5.

Chapter 6.

Symbols List.

$\in$ - electric permitivity of a continuous medium.
λ - is the spatial wavelength of an electromagnetic wave.
E – represent the magnitude the electric field of the propagated electromagnetic wave.
R – represent the reflectance coefficient on a surface of a incident ray.
T – represent the transmittance coefficient on a surface of a incident ray.
W – represent the wave-front of the propagated wave.
ϕ - is the phase of the propagated wave-front.
P – is the pupil function of a beam of light propagated.
h – is the impulse response of a linear system.
H – is the transfer function of a linear system.
$S_{\delta}^{\Delta W}$ - is the sensitivity of the LSI respect to the lateral shearing.
U – Energetic functional to estimate the squared error medium of a function with a field of observations.
$\Im$ - Fourier transform operator on *n*-variables.
$\Im_x$ - Fourier transform operator on *x*-direction.
$\Im_y$ - Fourier transform operator on *y*-direction.
$R(\bullet)$ – regularization term (first or second order potentials).

Tables List

Chapter 6.

Appendix D.

1 Introduction

1.0 Objectives of this work.

- The aim to this work is to develop a comprehensive reference guide about the characteristics of the Lateral Shear Interferometry (LSIrty) for the news in the theme and for the optical engineers that would like design an instrument using this kind of interferometer of not reference absolute.
- Present the basis of the interference phenomenon. From the requirements of coherence in the light, visibility in the fringes of interference, superposition of electromagnetic waves in space, through interference of two waves in any point in the space.
- Explore the characteristics of the Lateral Shear Interferometer (LSI) like different transformations over the wave-front under analysis, tilt, scale, lateral shear, some of this transformation are wished and other aren't. Additionally, some analysis to evaluate deviations of the ideal model of the LSI with the actual one. Apply some properties of Linear and spatial invariant systems to the LSI with infinite pupil. Analysis of sensitivity of the LSI to the lateral shear.
- Study the frequency response of LSI considering infinite and finite pupils.
- Obtain closed forms of the third order aberrations in the LSI and their approximations when the lateral shear tends to cero.
- Study the main characteristics of the Murty's interferometer, in order to calculate every part needed to built it.
- Exhibit the frequency response of the simple inverse filter used to recover the original wave-front.
- Analyze the Fried-Hudgin method for small lateral shear but it's inadequate for larger lateral shear.
- Show that the reconstruction of the wave-front using the technique of least squares with regularization has a better frequency response than the method of least squares integration without regularization in order to estimate the original wave-front; due mainly to the fact that we add extra information about the smoothness of the estimated wave-front is introduced.
- Use the technique of Least squares summation with regularization for large lateral shear.
- Analysis of lost of certain points in the wave-front due a large lateral shear upon the pupil's geometry. Estimation of the losses points in the wave-front by means of membrane and plate potentials.
- Representation of the linear set of equation in the quadratic functional in order to show the inconsistency in the solution of the linear set of equations, i.e. its determinant is cero due to the lost of information in the original wave-front.
- Show how the regularization introduces information that produces in the matrix representation a determinant different of cero for larger lateral shear than one CDD-pixel.
- Show how the regularization introduces error in high frequencies because is a smoothed filter.

- Show the results obtained using least squares minimization with regularization to estimate the original wave-front.

1.1 Objectives of this chapter.

- Present the objectives of the work.
- Present a historical introduction to the inferferometry.
- An introduction to the Basis of Lateral Shear Interferometry.

1.2 Historical Introduction.

The philosophers of antiquity speculated about the nature of light, being familiar with burning glasses, with the rectilinear propagation of light, and with refraction and reflection. The first systematic writings on optics of which we have any definite knowledge are due to the Greek philosophers and mathematicians, Empedocles (c. 490-430 B.C.), Euclid (c. 300 B.C.).

Amongst the founders of the new philosophy, René Descartes (1596-1650) may be singled out for mention as having formulated views on the nature of light on the basis of his metaphysical ideas. Descartes considered light to be essentially a pressure transmitted through a perfect elastic medium (the aether) which fills all space, and he attributed the diversity of colors to rotary motions with different velocities of the particles in this medium. But it was only after Galileo Galilei (1564-1642) had, by his development of mechanics, demonstrated the power of the experimental method that optics was put on a firm foundation. The law of reflection was known to the Greeks; the law of refraction was discovered experimentally in 1621 by Willebrord Snell. In 1657 Pierre de Fermat enunciated the celebrated Principle of least time. According to this principle, light always follows that path which brings it to its destination in the shortest time, and from this, in turn, and from the assumption of varying “resistance” in different media, the law of refraction follows.

The first phenomenon of interference, the colors exhibited by thin films now known as “Newton’s rings”, was discovered independently by Robert Boyle (1627-1691) and Robert Hooke (1635-1703). Hooke also observed the presence of light in the geometrical shadow, the “diffraction” of light.

The first accurate report and description of diffraction effects was made by Grimaldi and was published 1665. The measurements report were made with an experimental setup which consist of an aperture in an opaque screen was illuminated by a light source, chosen small enough to introduce a negligible penumbra effect; the intensity was observed across a plane some distance behind the screen. The corpuscular theory of light propagation, which was the accepted means of explaining optical phenomena at the time, predicted that the shadow behind the screen should be defined, with sharp borders. Grimaldi’s observations indicated, however that the transition from light to shadow was gradual rather than abrupt.

If the spectral purity of the light source had been better, he might have observed even more striking results, such as the presence of light and dark fringes extending far into the geometrical shadow of the screen. Such effect cannot be explained by a corpuscular theory of light, which requires rectilinear propagation of light rays in the absence of reflection and refraction.

The first proponent of the wave theory of light was Christian Huygens in the year 1678. He expressed the intuitive conviction that if each point on the wavefront of a disturbance, then the wavefront at a later instant could be found by constructing the "envelope" of the secondary wavelets.

The diffraction theory was impeded throughout the entire 18^{th} century by the fact that Isaac Newton, a scientist with a enormous reputation for his many contributions to physics in general and to optics in particular, favored the corpuscular theory of light as early as 1704. In 1804 Tomas Young, an English physician, strengthened the wave theory of light by introducing the critical concept of interference. The idea was a radical one at the time, for it stated that under proper conditions, light could be added to light and produce darkness. The dark and light zones that would be seen on a screen placed in the region of the added light are known interference fringes. It should be kept in mind that for a fringe pattern to be observed, the two sources need not be in phase with each other. A somewhat shifted but otherwise identical interference pattern will occur if there is some initial phase difference between the sources, so long as it remains constant. Such sources (which may or may not be in step but are always marching together) are said to be coherent.

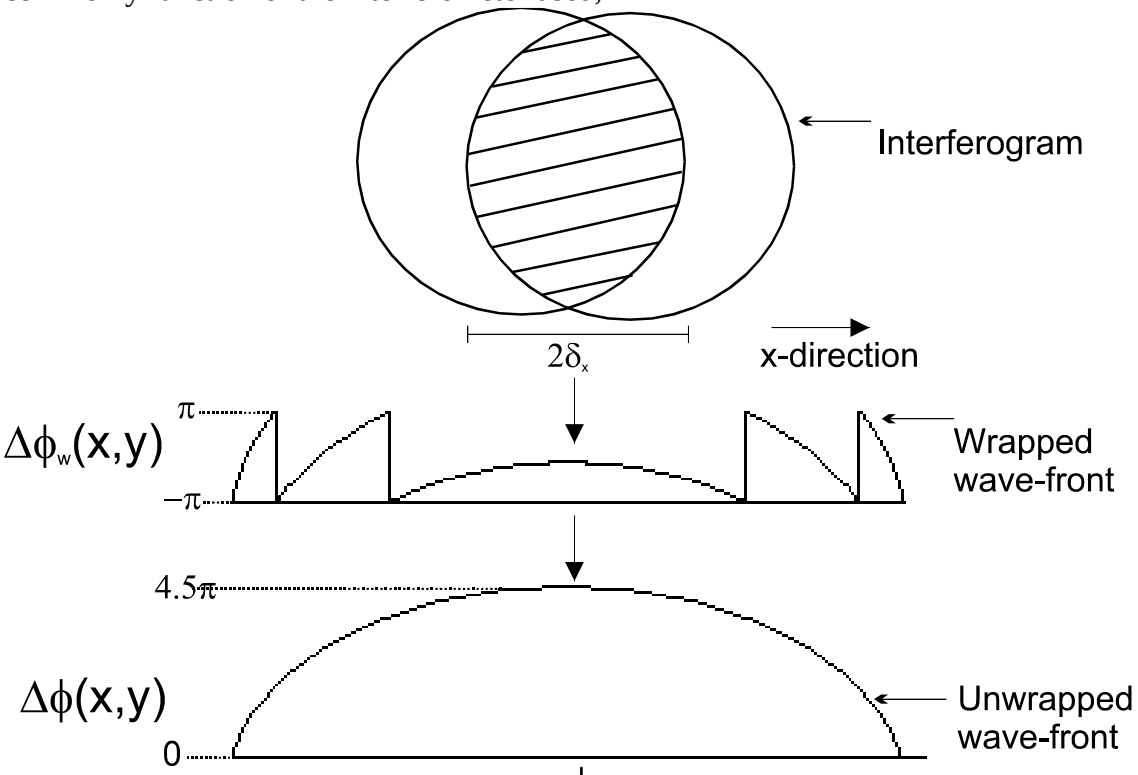
The ideas of Huygens and Young were brought together in 1818 in the famous memory of Augusting Jean Fresnel. By making some rather arbitrary assumptions about the amplitudes of phases of Huygens' secondary sources, and by allowing the various wavelets to mutually interference, Fresnel was able to calculate the distributions of light in diffractions patterns with excellent accuracy.

The Fresnel's theory was strongly disputed by the great French mathematician S. Poisson. He demonstrated the absurdity of the theory by showing that it predicted the existence of a bright spot at the center of the shadow of an opaque disk. F. Arago, performed such an experiment and found the predicted spot, since then the effect has been known as "Poisson's spot".

In 1860 Maxwell identified light as an electromagnetic wave, a step of big importance. But it was not until 1882 that the ideas of Huygens and Fresnel were put on a firmer mathematical foundation by Gustav Kirchhoff, who succeeded in showing that the amplitudes and phases ascribed to the secondary sources by Fresnel were indeed logical consequences of the wave nature of light.

1.3 Overview of the wave-front reconstruction from its lateral shear interferogram.

The process used to recover an estimated wave-front from it's interferogram is commonly function of the interferometer used,



$$\lim_{\delta_x \to 0} W(x-\delta_x, y) - W(x+\delta_x, y) = 2\delta_x \frac{\partial W(x,y)}{\partial x}$$

$$\downarrow$$

$$\int \Delta\phi(x,y)\,dx$$

$$\downarrow$$

$$\phi(x,y)$$

$$\downarrow$$

$$W(x,y)$$

Figure 1.1 Flow diagram to calculate $W(x,y)$.

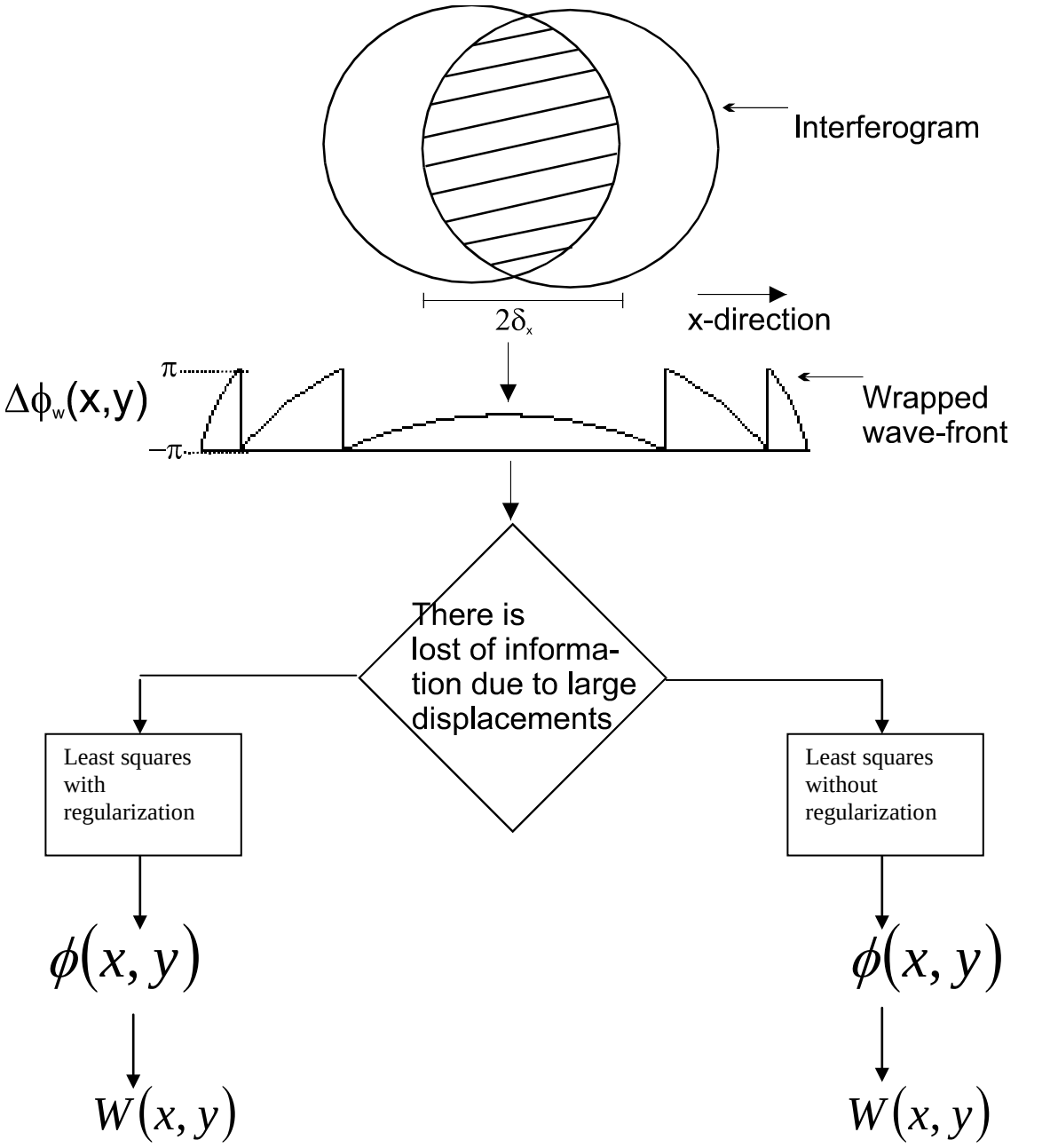


Figure 1.2 Flow diagram used to calculate $W(x,y)$.

1.4 Basis of the interferometry in the LSI.

For an interferogram produced by a LSI sheared along the *x*-direction. The pattern fringes can be modeled ideally by

$$I_x(x,y) = a(x,y) + b(x,y)\cos\left[\frac{2\pi}{\lambda}\left(W(x-\delta_x,y) - W(x+\delta_x,y)\right)\right]P(x-\delta_x,y)P(x+\delta_x,y) \tag{1.1}$$

where $I_x(x,y)$ is the recorded interferogram's intensity.

$a(x,y)$ is the low-frequency background illumination.

$b(x,y)$ is a low-frequency amplitude modulation.

$2\delta_x$ is the amount of shear introduced.

$P(x,y)$ is the aperture that limits the extent of the wave-front under analysis, $W(x,y)$. The function $P(x,y)$ is equals one within the beam of light where the wave-front being analysed propagates and equals zero otherwise.

Figure 1.1 shows a "common" circular pupil $P(x,y)$ that limits the shearogram, and consequently the wave-front $W(x,y)$.

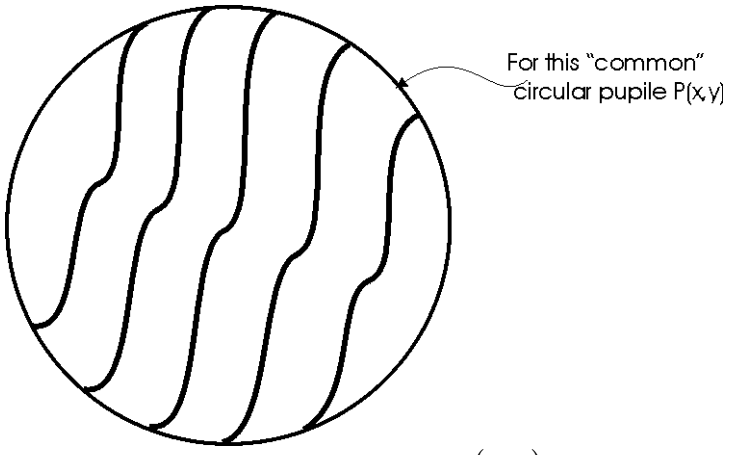


Figure 1.3 "Common" circular pupils $P(x,y)$.

The phase wrapped term is

$$\Delta\phi_w(x,y) = Wrapped\left[\frac{2\pi}{\lambda}\left(W(x-\delta_x,y) - W(x+\delta_x,y)\right)\right] \quad \text{with } -\pi \le \phi_w(x,y) \le \pi \tag{1.2}$$

where

$$Wrapped[x] = |x| \ Module\ 2\pi$$

The phase unwrapped term is

$$\Delta\phi(x,y)=\frac{2\pi}{\lambda}\left[W(x-\delta_x,y)-W(x+\delta_x,y)\right]=\frac{2\pi}{\lambda}\left[O.P.D.\right] \tag{1.3}$$

remembering that O.P.D. means Optical Path Difference.

Due that, the LSI is an interferometer of not absolute reference; the phase $\phi(x,y)$ that produce the fringes in the LSI is produced by the O.P.D., i.e. the difference of a wave-front $W(x,y)$ by itself lateral sheared $W(x-2\delta_x,y)$, for mathematical simplicity this shearing is represented by the displacement of the wave-front symmetrically around the origin along *x*-direction, i.e. $W(x-\delta_x,y)$ and $W(x+\delta_x,y)$. For small displacement, in the limit when $\delta_x \to 0$, we can approximate this difference to the first partial derivative respect to *x*-variable.

$$lim_{\delta_x\to 0}\left[W(x-\delta_x,y)-W(x+\delta_x,y)\right]=2\delta_x\frac{\partial W(x,y)}{\partial x} \tag{1.4}$$

This development will be showed in following chapters in a more exact manner.

1.5 Frequency response of the Lateral Shear Interferometer.

The lateral shear interferometer exhibits a peculiar frequency response. In some spatial frequencies has zero output (pattern of interference usually called interferogram), in this frequencies doesn't matter the input (wave-front going into the LSI). These frequencies with zero response eliminate some components of the input spectrum thus affecting the original input. These frequencies are function of the lateral shear

A more precise study of the frequency response of the Lateral Shear Interferometer will be shown in chapter 3 sections 3.6 Response in frequency of the lateral shear interferometer (considering infinite pupils) & 3.7 Response in frequency of the lateral shear interferometer (considering finite pupils).

1.6 Characteristics of a Lateral Shear Interferometer.

Interferograms obtained using ordinary interferometers, such as the Fizeau interferometer or the Twyman-Green interferometer, show the contour maps of the wave-front under test. The fringes in a Twyman-Green inferferogram are the loci of constant wave-front phase.

On the other hand, lateral shearing interferograms show the difference between a wave-front under test and a sheared wave-front, that is, the inclination of the wave-front.

The interferograms are produced by the optical path difference between a wave-front under test and the laterally sheared wave-front.

The fringes of a shearing interferogram are the loci of constant average wave-front slope over the shear distance, i.e., a lateral shear interferogram gives wave-front slope information only for the direction of shear (for a small shearing).

1.7 Examples of interferograms (computer generated).

In following we show different interferogram produced by the third order aberrations on a lateral shear interferometer.

Figure 1.4 Interferogram produced by defocusing

Figure 1.5 Interferogram produced by Spherical aberration

Figure 1.6 Interferogram produced by Spherical aberration and defocusing

Figure 1.7 Interferogram produced by Coma aberration, Sagittal Shear

Figure 1.8 Interferogram produced by Astigmatism

Figure 1.9 Interferogram produced by a Perfect wave-front.

1.8 Lost of information due to the pupil geometry.

Large lateral shearing produce lost of information due to the regions where no interference exist. This lost of information is function of the geometry, in general, for larger lateral displacements there will be larger lost of information.

It's possible introduce some a-priory information in order to interpolate in some spatial regions where the information was lost.

1.9 Some comments about the strategies to recover the lateral shear information.

As we saw in the section 1.3 Overview of the wave-front reconstruction from it's lateral shear inferferogram, in general, there are several options to get an estimation of the original wave-front. It is important consider the size of lateral shear in order to increase sensitivity but also is important to enclose this lateral shear to low lost of information.

We distinguish tree main aspects to be considered to better use the LSI:

- Sensitivity
- Lost of spectral components in the wave-front that goes into the LSI due to its frequency response.
- Lost of information due to a large lateral displacement.

Each case will have peculiar characteristics that make possible use different strategies using the advantages that each one exhibit as was show in section 1.3.

References.

Born, Max and Emil Wolf, *Principles of Optics*, 6ª ed., Pergamon Press, England, 1993.
Goodman, Joseph W., *Introduction to Fourier Optics,* 2ed., McGraw Hill, New York, 1996.
Hetch, Eugene, *Optics*, 2ed., Addison-Wesley, USA, 1990.

2 Theoretical basis of interference.

2.0 Objectives of this chapter.

Present the basis of the interference phenomenon. From the requirements of coherence in the light, visibility in the fringes of interference, superposition of electromagnetic waves in space, through interference of two waves in any point in the space. As a particular example is developed a vectorial treatment of two interfering plane wave-fronts on any point in the space. Finally, is presented a general mathematical description of the interference produced by a Lateral Shear Interferometer.

2.1 Introduction.

When two or more light beams are superposed, the distribution of intensity in the region of superposition is found to vary from point to point between maximums, which exceed the sum of the intensities in the beams, and minima which may be zero. This phenomenon is called *interference*. The superposition of beams of strictly monochromatic light always gives rise to interference. However, light produced by a real physical source is never strictly monochromatic but, the amplitude and phase undergo irregular fluctuations much too rapid for the eye or an ordinary physical detector to follow. If the two beams originate in the same source, the fluctuations in the two beams are in general correlated, and the beams are said to be completely or partially *coherent* depending on whether the correlation is complete or partial. In beams from different sources, the fluctuations are completely uncorrelated, and the beams are said to be mutually *incoherent*. When such beams from different sources are superposed, no interference is observed under ordinary experimental conditions, the total intensity being everywhere the sum of the intensities of the individual beams.

There are two general methods of obtaining beams from a single beam of light, and these provide a basis for classifying the arrangements used to produce interference. In one the beam is divided by passage through apertures placed side by side. This method, which is called *division of wave-front*, is useful only with sufficiently small sources. Alternatively the beam is divided at one or more partially reflecting surfaces, at each of which part of the light is reflected and part transmitted. This method is called division of amplitude; it can be used with extended sources, and so the effects may be of greater intensity than with division of wave-front. In either case, it is convenient to consider separately the effects which result from the superposition of two beams (*two-beam interference*), and those which result from the superposition of more than two beams (*multiple-beam interference*).

2.2 Coherence theory.

In the phenomena of superposition of waves the usual treatment are perturbations absolutely coherent or incoherent. This limit conditions are conceptual idealizations more than physical facts. There is a middle ground between this two extremes that is the domain of the partial coherence. In 1860, Emile Verdet shown that a primary source usually considered as incoherent, such as the sun, could produce visible fringes when near holes are illuminated (≤ 0.05mm) of Young's experiment. Theoretical interest in study of partial coherence lay dormant until it was revived in the 1930s by P.H. van Cittert and later by Fritz Zernike.

The average constituent wave-train exist roughly for a time Δt_c, which is the coherence time given by the inverse of the frequency bandwidth $\Delta\nu$.

It is often convenient, even if rather artificial, to divide coherence effects into two classifications, temporal and spatial. The former relates directly to finite bandwidth of the source, the latter to its finite extent in space.

If the light were monochromatic $\Delta\nu$ would be zero and Δt_c infinite, but this is unattainable. But over a shorter than Δt_c an actual wave behaves essentially as if it were monochromatic. In effect the coherence time is the temporal interval over which we can reasonably predict the phase of the light wave at a given point in space. This then is what is meant by temporal coherence; namely if Δt_c is large, the wave has a high degree of temporal coherence and vice versa.

The same characteristic can be viewed somewhat differently. To that end, imagine that we have two separate points P_1 and P_2 lying on the same radius drawn from a quasi-monochromatic point source. If the coherence length, $c\Delta t_c$, is much larger than the distance (r_{12}) between P_1 and P_2, then a single wave-train can easily extend over the whole separation. The disturbance at P_1 would then be highly correlated with the disturbance occurring at P_2. On the other hand, if this longitudinal separation were much greater than the coherence length, many wave-trains, each with an unrelated phase, would span the gap r_{12}. In that case, the disturbance at the two points is space would be independent at any given time. The degree to which a correlation exist is sometimes spoken of alternatively as the amount of longitudinal coherence. Whether we think in terms of coherence time Δt_c or coherence length $c\Delta t_c$, the effect still arises from the finite bandwidth of the source.

The idea of spatial coherence is most often used to describe effects arising from the finite spatial extent of ordinary light sources. Suppose then that we have a classical broad monochromatic source. Two point radiators on it, separated by a lateral distance that is large compared with λ, will presumably behave quite independently. That is to say, there will be a lack of correlation existing between the phases of the two emitted disturbance. Extended sources of this sort are generally referred to as incoherent. By other meant the

generation of interference fringes is then seemingly a very convenient measure of the coherence.

In both cases of temporal and spatial coherence we are really concerned with one phenomenon, namely, the correlation between optical disturbances. That is, we are generally interested in determining the effects arising from relative fluctuations in the fields at two points in space-time. Admittedly, the term temporal coherence seems to imply an effect that is exclusively temporal. However, it relates back to the finite extent of the wave-train in either space or time. Spatial coherence, or if you will, lateral spatial coherence, is perhaps easier to appreciate, because it is so closely related to the concept of the wave-front. Thus if two laterally displaced points reside on the same wave-front at a given time, the fields at those points are said to be spatially coherent.

2.3 Visibility.

The quality of the fringes produced by an interferometric system can be described quantitatively using the visibility V, which, as first formulated by Michelson, is given by

$$V \equiv \frac{I_{max} - I_{min}}{I_{max} + I_{min}} \qquad 0 \le V \le 1 \tag{2.1}$$

Here I_{max} and I_{min} are the irradiances corresponding to the maximum and adjacent minimum in the fringe system.

2.4 The superposition of waves.

The phenomena of polarization, interference, and diffraction. These all share a common conceptual basis in that they deal, for the most part, with various aspects of the same process. Stating this in the simplest terms, we are really concerned with what happens when two or more light waves overlap in some region of space. The precise circumstances governing this superposition, of course, determine the final optical disturbance. We are interested in learning how the specific properties of each constituent wave (amplitude, phase, frequency, etc.) influence the ultimate form of the composite disturbance.

Each field component of an electromagnetic wave (E_x, E_y, E_z, B_x, B_y, B_z) satisfies the scalar three-dimensional differential wave equation,

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} + \frac{\partial^2 \psi}{\partial z^2} = \frac{1}{\upsilon^2} \frac{\partial^2 \psi}{\partial t^2} \tag{2.2}$$

A significant feature of this expression is that it is linear; in other words, $\psi(\vec{r},t)$ and its derivatives appear only to the first power. Consequently, if $\psi_1(\vec{r},t)$, $\psi_2(\vec{r},t)$, ..., $\psi_n(\vec{r},t)$ are individual solutions of equation (2.2), any linear combination of them will, in turn, be a solution. Thus

$$\boxed{\psi(\vec{r},t)=\sum_{i=1}^{n}C_i\psi_i(\vec{r},t)} \tag{2.3}$$

satisfies the wave equation, where the coefficients C_i are simply arbitrary constants. Known as the principle of superposition, this property suggests that the resultant disturbance at any point in a medium is the algebraic sum of the separate constituent waves. However, large-amplitude waves, can generate a non linear response.

2.5 Interference.

The light is a vector phenomenon; the electric and magnetic fields are vector fields. Still, there are many situations in which the particular optical system can be so configured that the vector nature of light is of little practical significance. We will therefore derive the basic equations within of the vector model, thereafter delineating the conditions under which the scalar treatment is applicable.

In accordance with the principle of superposition, the electric field intensity $\vec{E}$, at a point in space, arising from the separate fields $\vec{E}_1$, $\vec{E}_2$, ... of various contributing sources is given by

$$\vec{E}=\vec{E}_1+\vec{E}_2+\cdots. \tag{2.4}$$

Note that the optical disturbance, or light field $\vec{E}$, varies in time at an exceedingly rapid rate, roughly 4.3×10^{14} Hz to 7.5×10^{14} Hz, making the actual field an impractical quantity to detect. On the other hand, the irradiance I can be measured directly with a wide variety of sensors (e.g., photocells, photographic emulsions, eyes). Indeed then, if we are to study interference, we had best approach the problem by way of the irradiance.

Consider two point sources, S_1 y S_2, emitting monochromatic waves of the same frequency in a homogeneous medium. Furthermore, let their separation a be much greater than λ. Locate the point of observation P far enough away from the sources so that at P the wave-fronts will be planes (Figure 2.1). For the moment, we will consider only linearly polarized waves of the form

$$\vec{E}_1(\vec{r},t)=\vec{E}_{o1}\cos(\vec{k}_1\cdot\vec{r}-\omega t+\varepsilon_1) \tag{2.5a}$$

and

$$\vec{E}_2(\vec{r},t) = \vec{E}_{o2}\cos\left(\vec{k}_2\cdot\vec{r} - \omega t + \varepsilon_2\right). \qquad (2.5b)$$

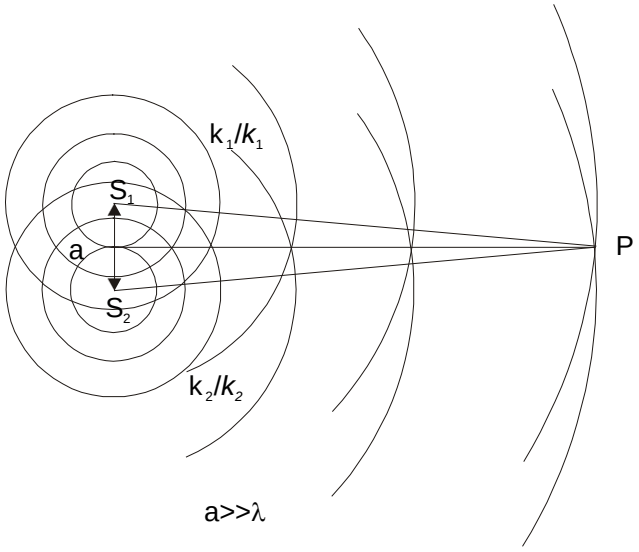


Figure 2.1 Waves from two point sources overlapping in space.

The irradiance at P is given by

$$I = \varepsilon\upsilon\left\langle E^2\right\rangle. \qquad (2.6)$$

Inasmuch as we will be concerned only with relative irradiances within the same medium, we will simply neglect the constants and set

$$I = \left\langle E^2\right\rangle. \qquad (2.7)$$

What is meant by $\left\langle E^2\right\rangle$ is of course the time average of the magnitude of the electric field intensity squared, or $\left\langle\vec{E}\cdot\vec{E}\right\rangle$. Accordingly

$$E^2 = \vec{E}\cdot\vec{E}, \qquad (2.8)$$

where

$$E^2 = \left(\vec{E}_1 + \vec{E}_2\right)\cdot\left(\vec{E}_1 + \vec{E}_2\right) \qquad (2.9)$$

and thus

$$E^2 = \vec{E}_1^2 + \vec{E}_2^2 + 2\vec{E}_1\cdot\vec{E}_2. \qquad (2.10)$$

Taking the time average of both sides, we find that the irradiance becomes

$$I = I_1 + I_2 + I_{12}, \qquad (2.11)$$

provided that

$$I_1 = \left\langle E_1^2 \right\rangle \tag{2.12}$$

$$I_2 = \left\langle E_2^2 \right\rangle \tag{2.13}$$

and

$$I_{12} = 2\left\langle \vec{E}_1 \cdot \vec{E}_2 \right\rangle . \tag{2.14}$$

The latter expression is known as the interference term. To evaluate it in specific instance, we form

$$\vec{E}_1 \cdot \vec{E}_2 = \vec{E}_{o1} \cdot \vec{E}_{o2} \cos\left(\vec{k}_1 \cdot \vec{r} - \omega t + \varepsilon_1\right)\cos\left(\vec{k}_2 \cdot \vec{r} - \omega t + \varepsilon_2\right), \tag{2.15}$$

or equivalently

$$\begin{aligned} \vec{E}_1 \cdot \vec{E}_2 = \vec{E}_{o1} \cdot \vec{E}_{o2} &\left[\cos\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\cos\left(\omega t\right) + \operatorname{sen}\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\operatorname{sen}\left(\omega t\right)\right] \\ &\times \left[\cos\left(\vec{k}_2 \cdot \vec{r} + \varepsilon_2\right)\cos\left(\omega t\right) + \operatorname{sen}\left(\vec{k}_2 \cdot \vec{r} + \varepsilon_2\right)\operatorname{sen}\left(\omega t\right)\right] \end{aligned} \tag{2.16}$$

The time average of some function $f(t)$, taken over an interval $2T$, is

$$\left\langle f(t)\right\rangle = \frac{1}{2T}\int_{-T}^{T} f(t')dt' . \tag{2.17}$$

Taking the time average of $I_1 = \left\langle E_1^2 \right\rangle$

$$\begin{aligned} I_1 = \left\langle E_1^2 \right\rangle &= \left\langle \vec{E}_1 \cdot \vec{E}_1 \right\rangle \\ &= \left\langle \left[\vec{E}_{o1}\cos\left(\vec{k}_1 \cdot \vec{r} - \omega t + \varepsilon_1\right)\right] \cdot \left[\vec{E}_{o1}\cos\left(\vec{k}_1 \cdot \vec{r} - \omega t + \varepsilon_1\right)\right] \right\rangle \\ &= \vec{E}_{o1} \cdot \vec{E}_{o1}\left\langle \cos^2\left(\vec{k}_1 \cdot \vec{r} - \omega t + \varepsilon_1\right) \right\rangle \\ &= E_{o1}^2\left\langle \cos^2\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\cos^2\left(\omega t\right) + \operatorname{sen}^2\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\operatorname{sen}^2\left(\omega t\right) \right\rangle \\ &= E_{o1}^2\left\{ \cos^2\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\left\langle \cos^2\left(\omega t\right) \right\rangle + \operatorname{sen}^2\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\left\langle \operatorname{sen}^2\left(\omega t\right) \right\rangle \right\} , \end{aligned}$$

solving

$$\left\langle \cos^2\left(\omega t\right) \right\rangle = \frac{1}{2T}\int_{-T}^{T}\left[\frac{1}{2} + \frac{1}{2}\cos\left(2\omega t'\right)\right]dt'$$

$$\left\langle \cos^2\left(\omega t\right) \right\rangle = \frac{1}{2}, \tag{2.18a}$$

and

$$\left\langle \text{sen}^2(\omega t)\right\rangle = \frac{1}{2T}\int_{-T}^{T}\left[\frac{1}{2}-\frac{1}{2}\cos(2\omega t')\right]dt'$$

$$\left\langle \text{sen}^2(\omega t)\right\rangle = \frac{1}{2}, \tag{2.18b}$$

hence,

$$\boxed{I_1 = \left\langle E_1^2\right\rangle = \frac{E_{o1}^2}{2}} \tag{2.19}$$

similarly

$$\boxed{I_2 = \left\langle E_2^2\right\rangle = \frac{E_{o2}^2}{2}} \tag{2.20}$$

After multiplying out and averaging equation (2.16),

$$\left\langle \vec{E}_1\cdot\vec{E}_2\right\rangle = \left\langle \begin{array}{l}\vec{E}_{o1}\cdot\vec{E}_{o2}\left[\cos(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\cos(\omega t)+\text{sen}(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\text{sen}(\omega t)\right]\\ \times\left[\cos(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\cos(\omega t)+\text{sen}(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\text{sen}(\omega t)\right]\end{array}\right\rangle$$

$$\left\langle \vec{E}_1\cdot\vec{E}_2\right\rangle = \vec{E}_{o1}\cdot\vec{E}_{o2}$$
$$\times\left\langle \begin{array}{l}\cos(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\cos(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\cos^2(\omega t)+\cos(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\text{sen}(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\cos(\omega t)\text{sen}(\omega t)\\ +\text{sen}(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\text{sen}(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\text{sen}^2(\omega t)+\text{sen}(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\cos(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\cos(\omega t)\text{sen}(\omega t)\end{array}\right\rangle$$

$$\left\langle \vec{E}_1\cdot\vec{E}_2\right\rangle = \vec{E}_{o1}\cdot\vec{E}_{o2}$$
$$\times\left[\begin{array}{l}\cos(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\cos(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\left\langle\cos^2(\omega t)\right\rangle+\cos(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\text{sen}(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\left\langle\cos(\omega t)\text{sen}(\omega t)\right\rangle\\ +\text{sen}(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\text{sen}(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\left\langle\text{sen}^2(\omega t)\right\rangle+\text{sen}(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\cos(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\left\langle\cos(\omega t)\text{sen}(\omega t)\right\rangle\end{array}\right]$$

such as $\left\langle\cos(\omega t)\text{sen}(\omega t)\right\rangle = 0$ (2.18c)

$$\left\langle \vec{E}_1\cdot\vec{E}_2\right\rangle = \frac{\vec{E}_{o1}\cdot\vec{E}_{o2}}{2}\left[\cos(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\cos(\vec{k}_2\cdot\vec{r}+\varepsilon_2)+\text{sen}(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\text{sen}(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\right]$$

$$\left\langle \vec{E}_1 \cdot \vec{E}_2 \right\rangle = \frac{\vec{E}_{o1} \cdot \vec{E}_{o2}}{2} \cos\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1 - \vec{k}_2 \cdot \vec{r} - \varepsilon_2\right),$$

the interference term is then

$$\boxed{I_{12} = \vec{E}_{o1} \cdot \vec{E}_{o2} \cos\left(\vec{k}_1 \cdot \vec{r} - \vec{k}_2 \cdot \vec{r} + \varepsilon_1 - \varepsilon_2\right)} \tag{2.21}$$

where $\left(\vec{k}_1 \cdot \vec{r} - \vec{k}_2 \cdot \vec{r} + \varepsilon_1 - \varepsilon_2\right)$ is the phase difference arising from a combined path-length and initial phase-angle difference. Notice that if $\vec{E}_{o1}$ and $\vec{E}_{o2}$ are perpendicular, $I_{12} = 0$ and $I = I_1 + I_2$.

Considering $\vec{E}_{o1}$ parallel to $\vec{E}_{o2}$. In that case, the irradiance reduces to a scalar value, under those conditions

$$\boxed{I_{12} = E_{o_1} E_{o_2} \cos\left(\vec{k}_1 \cdot \vec{r} - \vec{k}_2 \cdot \vec{r} + \varepsilon_1 - \varepsilon_2\right)} \tag{2.22}$$

using the equations (2.19) and (2.20), the interference term is written as

$$\boxed{I_{12} = 2\sqrt{I_1 I_2} \cos\left(\vec{k}_1 \cdot \vec{r} - \vec{k}_2 \cdot \vec{r} + \varepsilon_1 - \varepsilon_2\right)} \tag{2.23}$$

then the total irradiance is

$$\boxed{I = I_1 + I_2 + 2\sqrt{I_1 I_2} \cos\left(\vec{k}_1 \cdot \vec{r} - \vec{k}_2 \cdot \vec{r} + \varepsilon_1 - \varepsilon_2\right)} \tag{2.24}$$

The maximum visibility of fringes is obtained when amplitude of waves on the point P are equal $I_1 = I_2 = I_o$. Equation (2.23) can now be written as

$$I = 2I_o\left[1 + \cos\left(\vec{k}_1 \cdot \vec{r} - \vec{k}_2 \cdot \vec{r} + \varepsilon_1 - \varepsilon_2\right)\right] = 4I_o \cos^2\left(\frac{\vec{k}_1 \cdot \vec{r} - \vec{k}_2 \cdot \vec{r} + \varepsilon_1 - \varepsilon_2}{2}\right) \tag{2.25}$$

from which it follows that $I_{min} = 0$ and $I_{max} = 4I_o$ using equation (2.25) in equation (2.1),

$$V = \frac{4I_o - 0}{4I_o + 0} = 1.$$

2.5.1 The interference of two plane waves in space.

The interference of two plane waves, can be calculated using a vectorial form. This procedure can be used to estimate a theorethical extended source aproximating it, by a grid of point sources.

$$\vec{E}_1(\vec{r},t) = \vec{E}_{o1}\cos\left(\vec{k}_1 \cdot \vec{r} - \omega t + \varepsilon_1\right) \tag{2.5a}$$

and

$$\vec{E}_2(\vec{r},t) = \vec{E}_{o2}\cos\left(\vec{k}_2 \cdot \vec{r} - \omega t + \varepsilon_2\right) \tag{2.5b}$$

The irradiance is calculated as

$$I = \in v\left\langle E^2 \right\rangle \tag{2.6}$$

As we will be concerned only with relative irradiances within the same medium, simply neglect the constants and set

$$I = \left\langle E^2 \right\rangle \tag{2.7}$$

with $$E^2 = \vec{E} \cdot \vec{E} \tag{2.8}$$

where $$E^2 = \left(\vec{E}_1 + \vec{E}_2\right) \cdot \left(\vec{E}_1 + \vec{E}_2\right) \tag{2.9}$$

so on

$$E^2 = E_1^2 + E_2^2 + 2\vec{E}_1 \cdot \vec{E}_2 \tag{2.10}$$

Taking the time average of both sides, we find that the irradiance is

$$I = I_1 + I_2 + I_{12} \tag{2.11}$$

with

$$I_1 = \left\langle E_1^2 \right\rangle \tag{2.12}$$

$$I_2 = \left\langle E_2^2 \right\rangle \tag{2.13}$$

and

$$I_{12} = 2\left\langle \vec{E}_1 \cdot \vec{E}_2 \right\rangle \tag{2.14}$$

This last expression is known as the interference term.

$$\vec{E}_1 \cdot \vec{E}_2 = \vec{E}_{o1} \cdot \vec{E}_{o2} \cos\left(\vec{k}_1 \cdot \vec{r} - \omega t + \varepsilon_1\right)\cos\left(\vec{k}_2 \cdot \vec{r} - \omega t + \varepsilon_2\right) \tag{2.15}$$

in similar form

$$\begin{aligned}\vec{E}_1 \cdot \vec{E}_2 = \vec{E}_{o1} \cdot \vec{E}_{o2} &\left[\cos\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\cos(\omega t) + \operatorname{sen}\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\operatorname{sen}(\omega t)\right] \\ &\left[\cos\left(\vec{k}_2 \cdot \vec{r} + \varepsilon_2\right)\cos(\omega t) + \operatorname{sen}\left(\vec{k}_2 \cdot \vec{r} + \varepsilon_2\right)\operatorname{sen}(\omega t)\right]\end{aligned} \tag{2.16}$$

$$\begin{aligned}&= \vec{E}_{o1} \cdot \vec{E}_{o2} \\ &\left[\begin{aligned}&\cos\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\cos\left(\vec{k}_2 \cdot \vec{r} + \varepsilon_2\right)\cos^2(\omega t) + \cos\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\operatorname{sen}\left(\vec{k}_2 \cdot \vec{r} + \varepsilon_2\right)\cos(\omega t)\operatorname{sen}(\omega t) \\ &+ \operatorname{sen}\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\operatorname{sen}\left(\vec{k}_2 \cdot \vec{r} + \varepsilon_2\right)\operatorname{sen}^2(\omega t) + \operatorname{sen}\left(\vec{k}_1 \cdot \vec{r} + \varepsilon_1\right)\cos\left(\vec{k}_2 \cdot \vec{r} + \varepsilon_2\right)\cos(\omega t)\operatorname{sen}(\omega t)\end{aligned}\right]\end{aligned} \tag{2.16b}$$

remembering that the time average of a function $f(t)$, over the interval T, is

$$\langle f(t)\rangle = \frac{1}{T}\int_t^{t+T} f(t')dt' \tag{2.17}$$

The period τ of the harmonic function is $\frac{2\pi}{\omega}$, and for the actual case $T >> \tau$, in this case, the coefficient $\frac{1}{T}$ in front of the integral has a dominant effect

Taking the average in the time on the equation (2.16b), First calculating:

$$\left\langle \cos^2(\omega t)\right\rangle = \frac{1}{T}\int_t^{t+T} \cos^2(\omega t')dt'$$

using the trigonometric identity

$$\cos^2 A = \frac{1}{2} + \frac{1}{2}\cos 2A$$

$$\left\langle \cos^2(\omega t)\right\rangle = \frac{1}{T}\int_t^{t+T}\left(\frac{1}{2} + \frac{1}{2}\cos(2\omega t')\right)dt'$$

$$= \frac{1}{2T}\int_t^{t+T} dt' + \frac{1}{2T}\int_t^{t+T}\cos(2\omega t')dt'$$

$$= \frac{1}{2T}(t+T-t) + \frac{1}{2T}\frac{1}{2\omega}\left[\text{sen}(2\omega t')\right]_t^{t+T}$$

$$= \frac{T}{2T} + \frac{1}{2T}\frac{1}{\frac{4\pi}{\tau}}\left[\text{sen}\left(\frac{4\pi}{\tau}(t+T)\right) - \text{sen}\left(\frac{4\pi}{\tau}t\right)\right]$$

$$= \frac{1}{2} + \frac{\tau}{8\pi T}\left[\text{sen}\left(\frac{4\pi}{\tau}(t+T)\right) - \text{sen}\left(\frac{4\pi}{\tau}t\right)\right]$$

$$\left\langle \cos^2(\omega t)\right\rangle = \frac{1}{2} \tag{2.18a}$$

after this,

$$\left\langle \text{sen}^2(\omega t)\right\rangle = \frac{1}{T}\int_t^{t+T} \text{sen}^2(\omega t')dt'$$

using the trigonometric identity

$$\text{sen}^2 A = \frac{1}{2} - \frac{1}{2}\cos 2A$$

$$\left\langle \text{sen}^2(\omega t)\right\rangle = \frac{1}{T}\int_t^{t+T}\left(\frac{1}{2} - \frac{1}{2}\cos(2\omega t')\right)dt'$$

$$= \frac{1}{2T}\int_t^{t+T} dt' - \frac{1}{2T}\int_t^{t+T}\cos(2\omega t')dt'$$

$$= \frac{1}{2T}(t+T-t) - \frac{1}{2T}\frac{1}{2\omega}\left[\text{sen}(2\omega t')\right]_t^{t+T}$$

$$= \frac{T}{2T} - \frac{1}{2T}\frac{1}{\frac{4\pi}{\tau}}\left[\text{sen}\left(\frac{4\pi}{\tau}(t+T)\right) - \text{sen}\left(\frac{4\pi}{\tau}t\right)\right]$$

$$= \frac{1}{2} - \frac{\tau}{8\pi T}\left[\text{sen}\left(\frac{4\pi}{\tau}(t+T)\right) - \text{sen}\left(\frac{4\pi}{\tau}t\right)\right]$$

$$\left\langle \text{sen}^2(\omega t)\right\rangle = \frac{1}{2} \tag{2.18b}$$

and finally,

$$\langle \cos(\omega t)\,\mathrm{sen}(\omega t)\rangle = \frac{1}{T}\int_t^{t+T} \cos(\omega t')\,\mathrm{sen}(\omega t')\,dt'$$

using the trigonometric identity

$$\mathrm{sen}\,A\cos B = \frac{1}{2}\{\mathrm{sen}(A-B)+\mathrm{sen}(A+B)\}$$

$$\langle \cos(\omega t)\,\mathrm{sen}(\omega t)\rangle = \frac{1}{T}\int_t^{t+T} \frac{1}{2}\{\mathrm{sen}(\omega t'-\omega t')+\mathrm{sen}(2\omega t')\}dt'$$

$$= \frac{1}{2T}\int_t^{t+T} \mathrm{sen}(0)\,dt' + \frac{1}{2T}\int_t^{t+T} \mathrm{sen}(2\omega t')\,dt'$$

$$= \frac{-1}{2T}\frac{1}{2\omega}\left[\cos(2\omega t')\right]_t^{t+T}$$

$$= \frac{-1}{4T\dfrac{2\pi}{\tau}}\left[\cos\left(\frac{4\pi}{\tau}t'\right)\right]_t^{t+T}$$

$$\langle \cos(\omega t)\,\mathrm{sen}(\omega t)\rangle = \frac{-\tau}{8\pi T}\left[\cos\left(\frac{4\pi}{\tau}(t+T)\right)-\cos\left(\frac{4\pi}{\tau}(t)\right)\right]$$

$$\langle \cos(\omega t)\,\mathrm{sen}(\omega t)\rangle = 0 \qquad (2.18c)$$

making the substitution (2.18a), (2.18b) and (2.18c) on (2.16b) using (2.17), we have

$$\langle \vec{E}_1\cdot\vec{E}_2\rangle = \vec{E}_{o1}\cdot\vec{E}_{o2}\left\{\frac{1}{2}\cos(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\cos(\vec{k}_2\cdot\vec{r}+\varepsilon_2)+\frac{1}{2}\mathrm{sen}(\vec{k}_1\cdot\vec{r}+\varepsilon_1)\mathrm{sen}(\vec{k}_2\cdot\vec{r}+\varepsilon_2)\right\}$$

$$\langle \vec{E}_1\cdot\vec{E}_2\rangle = \frac{1}{2}\vec{E}_{o1}\cdot\vec{E}_{o2}\cos(\vec{k}_1\cdot\vec{r}+\varepsilon_1-\vec{k}_2\cdot\vec{r}-\varepsilon_2)$$

From equation (2.14), the interference term is then:

$$\boxed{I_{12} = \vec{E}_{o1}\cdot\vec{E}_{o2}\cos\delta} \qquad (2.21)$$

where δ is equal to $(\vec{k}_1\cdot\vec{r}-\vec{k}_2\cdot\vec{r}+\varepsilon_1-\varepsilon_2)$, is the phase difference arising from a combined path-length and initial phase-angle difference.

If $\vec{E}_{o1}$ is parallel to $\vec{E}_{o2}$ in this case, the irradiance reduces to the value found in the scalar treatment.

$$\boxed{I_{12} = E_{o1}E_{o2}\cos\delta} \tag{2.21}$$

rewriting I_1 and I_2,

$$\boxed{I_1 = \left\langle E_1^2 \right\rangle = \frac{E_{o1}^2}{2}} \tag{2.19}$$

$$\boxed{I_2 = \left\langle E_2^2 \right\rangle = \frac{E_{o2}^2}{2}} \tag{2.20}$$

The interference term can be expressed as

$$\boxed{I_{12} = 2\sqrt{I_1 I_2}\cos\delta} \tag{2.23}$$

where the total irradiance is:

$$\boxed{I = I_1 + I_2 + 2\sqrt{I_1 I_2}\cos\delta} \tag{2.24}$$

Using this relations, we can calculate the interference, i.e. distribution of light, on any point in the space. From (2.24) and choosing,

$$I_1 = I_2 = I_o$$

$$I = 2I_o + 2I_o\cos\delta$$

$$= 2I_o\left(1+\cos\delta\right)$$

with the trigonometric identity $\cos^2 A = \frac{1}{2} + \frac{1}{2}\cos(2A)$

$$2\cos^2 A = 1 + \cos(2A)$$

$$\text{with } 2A = \delta \;\Rightarrow\; A = \frac{\delta}{2}$$

$$I = 4I_o \cos^2\left(\frac{\delta}{2}\right) \tag{2.25}$$

Solving to get the total irradiance I on a plane $\sum_2$ at a distance d from a parallel plane $\sum_1$ that contain on the z-axis both point sources S_1, S_2. We consider that $d >> \lambda$ in order to have the interference of two plane waves. This can be seen in figure 2.2,

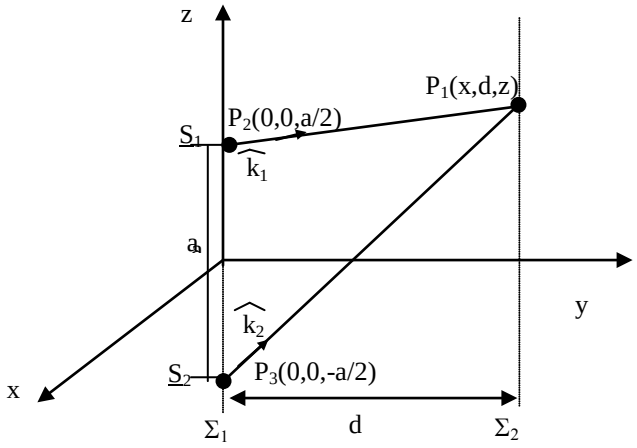


Figure 2.2 Geometrical construction of two interfering plane wave-fronts.

To define the straight line that goes from P_1 to P_2 and $\vec{P}_1$, $\vec{P}_2$ are the vectors to that points respectively. We can define parametrically the equation of a straight line,

$$\vec{l}(t) = \hat{a}t + \vec{b} \ .$$

For our description,

$$\frac{\left(\vec{P}_1 - \vec{P}_2\right)}{\left|\vec{P}_1 - \vec{P}_2\right|} t = \left(\vec{P} - \vec{P}_1\right)$$

we make the following relations,

$$\hat{a}_1 = \frac{\vec{P}_1 - \vec{P}_2}{\left|\vec{P}_1 - \vec{P}_2\right|}$$

$$\vec{b} = \vec{P}_1$$

$$\vec{l} = \vec{P} \ ,$$

where,

$$\vec{P} = (x, y, z)$$

as we know $\vec{k}_1 = |\vec{k}_1|\hat{k}_1$ then $\hat{a}_1 = \hat{k}_1$ that is a unit vector in the direction of the straight line P_1P_2. It's cosine directors are:

$$\cos\alpha_1 = \frac{x-0}{|\vec{P}_1 - \vec{P}_2|} \qquad \cos\beta_1 = \frac{d-0}{|\vec{P}_1 - \vec{P}_2|} \qquad \cos\gamma_1 = \frac{z-\left(\frac{a}{2}\right)}{|\vec{P}_1 - \vec{P}_2|} \tag{2.26}$$

where $|\vec{P}_1 - \vec{P}_2| = \sqrt{(x-0)^2 + (d-0)^2 + \left(z - a/2\right)^2}$ (2.27)

To define the straight line that goes from P_1 to P_3 and $\vec{P}_1$, $\vec{P}_3$ are the vectors to that points respectively. Analogous to the upper development. For our description,

$$\frac{(\vec{P}_1 - \vec{P}_3)}{|\vec{P}_1 - \vec{P}_3|} t = (\vec{P} - \vec{P}_1)$$

we establish the last relation needed,

$$\hat{a}_2 = \frac{\vec{P}_1 - \vec{P}_3}{|\vec{P}_1 - \vec{P}_3|}$$

as we know $\vec{k}_2 = |\vec{k}_2|\hat{k}_2$ then $\hat{a}_2 = \hat{k}_2$ that is a unit vector in the direction of the straight line P_1P_3. It's cosine directors are:

$$\cos\alpha_2 = \frac{x-0}{|\vec{P}_1 - \vec{P}_3|} \qquad \cos\beta_2 = \frac{d-0}{|\vec{P}_1 - \vec{P}_3|} \qquad \cos\gamma_2 = \frac{z-\left(-\frac{a}{2}\right)}{|\vec{P}_1 - \vec{P}_3|} \tag{2.28}$$

where $|\vec{P}_1 - \vec{P}_3| = \sqrt{(x-0)^2 + (d-0)^2 + \left(z + a/2\right)^2}$ (2.29)

We know that δ in the equation (23) is

$$\delta = \vec{k}_1 \cdot \vec{r} - \vec{k}_2 \cdot \vec{r} + \varepsilon_1 - \varepsilon_2 \tag{2.30}$$

where $\vec{r} = (x, y, z)$

and

$$\vec{k}_1 = \frac{2\pi}{\lambda}\hat{k}_1 \tag{2.31}$$

$$\hat{k}_1 = (\cos\alpha_1, \cos\beta_1, \cos\gamma_1) \tag{2.32}$$

$$\vec{k}_2 = \frac{2\pi}{\lambda}\hat{k}_2 \tag{2.33}$$

$$\hat{k}_2 = (\cos\alpha_2, \cos\beta_2, \cos\gamma_2) \tag{2.34}$$

considering the case when the initial phase on each one point sources is equal to zero, i.e. $\varepsilon_1 = 0$ and $\varepsilon_2 = 0$, or $\varepsilon_1 = \varepsilon_2 \neq 0$. Using (2.25), (2.26), (2.27), (2.28), (2.29), (2.30), (2.31), (2.32), (2.33), (2.34) can we calculate the intensity pattern on the plane that is y-direction at d distance from the origin of the coordinate system. This pattern is produced by the interference of two plane wave-fronts produced by two point sources symmetrically displaced from the origin a distance $\pm\frac{a}{2}$ on the z-axis, so on the intensity distribution is function of x and z, i.e. $f(x,z)$.

2.6 Basics of the Lateral shearing interferometer (BLSI).

The waves with spatial dependence can be represented as

$$\psi_1(\vec{r}) = A_1(\vec{r})e^{i\phi(r)}, \tag{2.35}$$

and the same wave with a displacement,

$$\psi_2(\vec{r} - \vec{D}_o) = A_2(\vec{r})e^{i\phi(\vec{r}-\vec{D}_o)}, \tag{2.36}$$

where $\phi(\vec{r})$ is the phase function of the wave.

$\vec{r} = x\hat{i} + y\hat{j} + z\hat{k}$.

$\vec{D}_o = \delta_{x_o}\hat{i} + \delta_{y_o}\hat{j} + \delta_{z_o}\hat{k}$.

The coherent addition of this waves is,

$$\begin{aligned}\psi &= \psi_1 + \psi_2 \\ &= A_1(\vec{r})e^{i\phi(\vec{r})} + A_2(\vec{r})e^{i\phi(\vec{r}-\vec{D}_o)},\end{aligned} \tag{2.37}$$

calculating the irradiance,

$$\begin{aligned}I = \psi\psi^* &= (\psi_1 + \psi_2)(\psi_1^* + \psi_2^*) \\ &= \left(A_1(\vec{r})e^{i\phi(\vec{r})} + A_2(\vec{r})e^{i\phi(\vec{r}-\vec{D}_o)}\right)\left(A_1(\vec{r})e^{-i\phi(\vec{r})} + A_2(\vec{r})e^{-i\phi(\vec{r}-\vec{D}_o)}\right) \\ &= A_1^2(\vec{r}) + A_2^2(\vec{r}) + A_1(\vec{r})A_2(\vec{r})e^{i\phi(\vec{r})}e^{-i\phi(\vec{r}-\vec{D}_o)} + A_1(\vec{r})A_2(\vec{r})e^{-i\phi(\vec{r})}e^{i\phi(\vec{r}-\vec{D}_o)} \\ &= A_1^2(\vec{r}) + A_2^2(\vec{r}) + A_1(\vec{r})A_2(\vec{r})\left[e^{i[\phi(\vec{r})-\phi(\vec{r}-\vec{D}_o)]} + e^{-i[\phi(\vec{r})-\phi(\vec{r}-\vec{D}_o)]}\right] \\ &= A_1^2(\vec{r}) + A_2^2(\vec{r}) + 2A_1(\vec{r})A_2(\vec{r})\cos(\phi(\vec{r}) - \phi(\vec{r} - \vec{D}_o)),\end{aligned} \tag{2.38}$$

making $I_1(\vec{r}) = A_1^2(\vec{r})$ and $I_2(\vec{r}) = A_2^2(\vec{r})$ then

$$I = I_1(\vec{r}) + I_2(\vec{r}) + 2\sqrt{I_1(\vec{r})I_2(\vec{r})}\cos\left(\phi(\vec{r}) - \phi(\vec{r} - \vec{D}_o)\right). \qquad (2.39)$$

The equation (2.39) is a model of the interference pattern obtained by a lateral shearing interferometer ideal.

If we define the difference of phase as (of 2.39):

$$\Delta\phi(\vec{r}) = k\Delta W_{\vec{r}}(\vec{r}) = k\left[W(\vec{r}) - W(\vec{r} - \vec{D}_o)\right], \qquad (2.40)$$

we obtain maximums when

$$\frac{2\pi}{\lambda}\Delta W_{\vec{r}}(\vec{r}) = 2\pi m, \qquad n = 0,\pm1,\pm2,...$$

$$\Delta W_{\vec{r}}(\vec{r}) = n\lambda. \qquad n = 0,\pm1,\pm2,... \qquad (2.41)$$

2.7 Conclusions.

We have presented the theoretical basis of the interference phenomenon. So that it is possible calculate the irradiance distribution on any point in the space if we know the characteristics of the two light sources that superpose in the space. The characteristics before said involve coherence, intensity and relative phase between the sources, geometrical distribution of the sources and the "observation point."

References.

Born, Max and Emil Wolf, *Principles of Optics*, 6ª ed., Pergamon Press, England, 1993.
Hecht, Eugene, *Optics*, 2ed. Addison –Wesley, USA, 1990.

3 Lateral Shear Interferometer.

3.0 Objectives of this Chapter.

This chapter explores the characteristics of the Lateral Shear Interferometer (LSI) like different transformations over the wave-front under analysis, tilt, scale, lateral shear, some of this transformation are wished and other aren't; also is studied the frequency response of LSI considering infinite and finite pupils. Lateral shear is limited according to the coherence properties of the light source in LSI. Closed forms of the third order aberrations in the LSI are shown and their approximations are developed when the lateral shear tends to cero. The Murty's interferometer is studied carefully, in order to calculate every part needed to built it. Some analysis is shown to evaluate deviations of the ideal model of the LSI with the actual one. Also we apply some properties of Linear and spatial invariant to the LSI with infinite pupil. Finally is analyzed the sensitivity of the LSI to the lateral shear.

3.1 Introduction.

The interferometers with absolute reference require a perfect wave-front, without a deformation, as a reference to determine a unknown wave-front. Produce this perfect wave-front spherical or plane, require perfect lenses i.e., the mirrors and beam splitter in a Twyman-Green interferometer must be absolutely planes.

There is an alternate option to avoid use a perfect reference wave-front. Under specific conditions, this can be obtained that the reference wave-front to be identical to the wave-front under analysis. Although, if the reference wave-front and the unknown wave-front are identical in all the aspects, will not exist interference fringes. As is shown in Figure 3.1.

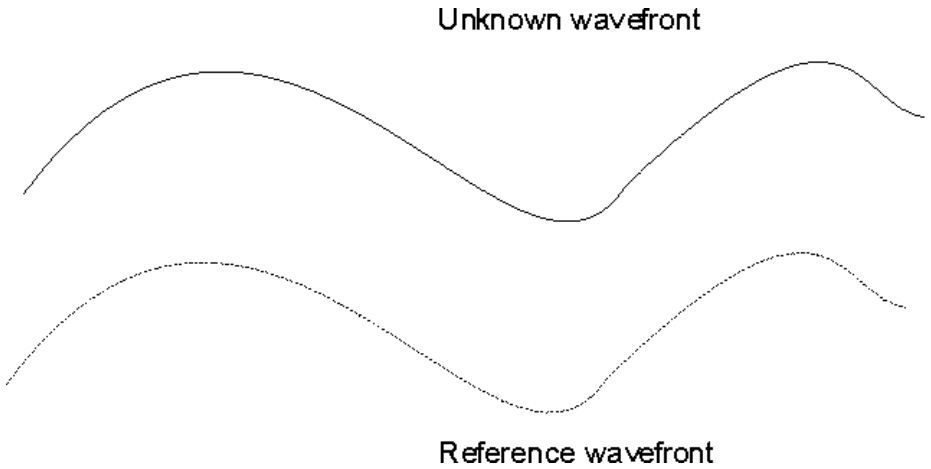


Figure 3.1. Reference wave-front and unknown wave-front identical.

In order to obtain interference fringes is enough produce some relative transformation on the reference wave-front. For example:

a) Tilt.
b) Scale.
c) Lateral shear.

We will interest on Lateral shear, the tilt and scale will be analyzed as an error produced by the experimental optical arrangement.

The lateral shear interferometers same than the others interferometers without an absolute reference produce two identical wave-fronts. Only one of them displaced laterally respect the other as can be seen in Figure 3.2. In this figure the incident wave-front and the two that go out from interferometer are collimated. In this interferometers, as is shown in Figure 3.3 the wave-fronts that come in and that go out can be collimated, convergent or divergent.

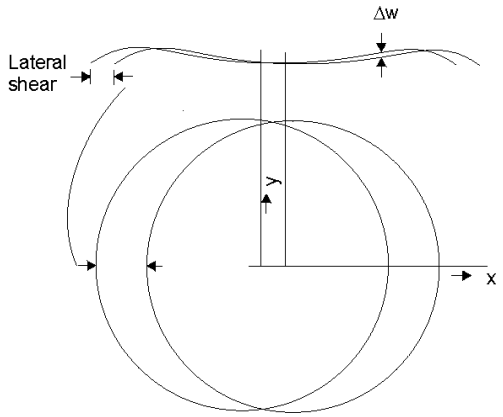
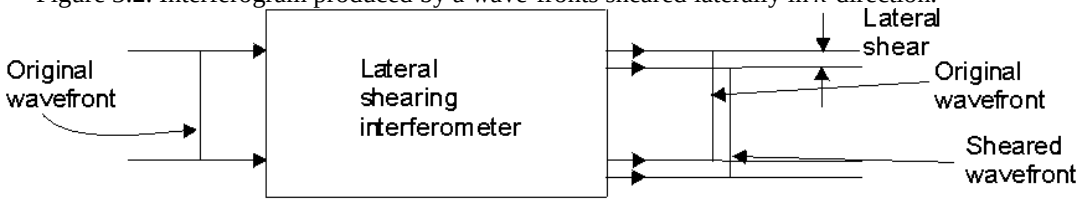


Figure 3.2. Interferogram produced by a wave-fronts sheared laterally in *x*-direction.



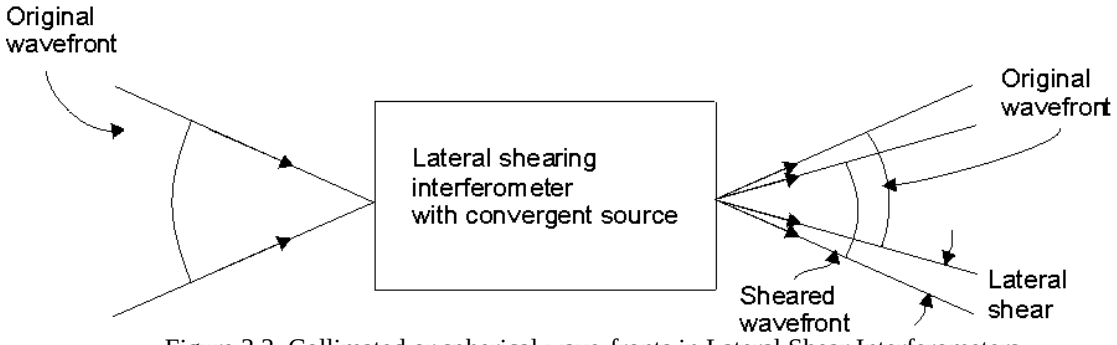


Figure 3.3. Collimated or spherical wave-fronts in Lateral Shear Interferometers.

The lateral shear interferometer was invented by Bates on 1947 since then has appeared big quantity of interferometers with different optical arrangements, different characteristics, using ordinary sources of light and lasers.

3.2 Considerations regarding coherence properties of the light source.

Figure 3.4 schematically illustrates the arrangement of a lateral shearing interferometer in which a shear takes place for a nearly plane wave-front obtained from the collimating lens. Let the full width of the wave-front be *d* and the amount of lateral shear X_o. If the focal length of the collimating lens used is *f*, there is full spatial coherence across the width of the wave-front when the size of the source is equal to the width of the central diffraction maximum (Airy disk) corresponding to the *f*-number of the particular collimating lens.

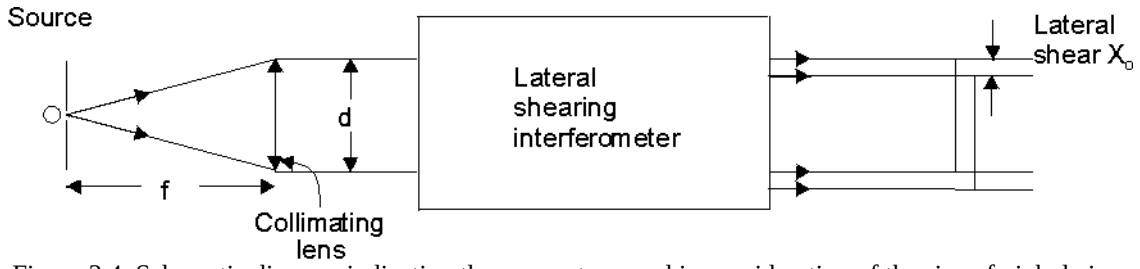


Figure 3.4. Schematic diagram indicating the parameters used in consideration of the size of pinhole in a Lateral Shear Interferometer.

The circular aperture of radius *a*, by diffraction can be calculated the Airy disk as is shown in Figure 3.5 where $\sum$ is the circular aperture screen and σ is the plane of the lens. The radius (*r*) of Airy disk is given by

$$r = 1.22\frac{\lambda R}{2a} \tag{3.1}$$

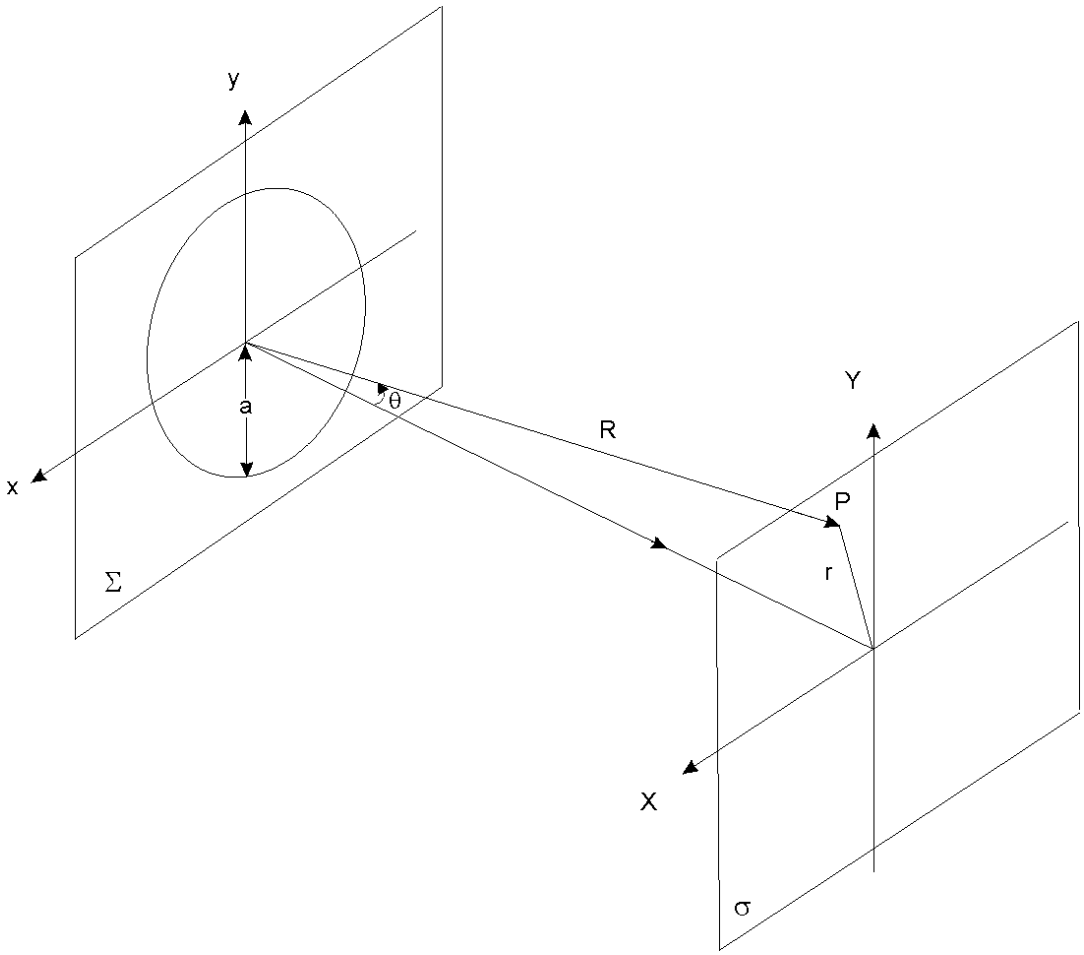


Figure 3.5. Opening with circular geometry.

If we want that 2*r* (Airy disk) to be equal to the diameter (*d*) of a lens when it is place in front of pinhole at focal length (*f*), as is shown in Figure 3.6. Rewriting equation (3.1),

$$2\left[r = 1.22\frac{\lambda f}{2a} \right]$$

$$2r = 2.44\frac{\lambda f}{D}$$

$$d = 2.44\frac{\lambda f}{D}$$

$$D = 2.44\frac{\lambda f}{d}$$

$$\boxed{D \approx \frac{\lambda f}{d}} \qquad (3.2)$$

$$D \approx \lambda\left(f/\#\right)$$

where $D = 2a$ is the diameter of pinhole.
$f/\#$ should be understood as a single symbol.

Thus the order of magnitude of the size of the pinhole to be used over the source to achieve spatial coherence is given by $\lambda f / d$, where λ is the wavelength of the particular spectral line of the source that is to be used. However, since the lateral shear is X_o, the spatial coherence should be sufficient so that interference can be observed between parts of the wave-front separated by the distance X_o, which is less than d. Hence the source (pinhole) size can be $\left(\lambda f / d\right)\left(d / X_o\right) = \left(\lambda / X_o\right) f$. Thus the pinhole size chosen is some multiple of the diffraction-limited pinhole size.

Here is necessary to explain that lateral shear X_o is not obligatory in x-direction, it could be whatever direction, but by simplicity in own explanation we use x-direction without loss of generality.

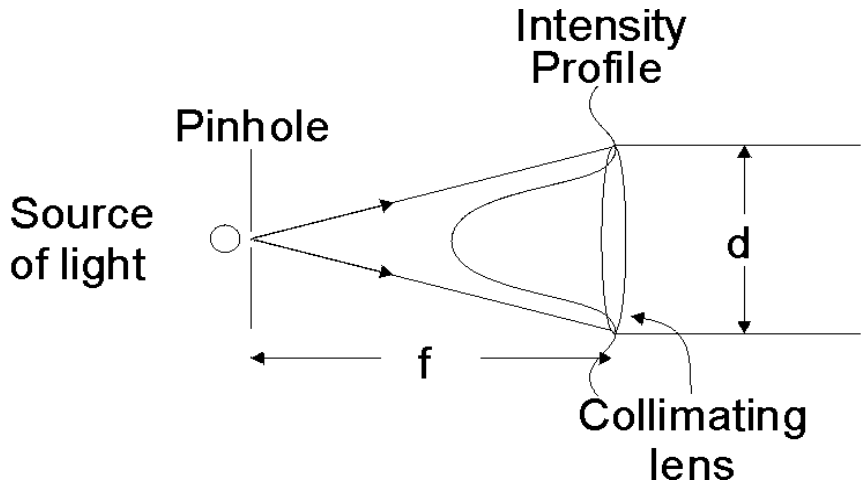


Figure 3.6. Airy disk produced the pinhole coincident with the diameter *d* of the lens placed at its focal length from pinhole.

3.3 Theory of Lateral Shearing Interferometry.

Figure 3.7 shows the original wave-front and also the laterally sheared wave-front. The wave-front is considered nearly plane so that wave-front errors may be small deviations from this plane. The wave-front error may be expressed as $W(x,y)$, where (x,y) are the coordinates of the point P. When this wave-front is sheared in the x-direction by an amount X_o, the error at the same point for the sheared wave-front is $W(x-X_o,y)$. The resulting optical path difference (OPD) ΔW_x at P between the original and the sheared wave-front is $W(x,y)-W(x-X_o,y)$. Thus, in the lateral shearing interferometry, it is the quantity, ΔW_x, that is determined, and when X_o is zero, there is no path difference anywhere in the common area of the wave-fronts and consequently no error can be seen, however large it may be.

$$\Delta W_x(x,y) = OPD = W(x,y) - W(x-X_o,y) \qquad (x,y) \in common\ area \tag{3.3}$$

Now remembering the definition of one dimension differentiation

$$\frac{df(x)}{dx} = lim_{X_o \to 0} \frac{f(x) - f(x-X_o)}{X_o} \tag{3.4}$$

Using equation (3.4) on equation (3.3) for a function of two variables.

$$lim_{X_o \to 0} X_o \left(\frac{\partial W(x,y)}{\partial x} \right) = lim_{X_o \to 0} W(x,y) - W(x-X_o,y) \qquad (x,y) \in common\ area \tag{3.5}$$

under the condition $X_o \to 0$, we can approximate $\Delta W_x(x,y)$ equation (3.3) as $X_o \left(\frac{\partial W(x,y)}{\partial x} \right)$ equation (3.5).

$$\Delta W_x(x,y) \approx X_o \left(\frac{\partial W(x,y)}{\partial x} \right) \tag{3.6}$$

The equation (3.6) becomes more exact as $X_o \to 0$, but we also have seen that the sensitivity decrease as $X_o \to 0$. Thus we must arrive at some compromise for the proper value of X_o if equation (3.6) is to be used exactly. This approximation is equivalent to expand equation (3.3) in a Taylor series.

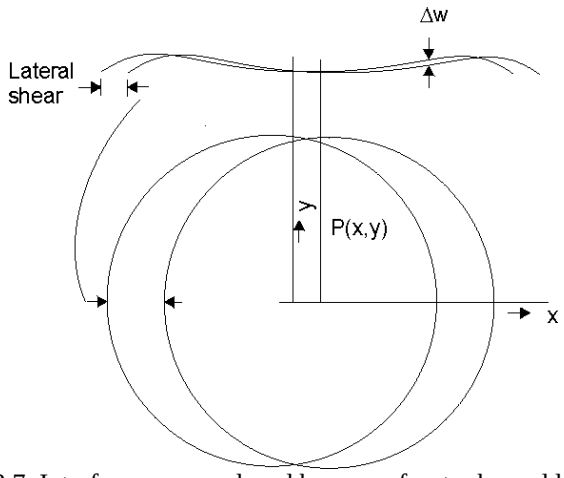


Figure 3.7. Interferogram produced by wave-fronts sheared laterally.

In the practice the condition for $X_o \to 0$ is that the wave-front slopes in the *x*-direction (displacement direction) may be considered almost constant in an interval X_o. This is equivalent to the condition that the fringe spatial frequency in the x-direction is almost constant in an interval X_o. If this condition is satisfied we can use precisely (3.6) then we can say that a lateral shear interferometer does not measure the wave-front deformation $W(x,y)$ in a direct manner, but its slope or transverse aberration in the direction of the lateral shear, equation (3.6a). To measure the two components of the transverse aberrations it is necessary to take two lateral-sheared interferograms in perpendicular directions.

$$OPD = X_o\left(\frac{\partial W(x,y)}{\partial x}\right) = \frac{TA_x(x,y)}{r} \qquad (3.6a)$$

where, $TA_x(x,y)$ is the transverse aberration of the ray perpendicular to the wave-front, measured at a plane containing the center of curvature of the wave-front.

3.3.1 Aberration polynomial for primary aberrations of third-order in a Lateral Shear Interferometer.

The aberration polynomial for primary aberrations of third-order can be written in *x* and *y* variables as:

$$W(x,y) = A\left(x^2+y^2\right)^2 + By\left(x^2+y^2\right) + C\left(x^2+3y^2\right) + D\left(x^2+y^2\right) + Ey + Fx + G$$

where:
A = spherical aberration coefficient.
B = coma coefficient.
C = astigmatism coefficient.
D = defocusing coefficient.

E = tilt about the x axis.
F = tilt about the y axis.
G = constant or piston term.

To introduce a displacement X_0 in the x-direction, we will have

$$W((x-X_0),y) = A\left((x-X_0)^2+y^2\right)+By\left((x-X_0)^2+y^2\right)+C\left((x-X_0)^2+3y^2\right) + D\left((x-X_0)^2+y^2\right)+Ey+F(x-X_0)+G$$

Similarly for a displacement Y_0 in the y-direction, we will have

$$W(x,(y-Y_0)) = A\left(x^2+(y-Y_0)^2\right)+B(y-Y_0)\left(x^2+(y-Y_0)^2\right)+C\left(x^2+3(y-Y_0)^2\right) + D\left(x^2+(y-Y_0)^2\right)+E(y-Y_0)+Fx+G$$

In the following we will study each aberration for a lateral shearing in x-direction.

In the case of *spherical aberration* alone,

$$\begin{aligned}
\Delta W_x(x,y) &= OPD = W(x,y)-W(x-X_0,y)\\
&= A\left(x^2+y^2\right)^2 - A\left[(x-X_0)^2+y^2\right]^2\\
&= A\left(x^4+2x^2y^2+y^4\right)-A\left[(x-X_0)^4+2(x-X_0)^2y^2+y^4\right]\\
&= A\left(x^4+2x^2y^2+y^4\right)\\
&\quad -A\left[\left(x^4-4x^3X_0+6x^2X_0^2-4x_0^3+X_0^4\right)^4+2x^2y^2-4xX_0^2y^2+2X_0^2y^2+y^4\right]\\
\Delta W_x(x,y) &= A\left[4x^3X_0-6x^2X_0^2+4xX_0^3-X_0^4+4xX_0y^2-2X_0^2y^2\right]
\end{aligned}$$

making the approximation $X_0 \to 0$

$$\begin{aligned}
\Delta W_x(x,y) &\approx X_0\left(\frac{\partial W(x,y)}{\partial x}\right)\\
&\approx AX_0(2)\left(x^2+y^2\right)(2x)\\
\Delta W_x(x,y) &\approx 4AX_0x^3+4AX_0xy^2
\end{aligned}$$

In the case of *coma aberration* alone,

$$\begin{aligned}
\Delta W_x(x,y) &= OPD = W(x,y)-W(x-X_0,y)\\
&= By\left(x^2+y^2\right)-By\left[(x-X_0)^2+y^2\right]\\
&= By\left(x^2+y^2\right)-By\left[x^2-2xX_0+X_0^2+y^2\right]\\
\Delta W_x(x,y) &= By\left[2xX_o-X_o^2\right]
\end{aligned}$$

making the approximation $X_0 \to 0$

$$\begin{aligned}\Delta W_x(x,y) &\approx X_0\left(\frac{\partial W(x,y)}{\partial x}\right)\\ &\approx X_0 B(2xy)\\ \Delta W_x(x,y) &\approx 2ByxX_0\end{aligned}$$

In the case of *astigmatism aberration* alone,

$$\begin{aligned}\Delta W_x(x,y) &= OPD = W(x,y) - W(x - X_0, y)\\ &= C\left(x^2 + 3y^2\right) - C\left[(x - X_0)^2 + 3y^2\right]\\ &= C\left(x^2 + 3y^2\right) - C\left[x^2 - 2xX_0 + X_0^2 + 3y^2\right]\\ \Delta W_x(x,y) &= C\left[2xX_0 - X_0^2\right]\end{aligned}$$

making the approximation $X_0 \to 0$

$$\begin{aligned}\Delta W_x(x,y) &\approx X_0\left(\frac{\partial W(x,y)}{\partial x}\right)\\ &\approx X_0 C(2x)\\ \Delta W_x(x,y) &\approx 2CxX_0\end{aligned}$$

In the case of *defocusing* alone,

$$\begin{aligned}\Delta W_x(x,y) &= OPD = W(x,y) - W(x - X_0, y)\\ &= D\left(x^2 + y^2\right) - D\left[(x - X_0)^2 + y^2\right]\\ &= D\left(x^2 + y^2\right) - D\left[x^2 - 2xX_0 + X_0^2 + y^2\right]\\ \Delta W_x(x,y) &= D\left[2xX_0 - X_0^2\right]\end{aligned}$$

making the approximation $X_0 \to 0$

$$\begin{aligned}\Delta W_x(x,y) &\approx X_0\left(\frac{\partial W(x,y)}{\partial x}\right)\\ \Delta W_x(x,y) &\approx 2DxX_0\end{aligned}$$

In the case of *tilt in x -direction* alone,

$$\begin{aligned}\Delta W_x(x,y) &= OPD = W(x,y) - W(x - X_0, y)\\ &= Ey - Ey\\ \Delta W_x(x,y) &= 0\end{aligned}$$

making the approximation $X_0 \to 0$

$$\Delta W_x(x,y) \approx X_0\left(\frac{\partial W(x,y)}{\partial x}\right)$$
$$\Delta W_x(x,y) \approx 0$$

In the case of *tilt in y -direction* alone,

$$\Delta W_x(x,y) = OPD = W(x,y) - W(x - X_0, y)$$
$$= Fx - F(x - X_0)$$
$$\Delta W_x(x,y) = FX_0$$

making the approximation $X_0 \to 0$

$$\Delta W_x(x,y) \approx X_0\left(\frac{\partial W(x,y)}{\partial x}\right)$$
$$\Delta W_x(x,y) \approx FX_0$$

3.4 Some kinds of lateral shear interferometers.

A very common lateral shear interferometer that uses a ordinary source of light is shown in Figure 3.8. This optical arrangement is based on the Mach-Zehnder interferometer, with a little tilt in one of the mirrors. This interferometer uses a monochromatic source of light of metallic vapor, i.e. mercury vapor or sodium vapor.

The Michelson's interferometer has been used as the base to built lateral shear interferometers. As is shown in Figure 3.9.

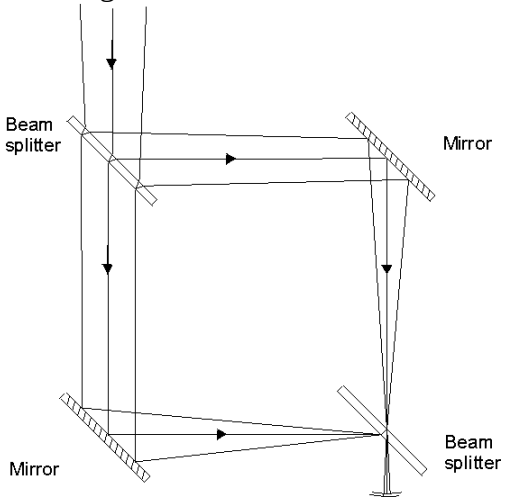


Figure 3.8. Lateral Shear Interferometer based on the Mach-Zehnder interferometer.

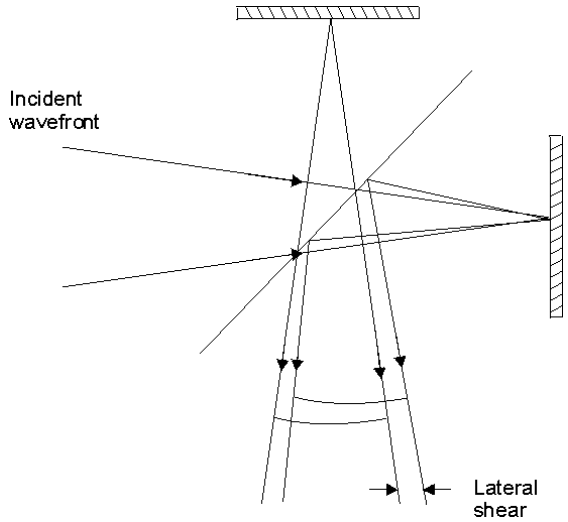


Figure 3.9. Lateral Shear Interferometer based on the Michelson's interferometer.

Others interferometers use the excellent properties of coherence of the lasers to produce the lateral shearing interferometry. There are numerous and interesting interferometers, but the most common interferometer is the Murty's type.

3.5 Murty's interferometer.

3.5.1 Introduction.

There are big variety of lateral shear interferometers. Some of them work with white light, others require partially monochromatic light, the last ones require absolutely monochromatic light. In other words, the necessary temporal coherence is wide. This produces a very wide optical variety. Maybe the most popular lateral shear interferometer due to their simplicity is the Murty´s interferometer. This is shown in Figure 3.10.

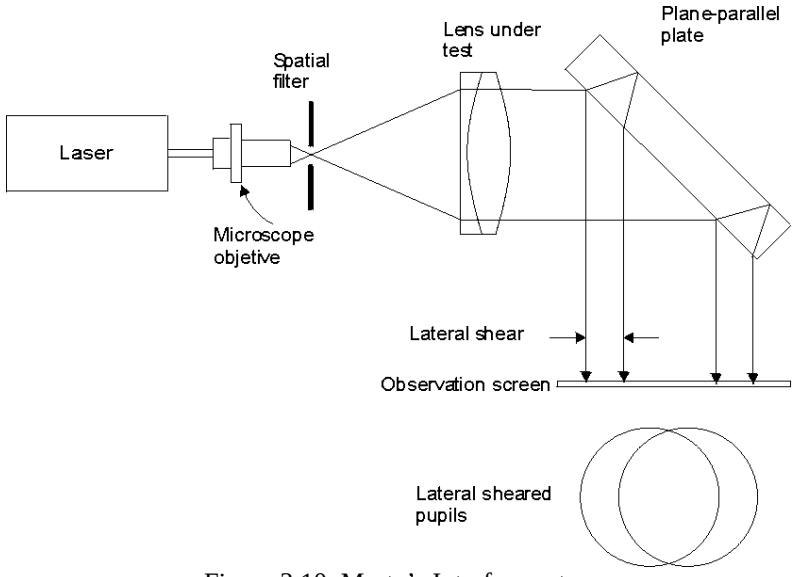


Figure 3.10. Murty's Interferometer.

As we can see in this figure, only is required a good collimating lens and a glass plate with the faces plane parallels. This simplicity produces an interferometer cheap and easy to built, very stable and easy to align. By this simplicity, the interferometer does not have compensation in their optical paths, due to the plate width, and is necessary use laser light.

3.5.2 Ideal Model.

When a ray is incident on the glass plate one part is reflected on the first face and the other refracted and reflected on the second face. The first face is partially reflecting and the second face has 100% reflectivity.

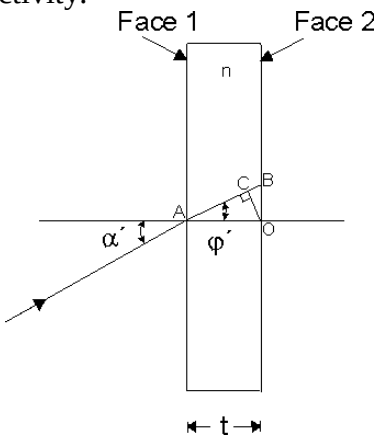


Figure 3.11. Geometry of a plane parallel plate used to generate the lateral shear X_o of a incident ray.

To calculate the lateral displacement X_o of a ray that has an incidence angle α' on the first face of the glass plate. The glass plate has a width t, the refractive index on the wavelength used ($\lambda = 632.8nm$) is $n = 1.5$. We use the geometry shown in the Figure 3.11.,

$$\begin{aligned}
\overline{OB} &= t\,\mathrm{tg}\,\varphi' \\
\overline{OC} &= \overline{OB}\cos\alpha' \\
\overline{OC} &= \left[t\,\mathrm{tg}\,\varphi'\right]\cos\alpha' \\
&= t\frac{\mathrm{sen}\,\varphi'}{\cos\varphi'}\cos\alpha' \\
&= \frac{t\,\mathrm{sen}\,\alpha'}{n\cos\varphi'}\cos\alpha' \\
&= \frac{t\,\mathrm{sen}\,\alpha'}{n}\left[\frac{\cos\alpha'}{\cos\varphi'}\right] \\
&= \frac{t\,\mathrm{sen}\,\alpha'}{n}\left[\frac{\sqrt{1-\mathrm{sen}^2\,\alpha'}}{\sqrt{1-\mathrm{sen}^2\,\varphi'}}\right]
\end{aligned}$$

making $s = \frac{1}{n}\operatorname{sen}\alpha'$ (3.7)

$$\overline{OC} = \frac{ts\sqrt{1-(ns)^2}}{\sqrt{1-\frac{\operatorname{sen}^2\alpha'}{n^2}}}$$

$$\overline{OC} = \frac{ts\sqrt{1-(ns)^2}}{\sqrt{1-s^2}}$$

after the reflection on the second face, the distance $\overline{OC}$ is twice, then

$$\boxed{X_o = \frac{2ts\sqrt{1-(ns)^2}}{\sqrt{1-s^2}}} \tag{3.8}$$

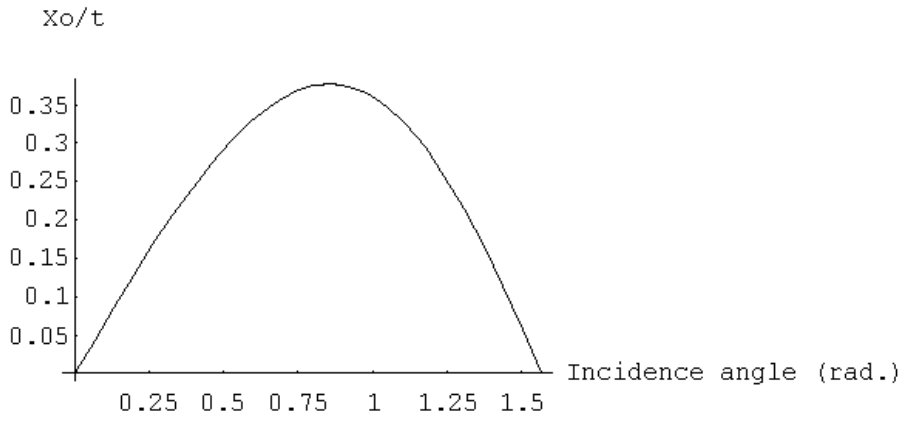


Figure 3.12. Graph of lateral shearing VS incident angle of the wave-front in a plane parallel plate.

As is obvious the lateral displacement is function of incidence angle and the width glass plate as can be seen in equation (3.8). This relationship is shown in Figure 3.12, we can observe a maximum lateral displacement in $\alpha' = 49.19^o$, this is proved in the following,

$$\frac{dX_o}{d\alpha'} = \frac{dX_o}{ds}\frac{ds}{d\alpha'}$$

$$\frac{ds}{d\alpha'} = \frac{1}{n}\cos\alpha'$$

$$\frac{dX_o}{ds}=\begin{bmatrix}\left(1-s^2\right)^{1/2}\left\{2st\left(\frac{1}{2}\right)\left(1-(ns)^2\right)^{-1/2}\left(-2(ns)\right)n+\left(1-(ns)^2\right)^{1/2}2t\right\}\\ -2st\left(1-(ns)^2\right)^{1/2}\left(\frac{1}{2}\right)\left(1-s^2\right)^{-1/2}(-2)s\end{bmatrix}\frac{\frac{1}{n}\cos\alpha'}{1-s^2}$$

to find a maximum, we do $\frac{dX_o}{ds}=0$

$$\left[\frac{-2n^2ts^2\sqrt{1-s^2}}{\sqrt{1-(ns)^2}}+2t\sqrt{1-(ns)^2}\sqrt{1-s^2}+\frac{2ts^2\sqrt{1-(ns)^2}}{\sqrt{1-s^2}}\right]\frac{1}{n}\cos\alpha'=0$$

$$\frac{-2n^2ts^2\left(1-s^2\right)+2t\left(1-(ns)^2\right)\left(1-s^2\right)+2ts^2\left(1-(ns)^2\right)}{\sqrt{1-(ns)^2}\sqrt{1-s^2}}=0$$

$$2t\left[-n^2s^2\left(1-s^2\right)+\left(1-(ns)^2\right)\left(1-s^2\right)+s^2\left(1-(ns)^2\right)\right]=0$$

$$-n^2s^2+n^2s^4+1-s^2-n^2s^2+n^2s^4+s^2-n^2s^4=0$$

$$n^2s^4-2n^2s^2+1=0$$

$$(1.5)^2s^4-2(1.5)^2s^2+1=0$$

$$2.25s^4-4.5s^2+1=0$$

$$\boxed{\alpha'_{max}=49.1945^o} \qquad (3.9)$$

3.5.3 Important parameters to be considered during the design of Murty's interferometer.

3.5.3.1 Material.

The plate plane parallel is made of glass with an index of refraction $n=1.5$ for the wavelength ($\lambda=632.8nm$) .

3.5.3.2 Incidence angle.

The lateral displacement is function of incidence angle α' on the first face of the glass plate, this is noted in equation (3.8) which is related to s as the Snell law, equation (3.7). There is an incident angle with maximum global in the lateral displacement as is indicated in equation (3.9).

3.5.3.3 Width.

The lateral shear displacement is directly proportional to the width t, as can be seen in equation (3.8).

3.5.3.4 Reflecting Surfaces.

In order to obtain a better visibility of fringes as was indicated in equation (2.1), the intensity of both interfering wave-fronts must be equal [see equation (2.25)]. To produce this we apply thin films on both faces of the plane parallel glass plate.

We will calculate the reflectivity of aluminum metallic thin films to obtain equal intensity in both wave-fronts.

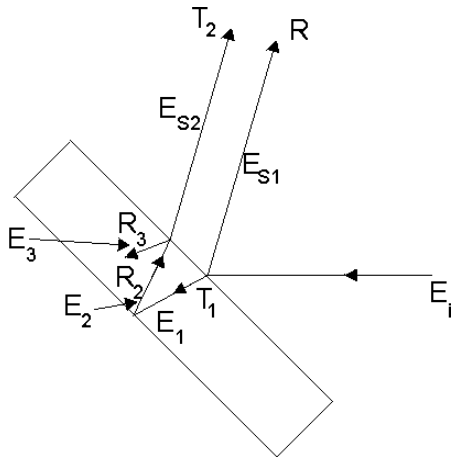


Figure 3.13. Evaluation of thin film layers for equal reflectivity on both wave-fronts.

Figure 3.13 shows a general description to calculate the reflectivity on first face, the reflectivity of the second face is 100%.

$$E_1 = (1 - A - R)E_i$$

$$\frac{E_1}{E_i} = T_1 = 1 - A - R \qquad (3.10a)$$

$$\frac{E_{s1}}{E_i} = R \qquad (3.10b)$$

$$E_{s1} = RE_i$$

$$E_2 = (1 - A)E_1 \qquad (3.10c)$$

$$\frac{E_2}{E_1} = R_2 = 1 - A$$

$$E_3 = (1 - A - T_2)E_2$$

$$\frac{E_3}{E_2} = R_3 = 1 - A - T_2$$

considering $E_3 << E_2$

$$E_3 = \frac{E_2}{10}$$
$$\frac{E_3}{E_2} = R_3 = \frac{1}{10}$$
$$T_2 = 1 - A - R_3$$
$$T_2 = \frac{9}{10} - A$$
$$\frac{E_{s2}}{E_2} = T_2$$
$$E_{s2} = T_2 E_2$$

using equation (3.10c)

$$E_{s2} = T_2\left[(1-A)E_1\right]$$

substituting equation (3.10a)

$$E_{s2} = T_2\left[(1-A)(1-A-R)E_i\right]$$

substituting equation (3.10b)

$$E_{s2} = T_2\left[(1-A)(1-A-R)\frac{E_{s1}}{R}\right]$$
$$\frac{E_{s2}}{E_{s1}} = \frac{T_2}{R}\left[(1-A)(1-A-R)\right]$$

making $E_{s1} = E_{s2}$

$$1 = \frac{T_2}{R}\left[(1-A)(1-A-R)\right]$$

where A is thin film absorption coefficient. Considering experimentally that aluminium metallic thin films have 10% of light energy absorption, $A = 0.1$

$$T_2 = \frac{8}{10}$$

The reflectivity R on the first face is

$$R = 0.3767$$

$$\boxed{R = 37.67\%} \tag{3.11}$$

3.5.4 Common errors in the ideal model.

3.5.4.1 Tilt.

This error is produced on the Murty's interferometer due to that glass plate faces are not perfectly parallel, taking in consideration this tilt in a perpendicular direction to the lateral displacement direction, because a tilt in the direction of the displacement do not produce visible fringes as is shown in following,

$$\Delta W_x(x,y) = Fx - F(x - X_o) = Fx - Fx + FX_o = FX_o$$

as we can observe the tilt produce a constant or a piston term (no fringes are produced).

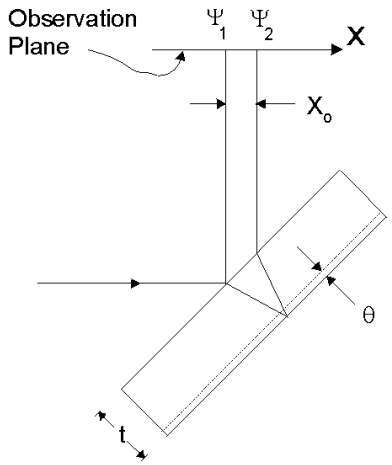


Figure 3.14. Evaluation of tilt error due to a small wedge on the plate plane parallel.

In the Figure 3.14, the y-axis is perpendicular to the plane of the paper. The x-axis is in the plane of the paper (the observation plane) and also provides the reference direction for the incident beam before reflection from the shearing plate. The wedge thickness is denoted by a dashed line and the plane of the wedge is perpendicular to the plane of the paper. With unit amplitude, the incident wave is if the form

$$\psi_1(x,y) = e^{i\phi(x,y)} \tag{3.12}$$

where $\phi(x,y)$ is the phase function describing the wave-front.

The resulting wave ψ producing an interference pattern at the observation plane is the sum of waves $\psi_1(x,y)$ and $\psi_2(x,y)$ given by

$$\psi(x,y) = \psi_1(x,y) + \psi_2(x,y)$$

$$= e^{i\phi(x,y)} + e^{i[\phi(x-X_o,y)+\beta y]} \tag{3.13}$$

where X_o is the shear produced by the plate and βy is a term due to the small wedge. Unnecessary constant phases have been neglected. For a spherical wave-front with large radius of curvature R (nearly plane), $\psi_1(x,y)$ is approximated by, remember that;

$$\phi(x,y) = kW(x,y) \tag{3.14}$$

where

$$k = \frac{2\pi}{\lambda} \tag{3.15}$$

approximating a spherical wave-front

$$W(x,y) = \frac{x^2 + y^2}{2R} \tag{3.16}$$

substituting equation (3.16), (3.15) and (3.14) on (3.12) then

$$\psi_1(x,y) = e^{i\frac{2\pi}{\lambda}\left(\frac{x^2+y^2}{2R}\right)} \tag{3.17}$$

making

$$\alpha = \frac{k}{2R}$$

substituting equation (3.17), (3.16), (3.15) and (3.14) on (3.13)

$$\begin{aligned}\psi(x,y) &= e^{i\alpha\left(x^2+y^2\right)} + e^{i\left[\alpha\left((x-X_o)^2+y^2\right)+\beta y\right]} \\ &= e^{i\alpha\left(x^2+y^2\right)} + e^{i\left[\alpha\left(x^2-2xX_o+X_o^2+y^2\right)+\beta y\right]} \\ \psi(x,y) &= e^{i\alpha\left(x^2+y^2\right)}\left[1 + e^{i\alpha\left(-2xX_o+X_o^2\right)+i\beta y}\right]\end{aligned} \tag{3.18}$$

This waveform describes straight parallel interference fringes. The slope of the interference lines with respect to the x-axis is determined by the locus of lines of constant phase. Setting the phase equal to a constant in equation (3.18) gives, remembering that the differentiation of a function in a point is equal to the slope of the tangent line in that point.

$$\begin{aligned}\frac{1}{\cos\gamma} &= \frac{\partial phase}{\partial x} = \frac{\partial}{\partial x}\left(-2i\alpha x X_o + i\alpha X_o^2 + i\beta y\right) \\ \frac{1}{\cos\gamma} &= -2i\alpha X_o\end{aligned} \tag{3.19a}$$

$$\frac{1}{\operatorname{sen}\gamma} = \frac{\partial phase}{\partial y} = \frac{\partial}{\partial y}\left(-2i\alpha x X_o + i\alpha X_o^2 + i\beta y\right)$$

$$\frac{1}{\operatorname{sen}\gamma} = i\beta \qquad (3.19b)$$

$$m = -\operatorname{tg}\gamma \qquad (3.20)$$

where m is the slope of the straight parallel interference fringes, substituting (3.19a) and (3.19b) on (3.20)

$$\operatorname{tg}\gamma = \frac{-2i\alpha X_o}{i\beta}$$

$$m = {2\alpha X_o}\big/{\beta} \qquad (3.21)$$

If the wave-front is collimated $(R \to \infty)$, $\alpha \approx 0$ and the fringes are parallel to the x-axis.

In some cases we introduce a wedge in the glass plate like in the use of a shearing plate to focus a collimating system. This characteristic is some times used to increase the initial resolution, producing some fringes, when equation (3.3) or (3.6) do not produce fringes by it self.

The wedge angle θ can be selected so that the angle between wave-fronts ψ_1 and ψ_2 is θ' and β in equation (3.13) is given in terms of θ' by

$$\beta = k \operatorname{sen}\theta'$$

$$\beta = \frac{2\pi \operatorname{sen}\theta'}{\lambda} \cong \frac{2\pi\theta'}{\lambda} \qquad (3.22)$$

for small angles.

The fringe spacing l is related to θ' by $\Delta W_x(x, y) = n\lambda$ as can be seen from (2.28)

$$y_1 \operatorname{sen}\theta' - y_2 \operatorname{sen}\theta' = (n+1)\lambda - n\lambda$$

$$(y_1 - y_2)\operatorname{sen}\theta' = \lambda$$

making $l = y_1 - y_2$

$$l = \frac{\lambda}{\operatorname{sen}\theta'} \cong \frac{\lambda}{\theta'} \qquad (3.23)$$

Substituting equations (3.23) and (3.22) in equation (3.21) and again making a small angle approximation gives

$$\begin{aligned} \text{tg}\,\gamma &= \frac{2\alpha X_o \lambda}{2\pi\theta'} \\ &= \frac{\alpha X_o \lambda l}{\pi\lambda} \\ &= \frac{k X_o l}{2R\pi} \\ \gamma &= \frac{2\pi}{\lambda}\frac{X_o l}{2R\pi} \end{aligned}$$

$$\boxed{\gamma = \frac{X_o l}{R\lambda}} \qquad (3.24)$$

3.5.4.2 Scale.

This error is produced on the Murty's interferometer due to that wave-front is not perfectly collimated

As was indicated in equation (3.21) we have a criterion to show divergence of a laser beam. In order to reduce this error, we can do a proof of collimate beam using a plane wave-front.

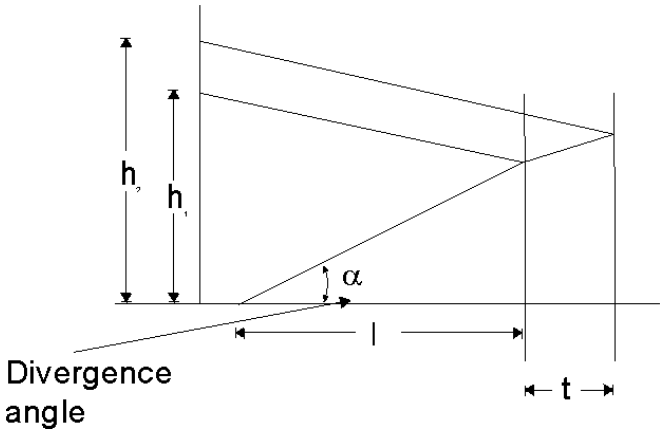


Figure 3.15. Evaluation of the error of scale due to a divergence of the beam.

The O.P.D is different in each one of wave-front, this produce that if we have not perfect collimated beam, there will be different amplification on the interfering beam, this is described as,

$$\Delta W_x(x,y) = W(x,y) - W(\xi(x - X_o), \xi y) \qquad (3.25)$$

where ξ is the amplification due to convergent or divergent beam. This can be calculated using Figure 3.15,

$$\frac{l+l}{l+t+l+t}=\frac{h_1}{h_2}$$
$$\frac{2l}{2l+2t}=\frac{h_1}{h_2}$$
$$\frac{2l}{2(l+t)}=\frac{h_1}{h_2}$$

$$\boxed{\xi=\frac{h_1}{h_2}=\frac{l}{l+t}} \qquad (3.26)$$

The divergence of the test beam is the width d of the beam (or the width of pattern limited by the shearing plate diameter) divided by the radius of curvature R. Using equation (3.24) the divergence is:

$$R=\frac{X_o l}{\gamma\lambda}$$
$$\frac{d}{R}=\frac{d}{\frac{X_o l}{\gamma\lambda}}$$

$$\boxed{\frac{d}{R}=\frac{\gamma\lambda d}{X_o l}} \qquad (3.27)$$

To illustrate the possible sensitivity, assume the fringe spacing $l=15mm$, $\lambda=0.6\mu m$, $X_o=5mm$, $d=100mm$ and the eye can resolve a fringe angle = $0.05rad$ (2.9^o). Substituting these values in equation (3.27) gives a beam divergence angle of $40\mu rad$. A line can be scribed on the surface of the shearing plate and the angle of the fringes can be judged by comparing the fringes to the shadow of the line.

3.6 Response in frequency of the Lateral Shear Interferometer (considering infinite pupils).

Considering that the aperture's extent is large enough that their common area extent over the whole plane. In these circumstances the frequency response of the lateral-shearing interferometer along the x-direction is

$$
\begin{aligned}
H_{xLSI}(\omega_x) &= \frac{\Im_x\{\Delta W_x(x,y)\}}{\Im_x\{W(x,y)\}} \\
&= \frac{\Im_x\{W(x+\delta_x,y)-W(x-\delta_x,y)\}}{\Im_x\{W(x,y)\}} \\
&= \frac{\Im_x\{W(x+\delta_x,y)\}-\Im_x\{W(x-\delta_x,y)\}}{\Im_x\{W(x,y)\}} \\
&= \frac{e^{i\delta_x\omega_x}\Im_x\{W(x,y)\}-e^{-i\delta_x\omega_x}\Im_x\{W(x,y)\}}{\Im_x\{W(x,y)\}} \\
&= \frac{\Im_x\{W(x,y)\}\left[e^{i\delta_x\omega_x}-e^{-i\delta_x\omega_x}\right]}{\Im_x\{W(x,y)\}} \\
&= 2i\,\mathrm{sen}(\delta_x\omega_x)
\end{aligned}
$$

$$
H_{xLSI}(\omega_x) = \frac{\Im_x\{\Delta W_x(x,y)\}}{\Im_x\{W(x,y)\}} = 2i\,\mathrm{sen}(\delta_x\omega_x)
$$

$$
\boxed{\left|H_{xLSI}(\omega_x)\right| = \left|\frac{\Im_x\{\Delta W_x(x,y)\}}{\Im_x\{W(x,y)\}}\right| = 2\left|\mathrm{sen}(\delta_x\omega_x)\right|}
\qquad (3.28)
$$

Note that the total displacement is equal to $2\delta_x$, this is a mathematical artifice to evaluate easier the Fourier transform, we can think $\delta_x = X_o/2$, symmetrically displaced both sides of the origin.

3.7 Response in frequency of the lateral shear interferometer (considering finite pupils).

We can't determinate an analytical expression for the frequency response Lateral Shear Interferometer, as will be show in following:

Figure 3.16. Block diagram of a Lateral Shear Interferometer considering finite pupil.

With $\Delta W_x(x,y) = [W(x-\delta_x,y)-W(x+\delta_x,y)]P(x-\delta_x,y)P(x+\delta_x,y)$

and $P(x)$ is the aperture that limits the extent of the wave-front under analysis, $W(x)$. The function $P(x)$ is equals one within the beam of light where the wave-front

being analyzed propagates and equal to zero otherwise. So on, we can write the pupil as, $P(x,y) = rect\left(\frac{x}{2A_o(y)}\right)$.

For clarity, we analyze in *x*-direction along $y = 0$,

$$P(x,y)\big|_{y=0} = P(x)$$
$$A_o(y)\big|_{y=0} = A_o$$
$$\Delta W_x(x,y)\big|_{y=0} = \Delta W(x)$$
$$W(x,y)\big|_{y=0} = W(x)$$
$$P(x,y)\big|_{y=0} = P(x)$$

then

Figure 3.17. Block diagram of a Lateral Shear Interferometer considering finite pupil.

With $\Delta W(x) = [W(x-\delta_x) - W(x+\delta_x)]P(x-\delta_x)P(x+\delta_x)$

Equivalently

$$\Delta W(x) = [W(x-\delta_x) - W(x+\delta_x)]rect\left(\frac{x-\delta_x}{2A_o}\right)rect\left(\frac{x+\delta_x}{2A_o}\right).$$

Calculating the Fourier Transform ob both sides of the last equation,

$$\Im\{\Delta W(x)\} = \Im\left\{[W(x-\delta_x) - W(x+\delta_x)]rect\left(\frac{x-\delta_x}{2A_o}\right)rect\left(\frac{x+\delta_x}{2A_o}\right)\right\}$$
$$\Im\{\Delta W(x)\} = \Im\left\{[W(x-\delta_x) - W(x+\delta_x)]rect\left(\frac{x}{2A_o - 2\delta_x}\right)\right\}$$
$$\Im\{\Delta W(x)\} = \Im\left\{W(x-\delta_x)rect\left(\frac{x}{2A_o - 2\delta_o}\right)\right\} - \Im\left\{W(x+\delta_x)rect\left(\frac{x}{2A_o - 2\delta_x}\right)\right\}$$
$$= \Im\{W(x-\delta_x)\} * \Im\left\{rect\left(\frac{x}{2A_o - 2\delta_x}\right)\right\} - \Im\{W(x+\delta_x)\} * \Im\left\{rect\left(\frac{x}{2A_o - 2\delta_x}\right)\right\}$$

$$= \left[e^{-i\delta_x\omega}\,\Im\{W(x)\}\right] * (2A_o - 2\delta_x)\,sinc\left(\frac{2A_o - 2\delta_x}{2}\,\omega\right)$$
$$- \left[e^{i\delta_x\omega}\,\Im\{W(x)\}\right] * (2A_o - 2\delta_x)\,sinc\left(\frac{2A_o - 2\delta_x}{2}\,\omega\right)$$

$$\Im\{\Delta W(x)\} = (2A_o - 2\delta_x)\left\{\begin{array}{l}\left[e^{-i\delta_x\omega}\,\Im\{W(x)\}\right] * sinc\left(\frac{2A_o - 2\delta_x}{2}\,\omega\right) \\ -\left[e^{i\delta_x\omega}\,\Im\{W(x)\}\right] * sinc\left(\frac{2A_o - 2\delta_x}{2}\,\omega\right)\end{array}\right\}$$

This last equation can be calculated numerically.

3.8 Lateral Shear Interferometer like a Linear System.

We present the Lateral Shear Interferometer, for clarity in one dimension (considering infinite pupil), with a block diagram

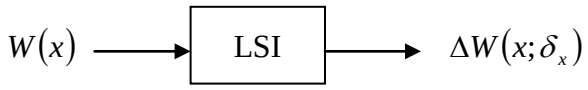


Figure 3.18. Block diagram of a Lateral Shear Interferometer considering infinite pupil.

If we consider infinite pupil function, the system is described by:

$$\boxed{\Delta W(x;\delta_x) = W(x+\delta_x) - W(x-\delta_x)} \qquad (3.30)$$

We apply some proofs to the lateral shear interferometer:

If an input $W_1(x)$ gives,

$$\Delta W_1(x;\delta_x) = W_1(x+\delta_x) - W_1(x-\delta_x) \qquad \text{equivalently } W_1(x) \to \Delta W_1(x;\delta_x)$$

and

$$\Delta W_2(x;\delta_x) = W_2(x+\delta_x) - W_2(x-\delta_x) \qquad \text{equivalently } W_2(x) \to \Delta W_2(x;\delta_x)$$

if the system is linear, it must to satisfy:

$$aW_1(x) + bW_2(x) \to a\Delta W_1(x;\delta_x) + b\Delta W_2(x;\delta_x)$$

therefore, in order to show this linear proof, we let the input to be $aW_1(x)+bW_2(x)$, to the system described by equation (3.30),

$$\begin{aligned}\Delta W_3(x;\delta_x) &= [aW_1(x+\delta_x)+bW_2(x+\delta_x)]-[aW_1(x-\delta_x)+bW_2(x-\delta_x)]\\ &= aW_1(x+\delta_x)+bW_2(x+\delta_x)-aW_1(x-\delta_x)-bW_2(x-\delta_x)\\ &= aW_1(x+\delta_x)-aW_1(x-\delta_x)+bW_2(x+\delta_x)-bW_2(x-\delta_x)\\ &= a[W_1(x+\delta_x)-W_1(x-\delta_x)]+b[W_2(x+\delta_x)-W_2(x-\delta_x)]\end{aligned}$$

$$\boxed{\Delta W_3(x;\delta_x) = a\Delta W_1(x;\delta_x)+b\Delta W_2(x;\delta_x)} \qquad (3.31)$$

so this system is linear. If the pupil is finite the LSI is linear. It important observe that can be exist lost of information due to a large lateral shearing δ_x [see chapter 5].

The proof for shift invariant displacement is,

If an input $W_1(x)$ gives,

$$\Delta W_1(x;\delta_x) = W_1(x+\delta_x)-W_1(x-\delta_x) \qquad \text{equivalently } W_1(x)\rightarrow \Delta W_1(x;\delta_x)$$

and

$$\Delta W_2(x;\delta_x) = W_2(x+\delta_x)-W_2(x-\delta_x) \qquad \text{equivalently } W_2(x)\rightarrow \Delta W_2(x;\delta_x)$$

making,

$$W_2(x) = W_1(x-X_1)$$

where X_1 is a constant.

$$\Delta W_2(x;\delta_x) = W_1(x+\delta_x-X_1)-W_1(x-\delta_x-X_1)$$

proposing a change of variable, $x = x'-x_o$ in $\Delta W_1(x;\delta_x)$

$$\begin{aligned}\Delta W_1(x'-X_1;\delta_x) &= W_1(x'-X_1+\delta_x)-W_1(x'-X_1-\delta_x)\\ \Delta W_1(x'-X_1;\delta_x) &= W_1(x'+\delta_x-X_1)-W_1(x'-\delta_x-X_1)\end{aligned}$$

as we can observe

$$\boxed{\Delta W_2(x;\delta_x) = \Delta W_1(x'-X_1;\delta_x)} \qquad (3.32)$$

therefore it is invariant in the space. If the pupil is finite, then the LSI is variant spatially. For a small input the system is stable.

For obtain their response to the impulse $h(x)$ we introduce a delta of Dirac $\delta(x)$ in the input of the interferometer (considering infinite pupil);

$$\boxed{h(x)=\delta(x+\delta_x)-\delta(x-\delta_x)} \tag{3.33}$$

The transfer function is obtained calculating the Fourier Transform of the impulse response.

$$\begin{aligned}\Im\{h(x)\}=H(\omega)&=\int_{-\infty}^{\infty}h(x)e^{-i\omega x}dx\\&=\int_{-\infty}^{\infty}\delta(x+\delta_x)e^{-i\omega x}dx-\int_{-\infty}^{\infty}\delta(x-\delta_x)e^{-i\omega x}dx\\&=e^{+i\omega\delta_x}-e^{-i\omega\delta_x}\end{aligned}$$

$$\boxed{H(\omega)=2i\,\mathrm{sen}(\omega\delta_x)} \tag{3.34}$$

the conditions for zero response in frequency are,

$$\omega\delta_x=n\pi \qquad\qquad n=0,\pm1,\pm2,\ldots$$

$$\boxed{\omega=\frac{n\pi}{\delta_x} \qquad\qquad n=0,\pm1,\pm2,\ldots} \tag{3.35}$$

The magnitude of the transfer function is given by,

$$\boxed{|H(\omega)|=2|\mathrm{sen}(\omega\delta_x)|} \tag{3.36}$$

Other equivalent way of find the transfer function (considering infinite pupils), using equation (3.30)

$$\Delta W(x;\delta_x)=W(x+\delta_x)-W(x-\delta_x)$$

evaluating the Fourier transform

$$\begin{aligned}\Im\{\Delta W(x;\delta_x)\}&=\Im\{W(x+\delta_x)-W(x-\delta_x)\}\\&=\Im\{W(x+\delta_x)\}-\Im\{W(x-\delta_x)\}\end{aligned}$$

$$= W(\omega)e^{+i\delta_x\omega} - W(\omega)e^{-i\delta_x\omega}$$

$$\boxed{H(\omega) = \frac{\Delta W(\omega)}{W(\omega)} = 2i\,\mathrm{sen}(\delta_x\omega)} \tag{3.34}$$

$$\boxed{|H(\omega)| = 2|\mathrm{sen}(\delta_x\omega)|} \tag{3.36}$$

The transfer function $\Delta W(\omega)$ is equal to zero for the following frequencies:

$$\omega\delta_x = n\pi \qquad\qquad n = 0,\pm1,\pm2,\ldots$$

$$\boxed{\omega = \frac{n\pi}{\delta_x} \qquad\qquad n = 0,\pm1,\pm2,\ldots} \tag{3.35}$$

If we consider that the TOTAL lateral shear displacement is δ_x, note that in the before case it represent the half of the total lateral shearing, is realized on one of the wave-front

$$\Delta W(x;\delta_x) = W(x) - W(x-\delta_x)$$

$$\begin{aligned}
\Im\{\Delta W(x;\delta_x)\} &= \Im\{W(x) - W(x-\delta_x)\} \\
&= \Im\{W(x)\} - \Im\{W(x-\delta_x)\} \\
&= W(\omega) - W(\omega)e^{-i\delta_x\omega} \\
&= W(\omega)\left[1 - e^{-i\delta_x\omega}\right] \\
&= W(\omega)\left[e^{-i\frac{\delta_x}{2}\omega}e^{i\frac{\delta_x}{2}\omega} - e^{-i\delta_x\omega}\right] \\
&= W(\omega)e^{-i\frac{\delta_x}{2}\omega}\left[e^{i\frac{\delta_x}{2}\omega} - e^{-i\frac{\delta_x}{2}\omega}\right]
\end{aligned}$$

$$= W(\omega)e^{-i\frac{\delta_x}{2}\omega}\left[\cos\left(\frac{\delta_x}{2}\omega\right) + i\,\mathrm{sen}\left(\frac{\delta_x}{2}\omega\right) - \cos\left(\frac{\delta_x}{2}\omega\right) + i\,\mathrm{sen}\left(\frac{\delta_x}{2}\omega\right)\right]$$

$$\boxed{\Delta W(\omega) = W(\omega)e^{-i\frac{\delta_x}{2}\omega}2i\,\mathrm{sen}\left(\frac{\delta_x}{2}\omega\right)} \tag{3.37}$$

The equation (3.37) have a change of phase $e^{-i\frac{\delta_x}{2}\omega}$ with respect to equation (3.34). This is due to that the lateral displacement is not symmetrical with respect to the origin.

The magnitude of the transfer function is given by,

$$\left|H(\omega)\right| = \left|\frac{\Delta W(\omega)}{W(\omega)}\right| = 2\left|\text{sen}\left(\frac{\delta_x}{2}\omega\right)\right| \qquad (3.36)$$

the conditions for zero response in frequency are,

$$\omega\frac{\delta_x}{2} = n\pi \qquad n = 0,\pm1,\pm2,\ldots$$

$$\boxed{\omega = \frac{2n\pi}{\delta_x} \qquad n = 0,\pm1,\pm2,\ldots} \qquad (3.38)$$

Implementing an example, we solve in the frequency domain,

If the input is:

$$W(x) = \cos(\omega_o x)$$

the Fourier transform is:

$$W(\omega) = \pi\left[\delta(\omega-\omega_o) + \delta(\omega+\omega_o)\right]$$

using the transfer function of the lateral shear interferometer, we can calculate output from the input in frequency domain:

$$\Delta W(\omega) = H(\omega)W(\omega)$$

from (3.34) $$H(\omega) = 2i\,\text{sen}(\delta_x\omega)$$

$$\Delta W(\omega) = 2i\,\text{sen}(\delta_x\omega)\pi\left[\delta(\omega-\omega_o) + \delta(\omega+\omega_o)\right]$$

$$\Delta W(\omega) = 2\pi i\,\text{sen}(+\delta_x\omega_o)\delta(\omega-\omega_o) + 2\pi i\,\text{sen}(-\delta_x\omega_o)\delta(\omega+\omega_o)$$

getting the inverse Fourier transform

$$\Im^{-1}\{\Delta W(\omega)\} = \Im^{-1}\{2\pi i \operatorname{sen}(+\delta_x\omega_o)\delta(\omega-\omega_o)\} + \Im^{-1}\{2\pi i \operatorname{sen}(-\delta_x\omega_o)\delta(\omega+\omega_o)\}$$

$$= \frac{2\pi i}{2\pi}\int_{-\infty}^{\infty}\operatorname{sen}(\delta_x\omega_o)\delta(\omega-\omega_o)e^{i\omega x}d\omega + \frac{2\pi i}{2\pi}\int_{-\infty}^{\infty}\operatorname{sen}(-\delta_x\omega_o)\delta(\omega+\omega_o)e^{i\omega x}d\omega$$

$$= i\operatorname{sen}(\delta_x\omega_o)e^{i\omega_o x} + i\operatorname{sen}(-\delta_x\omega_o)e^{-i\omega_o x}$$

$$= i\operatorname{sen}(\delta_x\omega_o)e^{i\omega_o x} - i\operatorname{sen}(\delta_x\omega_o)e^{-i\omega_o x}$$

$$= i\left[\frac{e^{i\delta_x\omega_o} - e^{-i\delta_x\omega_o}}{2i}\right]e^{i\omega_o x} - i\left[\frac{e^{i\delta_x\omega_o} - e^{-i\delta_x\omega_o}}{2i}\right]e^{-i\omega_o x}$$

$$= \frac{e^{i(x+\delta_x)\omega_o} - e^{i(x-\delta_x)\omega_o}}{2} - \frac{e^{-i(x-\delta_x)\omega_o} - e^{-i(x+\delta_x)\omega_o}}{2}$$

$$= \frac{e^{i(x+\delta_x)\omega_o} + e^{-i(x+\delta_x)\omega_o}}{2} - \frac{e^{i(x-\delta_x)\omega_o} + e^{-i(x-\delta_x)\omega_o}}{2}$$

$$= \cos(\omega_o(x+\delta_x)) - \cos(\omega_o(x-\delta_x))$$

$$\Delta W(x;\delta_x) = W(x+\delta_x) - W(x-\delta_x)$$

A critical case is when the output of the lateral shear interferometer is zero for whatever x, this is attained when:

$$\cos(\omega_o(x+\delta_x)) = \cos(\omega_o(x-\delta_x))$$

this is satisfied if

$$\omega_o\delta_x = n2\pi \qquad n = 0,\pm1,\pm2,\ldots$$

$$\delta_x = \frac{n2\pi}{\omega_o} = \frac{n2\pi}{\frac{2\pi}{T_o}} \qquad n = 0,\pm1,\pm2,\ldots$$

$$\delta_x = nT_o \qquad n = 0,\pm1,\pm2,\ldots$$

where T_o is the period of the cosine function at the input of the interferometer.

From a point of view of the spectral frequencies. The components in frequency of the cosine function there are located at the frequencies that the interferometer has zero frequency response.

3.9 Sensitivity of the Lateral Shear Interferometer.

The output sensitivity of the Lateral Shear Interferometer respect to lateral shearing (δ) can be calculated as:

$$S_{\delta}^{\Delta W} = \frac{\dfrac{\partial \Delta W(x)}{\Delta W(x)}}{\dfrac{\partial \delta}{\delta}} = \frac{\partial \Delta W(x)}{\partial \delta}\frac{\delta}{\Delta W(x)}$$

Considering infinite pupils $\Delta W(x)$ can be expressed as,

$$\Delta W(x) = W(x+\delta) - W(x-\delta)$$

we use δ instead of δ_x, for generalize the idea in *y*-direction or whatever generalized direction. Is calculated the sensitivity on the above equation,

$$\frac{\partial \Delta W(x)}{\partial \delta} = \frac{\partial}{2\delta}\left[W(x+\delta) - W(x-\delta)\right]$$

$$\frac{\partial \Delta W(x)}{2\delta} = \frac{\partial}{2\delta}W(x+\delta) - \frac{\partial}{2\delta}W(x-\delta)$$

$$S_{\delta}^{\Delta W} = \frac{\delta}{\Delta W(x)}\left[\frac{\partial}{\partial \delta}W(x+\delta) - \frac{\partial}{\partial \delta}W(x-\delta)\right]$$

$$\Im\left\{S_{\delta}^{\Delta W}\right\} = \Im\left\{\frac{\partial}{\partial \delta}\left[W(x+\delta) - W(x-\delta)\right]\right\} * \Im\left\{\frac{\delta}{\Delta W(x)}\right\}$$

$$= \frac{\partial}{\partial \delta}\left[\tilde{W}(\omega)e^{j\omega\delta} - \tilde{W}(\omega)e^{-j\omega\delta}\right] * \Im\left\{\frac{\delta}{\Delta W(x)}\right\}$$

$$= \left(2j\tilde{W}(\omega)\frac{\partial}{\partial \delta}\operatorname{sen}(\delta\omega)\right) * \Im\left\{\frac{\delta}{\Delta W(x)}\right\}$$

$$\Im^{-1}\left\{\Im\left\{S_{\delta}^{\Delta W}\right\}\right\} = \Im^{-1}\left\{\left(2j\omega\cos(\delta\omega)\tilde{W}(\omega)\right) * \Im\left\{\frac{\delta}{\Delta W(x)}\right\}\right\}$$

$$S_{\delta}^{\Delta W} = 2\frac{\partial}{\partial x}\left[\frac{\pi}{2\pi}\left[Dirac(x+\delta) + Dirac(x-\delta)\right] * W(x)\right]\frac{\delta}{\Delta W(x)}$$

This is the sensitivity of the LSI,

$$\boxed{S_{\delta}^{\Delta W} = \frac{\delta}{\Delta W(x)}\frac{\partial}{\partial x}\left[W(x+\delta) + W(x-\delta)\right]} \qquad (3.39)$$

making the change of variable $x' = x - \delta$ and $\partial x' = \partial x$

$$\boxed{S_{\delta}^{\Delta W} = \frac{\delta}{\Delta W(x'+\delta)} \frac{\partial}{\partial x'} \left[W(x'+2\delta) + W(x') \right]} \tag{3.40}$$

In the limit case, when $2\delta \to 0$, we can expand (3.40) by series' Taylor of $W(x'+2\delta)$, by

$$W(x'+2\delta) = W(x') + 2\delta \frac{\partial W(x')}{\partial x'} + \frac{(2\delta)^2}{2!} \frac{\partial^2 W(x')}{\partial x'^2} + \cdots \tag{3.41}$$

vanishing the most right-hand indicated term and subsequent terms. Re-writing equation (3.40)

$$S_{\delta}^{\Delta W} = \frac{\delta}{W(x'+2\delta) - W(x')} \frac{\partial}{\partial x'} \left[W(x'+2\delta) - W(x') \right] \tag{3.42}$$

substituting (3.41) on (3.42)

$$S_{\delta}^{\Delta W} \cong \frac{\delta}{W(x') + 2\delta \dfrac{\partial W(x')}{\partial x'} - W(x')} \frac{\partial}{\partial x'} \left[W(x') + 2\delta \frac{\partial W(x')}{\partial x'} + W(x') \right]$$

$$S_{\delta}^{\Delta W} \cong \delta \left\{ \frac{2\dfrac{\partial W(x')}{\partial x'} + 2\delta \dfrac{\partial^2 W(x')}{\partial x'^2}}{2\delta \dfrac{\partial W(x')}{\partial x'}} \right\}$$

$$S_{\delta}^{\Delta W} \cong \delta \left\{ \frac{1}{\delta} + \frac{\dfrac{\partial^2 W(x')}{\partial x'^2}}{\dfrac{\partial W(x')}{\partial x'}} \right\}$$

$$\boxed{S_{\delta}^{\Delta W} \cong 1 + \delta \frac{\partial W(x')}{\partial x'} \qquad \text{in the limit when } 2\delta \to 0} \tag{3.43}$$

3.10 Conclusions.

In this chapter we have studied the main characteristics and common errors produced by an actual Lateral Shear Interferometer, one special kind of LSI, Murty's type, was develop in order to built it. The expression for the frequency response of the LSI was obtained, giving us the idea that some frequencies are lost en the process developed by the LSI. The sensibility analysis show that is better a greater shear but as will be seen in

chapter 5 there is a compromise with a curve of lost of information that upon of the pupil function.

References.

Malacara, D., *Óptica Basica*, Fondo de Cultura Económica, México, 1989.
Malacara, D., *Optical Shop Testing*, John Wiley & Sons, USA, 1992.

4. Wave-front estimation from maps of phase lateral sheared unwrapped[1].

4.0 Objectives of this chapter.

In this chapter we pretend to show how the simple inverse filter has a worst frequency response than a regularized inverse filter. We can observe than the Fried-Hudgin method works well for a gradient field but it's inadequate for larger lateral shear. The reconstruction of the wave-front using the technique of least squares with regularization has a better frequency response than the method of least squares integration to estimate the original wave-front due mainly to the fact that we add extra information about the smooth of the estimated wave-front.

4.1 Introduction.

This chapter is based on the work of M. Servin, et. al. (1996), we pretend to explain the effects of try to recover the wave-front from simple inverse transfer function and that the use of the regularizing potentials is a must, to yield a stable solution of the least squares integration for lateral displacements greater than one pixel. We review the method of Fried and Hudgin, we observe the frequency response for this technique from least squares integration, i.e., integrating slope wave-front data. In this method the interferogram must be obtained with a little shearing (~1 CCD pixel). It Can be seen that except for a constant added to the estimated wave-front $\hat{W}(x,y)$, the problem has a well-defined and unique minimum. Therefore it is not necessary to regularize the problem for one CCD pixel lateral shearing. However, it is common to introduce large shearing (more than one CCD pixel) to increase the sensitivity of the measurement [this can be reviewed on Chapter 3 and Chapter 5].

4.2 Theory.

This development uses maps of phase lateral sheared unwrapped[2], assuming that the lateral shearing is along x-direction then the shearogram will have a fringe pattern that may be modeled as:

[1] This process is necesary due to, as was explained in Chapter 3, the Lateral Shear Interferometer (LSI) is of not absolute reference we need to integrate the field of diferences, i.e. equation (4.2) in order to get the wave-front $W(x,y)P(x,y)$.

[2] See Appendix C. Phase estimation Techniques.

$$I_x(x,y) = a(x,y) + b(x,y)\cos\left[\frac{2\pi}{\lambda}\left(2Dx + W(x-\delta_x,y) - W(x+\delta_x,y)\right)\right]P(x-\delta_x,y)P(x+\delta_x,y) \tag{4.1}$$

where $I_x(x,y)$ is the recorded interferogram's intensity.

$a(x,y)$ is the low-frequency background illumination.

$b(x,y)$ is a low-frequency amplitude modulation.

D is the amount of defocusing introduced to yield a carrier-frequency interferogram.

$2\delta_x$ is the amount of shear introduced.

$P(x,y)$ is the aperture that limits the extent of the wave-front under analysis, $W(x,y)$. The function $P(x,y)$ is equals one within the beam of light where the wave-front being analyzed propagates and equals zero otherwise.

Since we are using a known amount of defocusing, the phase detection may be achieved by direct or Fourier interferometry.

The detected and unwrapped phase of the interferogram modeled by equation (4.1), after tilt removing, is

$$\Delta W_x(x,y) = \left[W(x-\delta_x,y) - W(x+\delta_x,y)\right]P(x-\delta_x,y)P(x+\delta_x,y) \tag{4.2}$$

where $\Delta W_x(x,y)$ is the detected phase of the shearogram along the x-direction bounded by the overlapping region of the sheared wave-front's aperture, $P(x-\delta_x,y)P(x+\delta_x,y)$.

In figure 4.1 is presented in a block diagram of the Lateral Shear Interferometer (LSI).

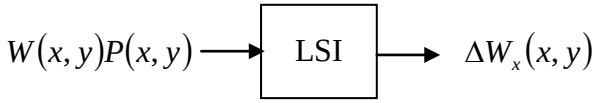


Figure 4.1 Block diagram. The Lateral Shear Interferometer in *x*-direction.

4.3 Frequency response of the LSI.

To simplify the exposition let us consider for the moment that the aperture's extent is large enough that $P(x,y)$ may be considered equal to one in the whole plane. Under this circumstances the frequency response of the lateral-shearing interferometer along the x-direction is

$$\Im_x\{\Delta W_x(x,y)\} = H_{LSI}(\omega_x)\Im_x\{W(x,y)\}$$

$$H_{LSI}(\omega_x) = \frac{\Im_x\{\Delta W_x(x,y)\}}{\Im_x\{W(x,y)\}}$$

$$= \frac{\Im_x\{W(x-\delta_x,y) - W(x+\delta_x,y)\}}{\Im_x\{W(x,y)\}}$$

$$= \frac{\Im_x\{W(x-\delta_x,y)\} - \Im_x\{W(x+\delta_x,y)\}}{\Im_x\{W(x,y)\}}$$

$$= \frac{e^{-i\delta_x\omega_x}\Im_x\{W(x,y)\} - e^{i\delta_x\omega_x}\Im_x\{W(x,y)\}}{\Im_x\{W(x,y)\}}$$

$$= \frac{\Im_x\{W(x,y)\}\left[e^{-i\delta_x\omega_x} - e^{i\delta_x\omega_x}\right]}{\Im_x\{W(x,y)\}}$$

$$= -2i\,\mathrm{sen}(\delta_x\omega_x)$$

$$\boxed{H_{LSI}(\omega_x) = \frac{\Im_x\{\Delta W_x(x,y)\}}{\Im_x\{W(x,y)\}} = -2i\,\mathrm{sen}(\delta_x\omega_x) \qquad -\infty \le \omega_x \le \infty} \tag{4.3}$$

$$\left|H_{LSI}(\omega_x)\right| = \left|\frac{\Im_x\{\Delta W_x(x,y)\}}{\Im_x\{W(x,y)\}}\right| = 2\,\mathrm{sen}(\delta_x\omega_x) \qquad -\infty \le \omega_x \le \infty$$

$$\Im_x\{W(x,y)\} = \frac{\Im_x\{\Delta W_x(x,y)\}}{H_{LSI}(\omega_x)}$$

$$\Im_x\{W(x,y)\} = \frac{\Im_x\{\Delta W_x(x,y)\}}{-2i\,\mathrm{sen}(\delta_x\omega_x)}$$

where $\Im_x\{\bullet\}$ denotes the Fourier transform operator along the x-direction. Equation (4.3) shows that the frequency response of a lateral shear interferometer has zero frequency response in the spatial frequencies,

$$\boxed{\delta_x\omega_x = k\pi \qquad k = 0,\pm1,\pm2,\pm3,\ldots \quad \text{and} \quad -\infty \le \omega_x \le \infty} \tag{4.4}$$

At these spatial frequencies, the information about the wave-front under test is lost; this is the reason that wave-front detection from a lateral-sheared interferogram is an ill-posed problem. Hadamard (Hadamard,1902) defined a mathematical problem to be well-posed if a unique solution exists that depends continuously on the data, and in this case the uniqueness requirement is violated.

If we consider that the detected phase of the shearogram $[\Delta W_x(x,y)]$ is sampled, with space between samples of Δx along the x-direction and Δy along the y-direction, i.e., with frequency sampling $f_{sx}=\frac{1}{\Delta x}$, $f_{sy}=\frac{1}{\Delta y}$ respectively, then $x=m\Delta x$ with $m=0,1,2,3,...$ and $y=n\Delta y$ with $n=0,1,2,3,...$ So far, we will work on the indices *m* and *n*, i.e. the discrete variables[3]. This development will be constricted to a soft restriction, it is that $\delta_x=k_x\Delta x$, $\delta_y=k_y\Delta y$, where $k_x=1,2,3,...$, $k_y=1,2,3,...$

Using these discrete variables on the equation (4.2) and (4.3) that can be re-write as,

$$\Delta W_x(m,n)=\left[W(m-k_x,n)-W(m+k_x,n)\right]P(m-k_x,n)P(m+k_x,n) \tag{4.2a}$$

$$H_{xLSI}(\Omega_x)=\frac{\Im_x\{\Delta W_x(m,n)\}}{\Im_x\{W(m,n)\}}=-2i\,\mathrm{sen}(k_x\Omega_x) \qquad -\pi\le\Omega_x\le\pi \tag{4.3a}$$

The frequency response of the lateral shear interferometer that has zero frequency response in *x*-direction are in the spatial frequencies:

$$k_x\Omega_x=k\pi \qquad k=0,\pm1,\pm2,\pm3,... \quad \text{and} \quad -\pi\le\Omega_x\le\pi \tag{4.4a}$$

Similarity in *y*-direction,

$$\Delta W_y(m,n)=\left[W(m,n-k_y)-W(m,n+k_y)\right]P(m,n-k_y)P(m,n+k_y) \tag{4.2b}$$

$$H_{yLSI}(\Omega_y)=\frac{\Im_y\{\Delta W_y(m,n)\}}{\Im_y\{W(m,n)\}}=-2i\,\mathrm{sen}(k_y\Omega_y) \qquad -\pi\le\Omega_y\le\pi \tag{4.3b}$$

[3] See Appendix A for a classification and description of the signals.

The frequency response of the lateral shear interferometer that has zero frequency response in *y*-direction are in the spatial frequencies:

$$k_y\Omega_y = k\pi \qquad k = 0,\pm1,\pm2,\pm3,... \quad \text{and} \quad -\pi \le \Omega_y \le \pi \tag{4.4b}$$

4.4 Simple inverse transfer function of the LSI.

To recover the wave-front $W(m,n)$ from $\Delta W_x(m,n)$ and $\Delta W_y(m,n)$, we can get the *simple* inverse transfer function in *x* and *y* – direction from (4.3a) and (4.3b) respectively, then

$$H_{xLSI}^{-1}(\Omega_x) = \frac{\Im_x\{W(m,n)\}}{\Im_x\{\Delta W_x(m,n)\}} = \frac{i}{2\,\mathrm{sen}(k_x\Omega_x)} \qquad -\pi \le \Omega_x \le \pi \tag{4.3c}$$

$$H_{yLSI}^{-1}(\Omega_y) = \frac{\Im_y\{W(m,n)\}}{\Im_y\{\Delta W_y(m,n)\}} = \frac{i}{2\,\mathrm{sen}(k_y\Omega_y)} \qquad -\pi \le \Omega_y \le \pi \tag{4.3d}$$

As was shown in chapter 3, the lateral shear interferometer ideally can be considered a linear system, so on, the inverse filters regularized in *x* and *y*-direction can be cascaded, then

$$H_{LSI}^{-1}(\Omega_x,\Omega_y) = H_{xLSI}^{-1}(\Omega_x)H_{yLSI}^{-1}(\Omega_y) \qquad \text{with} \quad |\Omega_x| \le \pi \quad \text{and} \quad |\Omega_y| \le \pi$$

In figure 4.2 is presented in a block diagram of the simple inverse filter (H_{LSI}^{-1}) used to recover $\Im\{W(m,n)\}$.

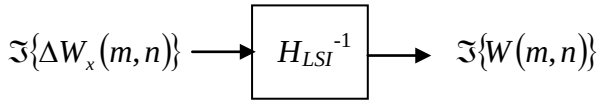


Figure 4.2 Block diagram of the simple inverse transfer function used to recover $W(m,n)$.

This simple inverse transfer function has two main limitations: first, consider infinite pupils; and second, it has poles at frequencies given by equations (4.4a) and (4.4b).

The undesirable efects of this two main drawbacks are showed in following cases, In figure 4.3, 4.4 and 4.5.

a) b)

Figure 4.3 a) Constat wave-front lateral shearing along *x*-direction with k_x=2 CCD pixels and b) Wave-front recovered using the simple inverse transfer function.

a) b)

Figure 4.4 a) Constat wave-front lateral shearing along *x*-direction with k_x=9 CCD pixels and b) Wave-front recovered using the simple inverse transfer function.

a) b)

Figure 4.5 a) Constat wave-front lateral shearing along *x*-direction with k_x=16 and b) Wave-front recovered using the simple inverse transfer function.

In figures 4.3, 4.4 and 4.5, we can think that the problem is that we don't know the regions out of the overlapping region, this in not the case for the figures 4.3 and 4.4, as can be verified from the curves of lost of information in chapter 5. For a circular pupile a little shearing so on the lost of information is minimum. The effect observed of spread out of the pupile on the recovery of the wave-front is due to the poles in the simple inverse transfer function. In figure 4.5 in addition to this effect, appears the lost of information due to a large shearing [see the curves of lost of information in chapter 5].

4.5 Reconstruction by Least Squares without Regularization.

Alternatively, to recover the wave-front $W(m,n)$ from $\Delta W_x(m,n)$ and $\Delta W_y(m,n)$, one may choose $\hat{W}(m,n)$ so that it minimizes the squared error U ,

$$\boxed{U = \sum_{(m,n)\in L} U_x^2(m,n) + U_y^2(m,n) \qquad (m,n)\in L} \tag{4.5}$$

where L is a finite two-dimensional regular lattice of size *M* along the x-direction and size *N* along the y-direction, then

$$\boxed{\begin{aligned} U_x(m,n) &= \left[\hat{W}(m+k_x,n) - \hat{W}(m-k_x,n) + \Delta W_x(m,n)\right] P(m+k_x,n) P(m-k_x,n) \\ U_y(m,n) &= \left[\hat{W}(m,n+k_y) - \hat{W}(m,n-k_y) + \Delta W_y(m,n)\right] P(m,n+k_y) P(m,n-k_y) \end{aligned}} \tag{4.6}$$

The lateral shears $\Delta W_x(m,n)$ and $\Delta W_y(m,n)$ are given, as usual, as

$$\boxed{\begin{aligned} \Delta W_x(m,n) &= \left[W(m-k_x,n) - W(m+k_x,n)\right] P(m-k_x,n) P(m+k_x,n) \\ \Delta W_y(m,n) &= \left[W(m,n-k_y) - W(m,n+k_y)\right] P(m,n-k_y) P(m,n+k_y) \end{aligned}} \tag{4.7}$$

making the substitution equation (4.6) in equation (4.5),

$$U = \sum_{(m,n)\in L} \left\{ \begin{aligned} &\left[\hat{W}(m+k_x,n) - \hat{W}(m-k_x,n) + \Delta W_x(m,n)\right]^2 \\ &+ \left[\hat{W}(m,n+k_y) - \hat{W}(m,n-k_{yx}) + \Delta W_y(m,n)\right]^2 \end{aligned} \right\} \qquad (m,n)\in L$$

expanding this last equation in the summation,

$$\begin{aligned} U = \cdots &+ \left[\hat{W}(m,n) - \hat{W}(m-2k_x,n) + \Delta W_x(m-k_x,n)\right]^2 + \cdots \\ &+ \left[\hat{W}(m+k_x,n) - \hat{W}(m-k_x,n) + \Delta W_x(m,n)\right]^2 + \cdots \\ &+ \left[\hat{W}(m+2k_x,n) - \hat{W}(m,n) + \Delta W_x(m+k_x,n)\right]^2 + \cdots \\ &+ \left[\hat{W}(m,n) - \hat{W}(m,n-2k_y) + \Delta W_y(m,n-k_y)\right]^2 + \cdots \\ &+ \left[\hat{W}(m,n+k_y) - \hat{W}(m,n-k_y) + \Delta W_y(m,n)\right]^2 + \cdots \\ &+ \left[\hat{W}(m,n+2k_y) - \hat{W}(m,n) + \Delta W_y(m,n+k_y)\right]^2 + \cdots \end{aligned}$$

$$\frac{\partial U}{\partial \hat{W}(m,n)} = 2\left[\hat{W}(m,n) - \hat{W}(m-2k_x,n) + \Delta W_x(m-k_x,n)\right]$$
$$+ 2\left[\hat{W}(m+2k_x,n) - \hat{W}(m,n) + \Delta W_x(m+k_x,n)\right](-1)$$
$$+ 2\left[\hat{W}(m,n) - \hat{W}(m,n-2k_y) + \Delta W_y(m,n-k_y)\right]$$
$$+ 2\left[\hat{W}(m,n+2k_y) - \hat{W}(m,n) + \Delta W_y(m,n+k_y)\right](-1) = 0$$

$$2\left\{U_x(m-k_x,n) - U_x(m+k_x,n) + U_y(m,n-k_y) - U_y(m,n+k_y)\right\} = 0$$
$$U_x(m-k_x,n) - U_x(m+k_x,n) + U_y(m,n-k_y) - U_y(m,n+k_y) = 0 \qquad (4.7a)$$

Taking the Fourier transform in x-direction ($\Im_x\{\bullet\}$) of equation (4.7a), considering that the aperture's extent is large enough that $P(x,y)$ may be considered equal to one in the whole plane.

$$\Im_x\left\{U_x(m-k_x,n) - U_x(m+k_x,n) + U_y(m,n-k_y) - U_y(m,n+k_y)\right\} = 0$$
$$\Im_x\{U_x(m-k_x,n)\} - \Im_x\{U_x(m+k_x,n)\} + \Im_x\{U_y(m,n-k_y)\} - \Im_x\{U_y(m,n+k_y)\} = 0$$
$$\Im_x\{\hat{W}(m,n)\} - \Im_x\{\hat{W}(m-2k_x,n)\} + \Im_x\{\Delta W_x(m-k_x,n)\}$$
$$- \Im_x\{\hat{W}(m+2k_x,n)\} + \Im_x\{\hat{W}(m,n)\} - \Im_x\{\Delta W_x(m+k_x,n)\} = 0$$
$$\Im_x\{\hat{W}(m,n)\}\left[2 - e^{-2k_x\Omega_x} - e^{2k_x\Omega_x}\right] + \Im_x\{\Delta W_x(m,n)\}\left[e^{-k_x\Omega_x} - e^{k_x\Omega_x}\right] = 0$$
$$\Im_x\{\hat{W}(m,n)\}\left[2 - 2\cos(2k_x\Omega_x)\right] = -\Im_x\{\Delta W_x(m,n)\}\left(-2i\,\mathrm{sen}(k_x\Omega_x)\right)$$

In this circumstances the frequency response of the inverse filter *not regularized* used to recover $\hat{W}(m,n)$ along the x-direction is

$$H^{-1}_{xNR}(\Omega_x) = \frac{\Im_x\{\hat{W}(m,n)\}}{\Im_x\{\Delta W_x(m,n)\}} = \frac{2i\,\mathrm{sen}(k_x\Omega_x)}{2 - 2\cos(2k_x\Omega_x)} \qquad -\pi \le \Omega_x \le \pi$$

$$H^{-1}_{xNR}(\Omega_x) = \frac{\Im_x\{\hat{W}(m,n)\}}{\Im_x\{\Delta W_x(m,n)\}} = \frac{i\,\mathrm{sen}(k_x\Omega_x)}{1 - \cos(2k_x\Omega_x)} \qquad -\pi \le \Omega_x \le \pi\,.$$

Similarity in *y*-direction,

$$H^{-1}_{yNR}(\Omega_y) = \frac{\Im_y\{\hat{W}(m,n)\}}{\Im_y\{\Delta W_y(m,n)\}} = \frac{i\,\mathrm{sen}(k_y\Omega_y)}{1 - \cos(2k_y\Omega_y)} \qquad -\pi \le \Omega_y \le \pi$$

As was shown in chapter 3, the lateral shear interferometer ideally can be considered a linear system, so on, the inverse filters not regularized in *x* and *y*-direction can be cascaded, then

$$H_{NR}^{-1}(\Omega_x,\Omega_y)=H_{xNR}^{-1}(\Omega_x)H_{yNR}^{-1}(\Omega_y) \qquad \text{with} \quad |\Omega_x|\le\pi \qquad \text{and} \quad |\Omega_y|\le\pi$$

In figure 4.6 is presented in a block diagram of the inverse filter not regularized (H_{NR}^{-1}) used to recover $\Im\{\hat{W}(m,n)\}$.

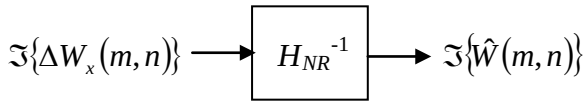


Figure 4.6 Block diagram of the inverse filter not regularized used to recover $\Im\{\hat{W}(m,n)\}$.

The magnitude of the frequency response of the inverse filter not regularized that is get from the minimization of equation (4.5) is:

$$|H(\Omega_x)|=\left|\frac{\Im_x\{\hat{W}(m,n)\}}{\Im_x\{\Delta W_x(m,n)\}}\right|=\frac{\text{sen}(k_x\Omega_x)}{1-\cos(2k_x\Omega_x)} \qquad -\pi\le\Omega_x\le\pi \tag{4.7b}$$

The equation (4.7b) is indeterminate, when

$$\Omega_x=\frac{k\pi}{k_x} \qquad k=0,\pm1,\pm2,\pm3,... \qquad -\pi\le\Omega_x\le\pi$$

and, similarity for *y*-direction

$$\Omega_y=\frac{k\pi}{k_y} \qquad k=0,\pm1,\pm2,\pm3,... \qquad -\pi\le\Omega_x\le\pi$$

However, for the reason given above (the indeterminacies in the spatial frequency plane), this minimization problem [relation (4.7a)] is not well posed.

4.6 Reconstruction by Least Squares with Regularization.

To regularize it inverse filter not regularized, it is necessary to add a term to U that corresponds to an a priori smoothness assumption. In classical regularization theory (Thikonov,1963), this term consist of a linear combination of the squared magnitude of derivatives of $\hat{W}(m,n)$ inside the domain of interest. In particular, one may use a discrete approximation to the Laplacian to obtain the second order potentials

$$\frac{\partial\hat{W}(x,y)}{\partial x}\approx\hat{W}(m,n)-\hat{W}(m-1,n)$$
$$\frac{\partial^2\hat{W}(x,y)}{\partial x^2}\approx\hat{W}(m,n)-\hat{W}(m-1,n)-\hat{W}(m-1,n)+\hat{W}(m-2,n)$$

$$\approx \hat{W}(m,n) - 2\hat{W}(m-1,n) + \hat{W}(m-2,n)$$
$$\approx \hat{W}(m-2,n) - 2\hat{W}(m-1,n) + \hat{W}(m,n)$$

then

$$\boxed{R_x(m,n) = \left[\hat{W}(m-1,n) - 2\hat{W}(m,n) + \hat{W}(m+1,n)\right] P(m-1,n)P(m,n)P(m+1,n)} \tag{4.8a}$$

similarity for *y*-direction

$$\frac{\partial \hat{W}(x,y)}{\partial y} = \hat{W}(m,n) - \hat{W}(m,n-1)$$
$$\frac{\partial^2 \hat{W}(x,y)}{\partial y^2} \approx \hat{W}(m,n) - \hat{W}(m,n-1) - \hat{W}(m,n-1) + \hat{W}(m,n-2)$$
$$\approx \hat{W}(m,n) - 2\hat{W}(m,n-1) + \hat{W}(m,n-2)$$
$$\approx \hat{W}(m,n-2) - 2\hat{W}(m,n-1) + \hat{W}(m,n)$$

then

$$\boxed{R_y(m,n) = \left[\hat{W}(m,n-1) - 2\hat{W}(m,n) + \hat{W}(m,n+1)\right] P(m,n-1)P(m,n)P(m,n+1)} \tag{4.8b}$$

Note that in this case the domain of the regularizing potentials is equal to the full aperture $P(m,n)$. The final cost function then becomes

$$\boxed{U = \sum_{(m,n)\in L} U_x^2(m,n) + U_y^2(m,n) + \alpha\left[R_x^2(m,n) + R_y^2(m,n)\right] \qquad (m,n) \in L} \tag{4.9}$$

The solution obtained with these potentials behaves like a thin metallic plate attached to the observations with linear springs. The regularizing potentials discourage large changes in the wave-front among neighboring pixels. As a consecuence, the solutions for $W(m,n)$ will be relatively smooth. The parameter α controls the amount of smoothness of the estimated wave-front $\hat{W}(m,n)$ [see figure 5.11 in chapter 5].

It should be remarked that the use of regularizing potentials is a must, even for noise-free observations, to yield a stable solution of the least squares integration for lateral displacements greater than two pixels [this will be explained in detail in chapter 6].

Taking the derivative of equation (4.9) with respect to $\hat{W}(m,n)$ and equating it to zero, we end up with the set of linear equations. Expanding the equation (4.9) in the summation,

$$
\begin{aligned}
U = {} & \cdots + \left[\hat{W}(m,n) - \hat{W}(m-2k_x,n) + \Delta W_x(m-k_x,n)\right]^2 + \cdots + \\
& \cdots + \left[\hat{W}(m+k_x,n) - \hat{W}(m-k_x,n) + \Delta W_x(m,n)\right]^2 + \cdots + \\
& \cdots + \left[\hat{W}(m+2k_x,n) - \hat{W}(m,n) + \Delta W_x(m+k_x,n)\right]^2 + \cdots + \\
& \vdots \\
& \cdots + \left[\hat{W}(m,n) - \hat{W}(m,n-2k_y) + \Delta W_y(m,n-k_y)\right]^2 + \cdots + \\
& \cdots + \left[\hat{W}(m,n+k_y) - \hat{W}(m,n-k_y) + \Delta W_y(m,n)\right]^2 + \cdots + \\
& \cdots + \left[\hat{W}(m,n+2k_y) - \hat{W}(m,n) + \Delta W_y(m,n+k_y)\right]^2 + \cdots + \\
+ \alpha \Bigg\{ & \begin{array}{l}
\cdots + \left[\hat{W}(m-2,n) - 2\hat{W}(m-1,n) + \hat{W}(m,n)\right]^2 \\
+ \left[\hat{W}(m-1,n) - 2\hat{W}(m,n) + \hat{W}(m+1,n)\right]^2 \\
+ \left[\hat{W}(m,n) - 2\hat{W}(m+1,n) + \hat{W}(m+2,n)\right]^2 + \\
\vdots \\
+ \left[\hat{W}(m,n-2) - 2\hat{W}(m,n-1) + \hat{W}(m,n)\right]^2 \\
+ \left[\hat{W}(m,n-1) - 2\hat{W}(m,n) + \hat{W}(m,n+1)\right]^2 \\
+ \left[\hat{W}(m,n) - 2\hat{W}(m,n+1) + \hat{W}(m,n+2)\right]^2 + \cdots
\end{array} \Bigg\}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial U}{\partial \hat{W}(m,n)} = 0 = {} & 2\left[\hat{W}(m,n) - \hat{W}(m-2k_x,n) + \Delta W_x(m-k_x,n)\right] + \\
& 2\left[\hat{W}(m+2k_x,n) - \hat{W}(m,n) + \Delta W_x(m+k_x,n)\right](-1) + \\
& 2\left[\hat{W}(m,n) - \hat{W}(m,n-2k_y) + \Delta W_y(m,n-k_y)\right] + \\
& 2\left[\hat{W}(m,n+2k_y) - \hat{W}(m,n) + \Delta W_y(m,n+k_y)\right](-1) + \\
+ \alpha \Bigg\{ & \begin{array}{l}
2\left[\hat{W}(m-2,n) - 2\hat{W}(m-1,n) + \hat{W}(m,n)\right] + \\
2\left[\hat{W}(m-1,n) - 2\hat{W}(m,n) + \hat{W}(m+1,n)\right](-2) + \\
2\left[\hat{W}(m,n) - 2\hat{W}(m+1,n) + \hat{W}(m+2,n)\right] + \\
2\left[\hat{W}(m,n-2) - 2\hat{W}(m,n-1) + \hat{W}(m,n)\right] + \\
2\left[\hat{W}(m,n-1) - 2\hat{W}(m,n) + \hat{W}(m,n+1)\right](-2) + \\
2\left[\hat{W}(m,n) - 2\hat{W}(m,n+1) + \hat{W}(m,n+2)\right]
\end{array} \Bigg\}
\end{aligned}
$$

$$
\frac{\partial U}{\partial \hat{W}(m,n)} = 2\left\{U_x(m-k_x,n) - U_x(m+k_x,n) + U_y(m,n-k_y) - U_y(m,n+k_y)\right\} +
$$

$$
2\alpha\left\{R_x(m-1,n) - 2R_x(m,n) + R_x(m+1,n) + R_y(m,n-1) - 2R_y(m,n) + R_y(m,n+1)\right\} = 0
$$

$$\boxed{\begin{aligned}\frac{\partial U}{\partial \hat{W}(m,n)} &= U_x(m-k_x,n)-U_x(m+k_x,n)+U_y(m,n-k_y)-U_y(m,n+k_y)\\ &+\alpha\left[R_x(m-1,n)-2R_x(m,n)+R_x(m+1,n)\right]\\ &+\alpha\left[R_y(m,n-1)-2R_y(m,n)+R_y(m,n+1)\right]=0\end{aligned}} \tag{4.10}$$

This linear system may be solved numerically for the estimated wave-front $\hat{W}(m,n)$. Probably the simplest (although not the most efficient) method is gradient descent, in which the solution is found as the stable point of the following iterative system:

$$\boxed{\hat{W}(m,n)^{k+1}=\hat{W}(m,n)^{k}-\gamma\frac{\partial U}{\partial \hat{W}(m,n)}} \tag{4.11}$$

where γ is a small constant that controls the convergence rate of the algorithm [see Appendix E for alternative methods for solve a linear system of equations].

Let us now study the frequency response in *x*-direction of the system given by equation (4.10), assuming an aperture of infinite size. Taking the Fourier transform in *x*-direction on both sides of equation (4.10)

$$\begin{aligned}\Im_x\left\{\frac{\partial U}{\partial \hat{W}(m,n)}\right\} &= -\Im_x\{U_x(m+k_x,n)\}+\Im_x\{U(m-k_x,n)\}\\ &-\Im_x\{U_y(m,n+k_y)\}+\Im_x\{U_y(m,n-k_y)\}\\ &+\alpha\left[\Im_x\{R_x(m+1,n)\}-2\Im_x\{R_x(m,n)\}+\Im_x\{R_x(m-1,n)\}\right]\\ &+\alpha\left[\Im_x\{R_y(m,n+1)\}-2\Im_x\{R_y(m,n)\}+\Im_x\{R_y(m,n-1)\}\right]=0\end{aligned}$$

$$\Im_x\left\{\frac{\partial U}{\partial \hat{W}(m,n)}\right\}=\Im_x\{U_x(m,n)\}\left[-e^{ik_x\Omega_x}+e^{-ik_x\Omega_x}\right]+$$

$$\alpha\Im_x\begin{bmatrix}\hat{W}(m,n)-2\hat{W}(m+1,n)+\hat{W}(m+2,n)-2\hat{W}(m-1,n)+4\hat{W}(m,n)-\\ 2\hat{W}(m+1,n)+\hat{W}(m-2,n)-2\hat{W}(m-1,n)+\hat{W}(m,n)\end{bmatrix}=0$$

$$\Im_x\left\{\frac{\partial U}{\partial \hat{W}(m,n)}\right\}=\alpha\begin{bmatrix}6\Im_x\{\hat{W}(m,n)\}-4\Im_x\{\hat{W}(m,n)\}\left[e^{i\Omega_x}+e^{-i\Omega_x}\right]+\\ \Im_x\{\hat{W}(m,n)\}\left[e^{i2\Omega_x}+e^{-i2\Omega_x}\right]\end{bmatrix}+$$
$$\Im_x\{U_x(m,n)\}\left(-2i\,\mathrm{sen}(k_x\Omega_x)\right)=0$$

$$\Im_x\{U_x(m,n)\}\left[-2i\,\mathrm{sen}(k_x\Omega_x)\right]+\Im_x\{\hat{W}(m,n)\}\alpha\left[6-8\cos(\Omega_x)+2\cos(2\Omega_x)\right]=0$$

$$\left[\Im_x\{\hat{W}(m+k_x,n)\}-\Im_x\{\hat{W}(m-k_x,n)\}+\Im_x\{\Delta W_x(m,n)\}\right]\left[2i\,\mathrm{sen}(k_x\Omega_x)\right]=$$
$$\Im_x\{\hat{W}(m,n)\}\alpha\left[6-8\cos(\Omega_x)+2\cos(2\Omega_x)\right]$$

$$\left[\Im_x\{\hat{W}(m,n)\}\left[e^{ik_x\Omega_x}-e^{-ik_x\Omega_x}\right]+\Im_x\{\Delta W_x(m,n)\}\right]2i\,\mathrm{sen}(k_x\Omega_x)=$$
$$\Im_x\{\hat{W}(m,n)\}\alpha\left[6-8\cos(\Omega_x)+2\cos(2\Omega_x)\right]$$

$$\Im_x\{\hat{W}(m,n)\}\left[2i\,\mathrm{sen}(k_x\Omega_x)\right]\left(2i\,\mathrm{sen}(k_x\Omega_x)\right)+\Im_x\{\Delta W_x(m,n)\}\left(2i\,\mathrm{sen}(k_x\Omega_x)\right)=$$
$$\Im_x\{\hat{W}(m,n)\}\alpha\left[6-8\cos(\Omega_x)+2\cos(2\Omega_x)\right]$$

$$\Im_x\{\Delta W_x(m,n)\}2i\,\mathrm{sen}(k_x\Omega_x)=\Im_x\{\hat{W}(m,n)\}\left[4\,\mathrm{sen}^2(k_x\Omega_x)+\alpha\left[6-8\cos(\Omega_x)+2\cos(2\Omega_x)\right]\right]$$

$$H_{xR}^{-1}(\Omega_x,\alpha)=\frac{\Im_x\{\hat{W}(m,n)\}}{\Im_x\{\Delta W_x(m,n)\}}=\frac{2i\,\mathrm{sen}(k_x\Omega_x)}{4\,\mathrm{sen}^2(k_x\Omega_x)+\alpha\left[6-8\cos(\Omega_x)+2\cos(2\Omega_x)\right]}$$

$$\boxed{H_{xR}^{-1}(\Omega_x,\alpha)=\frac{2i\,\mathrm{sen}(k_x\Omega_x)}{2-2\cos(2k_x\Omega_x)+\alpha\left[6-8\cos(\Omega_x)+2\cos(2\Omega_x)\right]}} \quad (4.12a)$$

the magnitude is

$$\boxed{\left|H_{xR}^{-1}(\Omega_x,\alpha)\right|=\frac{2\,\mathrm{sen}(k_x\Omega_x)}{2-2\cos(2k_x\Omega_x)+\alpha\left[6-8\cos(\Omega_x)+2\cos(2\Omega_x)\right]}} \quad (4.13a)$$

with $-\pi\le\Omega_x\le\pi$. Similarity in *y*-direction,

$$\boxed{H_{yR}^{-1}(\Omega_y,\alpha)=\frac{2i\,\mathrm{sen}(k_y\Omega_y)}{2-2\cos(2k_y\Omega_y)+\alpha\left[6-8\cos(\Omega_y)+2\cos(2\Omega_y)\right]}} \quad (4.12b)$$

the magnitude is

$$\boxed{\left|H_{yR}^{-1}(\Omega_y,\alpha)\right|=\frac{2\,\mathrm{sen}(k_y\Omega_y)}{2-2\cos(2k_y\Omega_y)+\alpha\left[6-8\cos(\Omega_y)+2\cos(2\Omega_y)\right]}} \quad (4.13b)$$

with $-\pi\le\Omega_y\le\pi$.

The equations (4.12) and (4.13) may be regarded as the frequency response of the inverse filter used to recover the sheared information along the *x* and *y*-direction.

As was shown in chapter 3, the lateral shear interferometer ideally can be considered a linear system, so on, the inverse filters regularized in *x* and *y*-direction can be cascaded, then

$$H_R^{-1}(\Omega_x,\Omega_y)=H_{xR}^{-1}(\Omega_x)H_{yR}^{-1}(\Omega_y) \qquad \text{with} \quad |\Omega_x|\le\pi \qquad \text{and} \quad |\Omega_y|\le\pi$$

In figure 4.7 is presented in a block diagram of the inverse filter *regularized* (H_R^{-1}) used to recover $\Im\{\hat{W}(m,n)\}$.

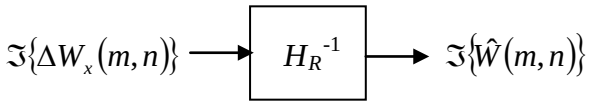


Figure 4.7 Block diagram of the inverse filter regularized used to recover $\Im\{\hat{W}(m,n)\}$.

4.7 Fried-Hudgin Method.

Making the analysis of methods proposed earlier (Fried,1977; Hudgin,1977; Noll,1978; Hunt,1979; Takajo,1988; Ghiglia,1989) for integrating slope wave-front data. Following the notation used in this chapter, the Fried (Fried,1977) and Hudgin (Hudgin,1977) method as applied to the shearing data given in equation (4.5) is reduced to finding the wave-front $\hat{W}(m,n)$ that minimizes the following quadratic cost functional:

$$U=\sum_{(m,n)\in L}U_x^2(m,n)+U_y^2(m,n) \qquad (m,n)\in L \tag{4.14}$$

where

$$U_x(m,n)=\left[\hat{W}(m+1,n)-\hat{W}(m,n)-\frac{\Delta W_x(m,n)}{2k_x}\right]P(m+1,n)P(m,n)$$

$$U_y(m,n)=\left[\hat{W}(m,n+1)-\hat{W}(m,n)-\frac{\Delta W_y(m,n)}{2k_y}\right]P(m,n+1)P(m,n) \tag{4.15}$$

Note that except for a constant added to the estimated wave-front $\hat{W}(m,n)$ (which may specified as equal to zero), the problem has a well-defined and unique minimum. Therefore it is not necessary to regularize the problem. As the equations (4.14) and (4.15) show, the Fried-Hudgin method assumes small wave-front differences. In shearing interferometry, however, it is common to introduce large shearing (more than one CCD pixel) to increase the sensitivity of the measurement.

Substituting (4.15) in (4.14) and expanding on the sum,

$$U = \cdots + \left[\hat{W}(m,n) - \hat{W}(m-1,n) - \frac{\Delta W_x(m-1,n)}{2k_x}\right]^2 +$$
$$\left[\hat{W}(m+1,n) - \hat{W}(m,n) - \frac{\Delta W_x(m,n)}{2k_x}\right]^2 +$$
$$\vdots$$
$$+\left[\hat{W}(m,n) - \hat{W}(m,n-1) - \frac{\Delta W_y(m,n-1)}{2k_y}\right]^2 +$$
$$\left[\hat{W}(m,n+1) - \hat{W}(m,n) - \frac{\Delta W_y(m,n)}{2k_y}\right]^2 + \cdots$$

$$\frac{\partial U}{\partial \hat{W}(m,n)} = 2\left[\hat{W}(m,n) - \hat{W}(m-1,n) - \frac{\Delta W_x(m-1,n)}{2k_x}\right]$$
$$+ 2\left[\hat{W}(m+1,n) - \hat{W}(m,n) - \frac{\Delta W_x(m,n)}{2k_x}\right](-1)$$
$$+ 2\left[\hat{W}(m,n) - \hat{W}(m,n-1) - \frac{\Delta W_y(m,n-1)}{2k_y}\right]$$
$$+ 2\left[\hat{W}(m,n+1) - \hat{W}(m,n) - \frac{\Delta W_y(m,n)}{2k_y}\right](-1) = 0$$

$$\frac{\partial U}{\partial \hat{W}(m,n)} = 2\left\{U_x(m-1,n) - U_x(m,n) + U_y(m,n-1) - U_y(m,n)\right\} = 0$$

$$\boxed{\frac{\partial U}{\partial \hat{W}(m,n)} = U_x(m-1,n) - U_x(m,n) + U_y(m,n-1) - U_y(m,n) = 0} \tag{4.16}$$

The minimizes of equation (4.14) may be found recursively by simple gradient descent [substituting equation (4.16) on equation (4.11)] as

$$\boxed{\hat{W}(m,n)^{k+1} = \hat{W}(m,n)^k + \gamma\left[U_x(m-1,n) - U_x(m,n) + U_y(m,n-1) - U_y(m,n)\right]} \tag{4.16a}$$

Assuming a very large aperture $P(m,n)$, we may find the frequency response of the minimizes of the least-squares estimator given by equation (4.14) along the *x*-direction, i.e., taking the Fourier transform along the *x*-direction of equation (4.16), then

$$\Im_x\left\{\frac{\partial U}{\partial \hat{W}(m,n)}\right\} = \Im_x\{U_x(m-1,n)\} - \Im_x\{U_x(m,n)\} + \Im_x\{U_y(m,n-1)\} - \Im_x\{U_y(m,n)\} = 0$$

$$= \Im_x\{\hat{W}(m,n)\} - \Im_x\{\hat{W}(m-1,n)\} - \frac{1}{2k_x}\Im_x\{\Delta W_x(m-1,n)\}$$

$$- \Im_x\{\hat{W}(m+1,n)\} + \Im_x\{\hat{W}(m,n)\} + \frac{1}{2k_x}\Im_x\{\Delta W_x(m,n)\} = 0$$

$$2\Im_x\{\hat{W}(m,n)\} - \Im_x\{\hat{W}(m,n)\}\left[e^{-i\Omega_x} + e^{i\Omega_x}\right] + \frac{1}{2k_x}\Im_x\{\Delta W_x(m,n)\}\left[-e^{-i\Omega_x} + 1\right] = 0$$

$$\Im_x\{\hat{W}(m,n)\}[2 - 2\cos(\Omega_x)]2k_x = \Im_x\{\Delta W_x(m,n)\}[1 - \cos(\Omega_x) + i\,\mathrm{sen}(\Omega_x)]$$

$$\boxed{H_{xFH}^{-1}(\Omega_x) = \frac{\Im_x\{\hat{W}(m,n)\}}{\Im_x\{\Delta W_x(m,n)\}} = \frac{1 - \cos(\Omega_x) + i\,\mathrm{sen}(\Omega_x)}{2k_x[2 - 2\cos(\Omega_x)]}}$$

. (4.17a)

and it's magnitude is,

$$\boxed{\left|H_{xFH}^{-1}(\Omega_x)\right| = \frac{\left\{[1 - \cos(\Omega_x)]^2 + \mathrm{sen}^2(\Omega_x)\right\}^{1/2}}{2k_x[2 - 2\cos(\Omega_x)]}}$$

.

Similarity, in *y*-direction

$$\boxed{H_{yFH}^{-1}(\Omega_y) = \frac{\Im_y\{\hat{W}(m,n)\}}{\Im_y\{\Delta W_x(m,n)\}} = \frac{1 - \cos(\Omega_y) + i\,\mathrm{sen}(\Omega_y)}{2k_y[2 - 2\cos(\Omega_y)]}}$$

(4.17b)

The equations (4.17a) and (4.17b) may be regarded as the frequency response of the inverse filter used to recover the sheared information along the *x* and *y*-direction respectively using Fried-Hudgin method.

As was shown in chapter 3, the lateral shear interferometer ideally can be considered a linear system, so on, the inverse filters regularized in *x* and *y*-direction can be cascaded, then

$$H_{FH}^{-1}(\Omega_x,\Omega_y) = H_{xFH}^{-1}(\Omega_x)H_{yFH}^{-1}(\Omega_y) \qquad \text{with} \quad |\Omega_x| \le \pi \qquad \text{and} \quad |\Omega_y| \le \pi$$

In figure 4.8 is presented in a block diagram of the inverse filter of Fried-Hudgin method (H_{FH}^{-1}) used to recover $\Im\{\hat{W}(m,n)\}$.

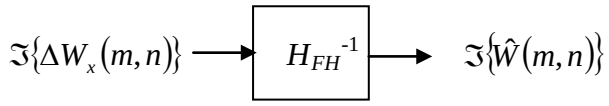


Figure 4.8 Block diagram of the inverse filter of the method of Fried-Hudgin used to recover $\Im\{\hat{W}(m,n)\}$.

4.8 Conclusions.

The process involved in the Lateral Shear Interferometer produces that some frequencies in the original wave-front to be lost. This produce the effect that is shown in figure 4.3 and figure 4.4 and when the lateral displacement is large enough the finite pupils produce the effect that is shown in the figure 4.5. So on in order to solve this problem Fried-Hudgin proposed a method from Least squares integration, i.e. integrating slop wave-front data the problem of this is that is restricted to small lateral shear. Under this circumstance the minimization problem has a well-defined and unique minimum. For larger lateral shear we need to regularize the problem to produce a stable solution. The regularization terms were proved as membrane and plate potentials.

References.

D. L. Fried, “Least-squares fitting a wave-front distortion estimate to an array of phase-difference measurements”, J. Opt. Soc. Am. 67, 370-375 (1977).
D. C. Ghiglia and L.A. Romero, “Direct phase estimation from phase differences using fast elliptic partial differential equation solvers”, Opt. Lett. 14, 1107-1109 (1989).
J. Hadamard, “Sur les problems aux derives partielles et leur signification physique”, Princeton Unioversity Bulletin 13 (Princeton University, Princeton, N. J., 1902).
R. H. Hudgin, “Wave-front reconstruction for compensated imaging”, J. Opt. Soc. Am. 67, 375-378 (1977).
B. R. Hunt, “Matrix Formulation of the reconstruction of phase values from phase differences”, J. Opt. Soc. Am. 69, 393-399 (1979).
R. J. Noll, “Phase estimates from slope-type wave-front sensors”, J. Opt. Soc. Am. 68, 139-140 (1978).
M. Servin, D. Malacara and J. L. Marroquin, “Wave-front recovery from two orthogonal sheared interferograms”, Appl. Opt., 35, 4343 (1996).
H. Takajo and Takahashi, “Least-squares phase estimation from the phase difference”, J. Opt. Soc. Am. A 5, 416-425 (1988).
A. N. Thikonov, “Solution of incorrectly formulated problems and the regularization method”, Sov. Math. Dokl. 4, 1035-1038 (1963).

5 Analysis of loss of information due to large displacement in the lateral shear interferometer and recovery of the wave-front using least squares with regularization.

5.0 Objectives of this Chapter.

Use the technique of Least squares summation with regularization for large lateral shear. Analysis of lost of certain points in the wave-front due a large lateral shear upon the pupil's geometry. Estimation of the losses points in the wave-front by means of membrane and plate potentials. Representation of the linear set of equation in the quadratic functional in order to show the inconsistency in the solution of the linear set of equations, i.e. its determinant is cero due to the lost of information in the original wave-front. Show how the regularization introduces information that produces in the matrix representation a determinant different of cero for larger lateral shear than one CDD-pixel. Show how the regularization introduces error in high frequencies because is a smoothed filter.

5.1 Introduction.

The use of shearing interferometry in optical testing dates from papers by Bates (Bates,1947). In an early paper, he developed a complex shearing interferometric technique for measuring the asphericity of a wave-front with white light. Drew (Drew,1951) explored a simplification of the Bates system and discussed the applications to a wider variety of testing problems. Murty (Murty,1964) showed the use of a laser to obtain a high intensity interference pattern from a shearing plate. Murty provided photographic examples but did not provide a quantitative treatment of the approach. There are lateral shear interferometers that are highly robust to mechanical vibrations, cost little, and are easy to build. Among the best known are the Ronchi interferometer (Cornejo-Rodriguez,1992 and Rimmer,1975) that uses only a diffracting Ronchi ruling and the lateral shear interferometers of the Murty type.

Lateral shearing interferometry has been used extensively in diverse applications such as the testing of optical components, systems, optical alignment (Dickey,1978), collimation (Sirohi,1987a) and determination of the refractive index of a lens (Kasana,1983a) and the study of flow and diffusion phenomena in gases and liquids (Mantravadi,1992). DeVany (DeVany,1971) discussed the use of a shearing interferometer using a testing plate to test the homogeneity of optical materials.

The standard least-squares procedure to integrate the phase obtained by these interferograms [originally proposed by Fried (Fried,1977) and Hudgin (Hudgin,1977)] uses an optimal linear estimator to find the desired wave-front , but this approach assumes that the lateral shear is so small that the lateral sheared wave-front may be considered the representation of the wave-front slope (Fried,1977, Hudgin,1977, Noll,1978, Hunt,1979, Takajo,1988 and Ghiglia,1989). Nevertheless this situation does not hold in several

interesting cases where a large shear is introduced in order to improve the sensitivity of the test.

With the information obtained from two orthogonal lateral sheared interferograms, we cannot recover in a unique way the wave-front that has produced it. The reason as review later, is that the spatial frequency response of the lateral shear interferometer has zeros within the region of interest (Servin,1996 and Wyant,1973). In summary, the shearing operation is not invertible in a unique way. That is there are poles in the *simple* inverse transfer function. The recovery of a wave-front from its sheared information is an ill-posed inverse problem in the sense of Hadamard (Hadamard,1902). In the frequency response of a lateral shear interferometer there are some frequencies with zero frequency response, these frequency components of a wave-front are lost, the only thing that we can do is estimate them, so the inverse problem has infinite solutions. This problem may be turned into a well-posed one (by regularization) when some prior information about the expected shape of the wave-front under analysis is given to the wave-front recovery system. A typical way used to regularize an ill-posed problem is to assume that the solution to the problem is smooth. Therefore assuming that all the wave-fronts of interest to us are smooth functions, we may convert the ill-posed problem into an invertible problem by using regularization theory (Thikonov,1963).

In a previous work presented by M. Servin (Servin,1996) he used the quadratic cost functional to estimate the wave-front, but he did not study large lateral shear displacements. The traditional idea using a lateral shear interferometer is introduce a small displacement to obtain the derivative of the wave-front under test, loosing sensitivity due to this small lateral displacement.

5.2 Theoretical Basis.

Considering the lateral shearing of an optical wave-front, using a Murty interferometer (figure 5.1), will be assume that the lateral shearing of the wave-front is along the x-direction; then the shearogram will have a fringe pattern that may be modeled as

$$I_x(x,y)=a(x,y)+b(x,y)\cos\left[\frac{2\pi}{\lambda}\left[W(x+\delta_x,y)-W(x-\delta_x,y)\right]\right]P(x+\delta_x,y)P(x-\delta_x,y), \qquad (5.1)$$

where $I_x(x,y)$ is the recorded interferogram's intensity, $a(x,y)$ is the low-frequency background illumination, $b(x,y)$ is a low-frequency amplitude modulation, $2\delta_x$ is the amount of lateral shear introduced in the x direction, and $P(x,y)$ is the aperture (or pupil function) that limits the extent of the wave-front under analysis, $W(x,y)$; the function $P(x,y)$ equals one within the beam of light where the wave-front being analyzed propagates and equal zero otherwise.

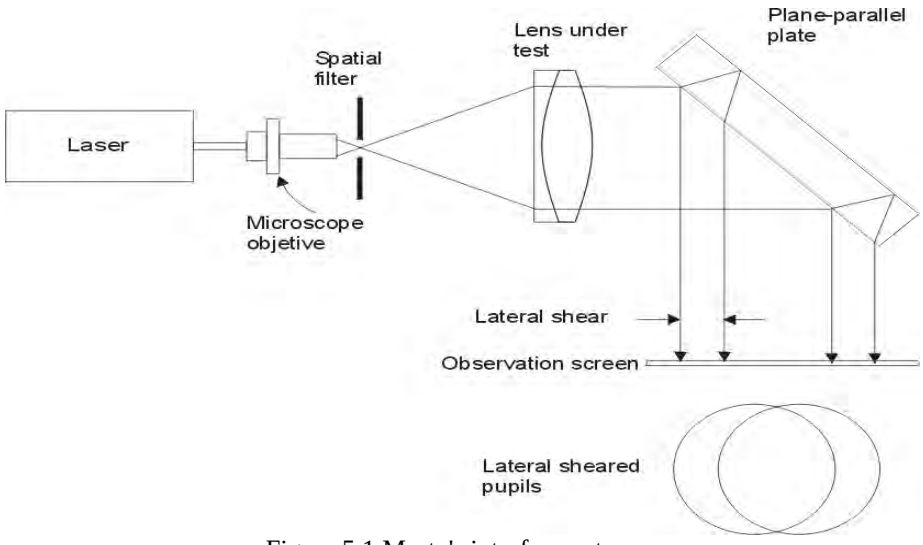


Figure 5.1 Murty's interferometer.

There are several techniques to extract the phase information from the interferogram, for example the phase stepping technique [see Appendix C]. When carrier fringes are introduced the Takeda's technique [see Appendix C] is usually the first choice. Once a given fringe demodulating technique is applied. The detected and unwrapped phase of the interferogram bounded is:

$$\Delta W_x(x,y) = \left[W(x+\delta_x,y) - W(x-\delta_x,y)\right] P(x+\delta_x,y) P(x-\delta_x,y), \qquad (5.2)$$

where $P(x+\delta_x,y)P(x-\delta_x,y)$ is the overlapping region of the sheared wave-front aperture, figure 5.2. S is a finite region defined where the product of the lateral sheared pupil function $P(x+\delta_x,y)P(x-\delta_x,y)$ is different from zero. When no displacement is effected ($\delta_x = 0$), S is defined on the wave-front aperture $P(x,y)$. The displacement is equal to $2\delta_x$ which can be any value in the domain $\delta_x \leq M_y/2$, where M_y is the segment defined on S region in x-direction (displacement direction).

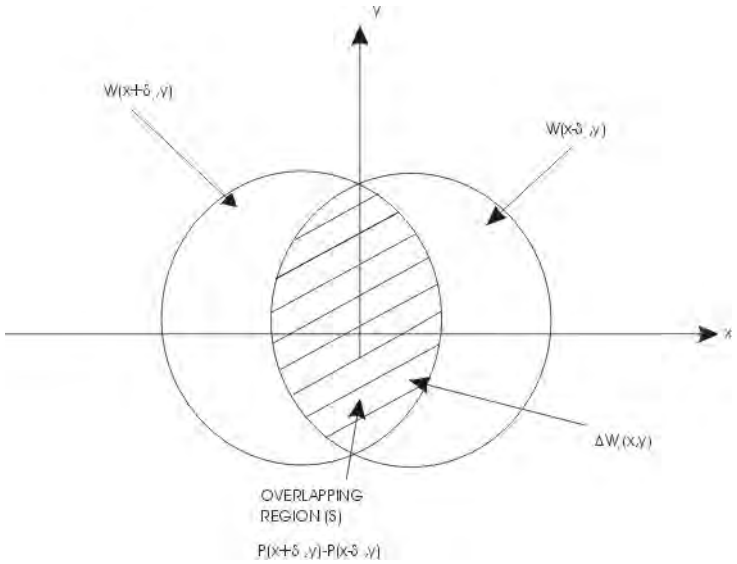


Figure 5.2 Lateral sheared pupils.

The fringes in the shearogram, equation (5.1), are modulated by the argument of the cosine function, i.e. the phase difference unwrapped given in equation (5.2). When the displacement in the interferometer is little, $2\delta \to 0$ the modulation is small then the following approximation is valid,

$$W(x+\delta_x, y) - W(x-\delta_x, y) \approx 2\delta \frac{\partial W(x,y)}{\partial x}, \qquad (5.3)$$

this approximation allow use the integration method to recover the wave-front. But from equation (5.3) can be observed that the sensitivity decrease as $2\delta \to 0$. To avoid this problem and obtain higher sensitivity large displacements are used. In order to recover the wave-front an optimal linear estimator can be used.

The amount of shearing obtained from a Murty lateral shear interferometer (in CCD pixels) is, in general, a real number, but it is easier to deal with shearing interferograms that have a displacement of an integer number of pixels of the CCD video camera used to take the image. To overcome this problem can be used the closest integer number (δ_x') to the actual real lateral shearing (δ_x) and then correcting the corresponding wave-front difference, using a linear approximation,

$$\Delta W_x'(x,y) = \frac{\delta'}{\delta} \Delta W_x(x,y). \qquad (5.4)$$

The corrected field $\Delta W_x'(x,y)$ is then used instead of the observed field $\Delta W_x(x,y)$ to yield to the estimation $\hat{W}(x,y)$ of the wave-front under analysis. The error introduced by this approximation is usually negligible, in particular for large shearing (Servin,1996). After

this lineal transformation of quantization [equation (5.4)] x and y will be discrete variables. The equation (5.2) can be rewrite as:

$$\Delta W_x'(x,y)=\frac{\delta'}{\delta}\left[W(x+\delta_x',y)-W(x-\delta_x',y)\right]P(x+\delta_x',y)P(x-\delta_x',y), \qquad (5.5)$$

equation (5.5) show the discrete case for two-dimensions.

The lateral shear displacement in the orthogonal direction (y-direction) is equivalent.

For clarity will be consider one line on the x axis of the original wave-front $W(x,y)$ which will be described as the function $W(x)$ in one-dimension, as shown in figure 5.3. $\hat{W}(x)$ will be estimated on that line. The detected and unwrapped phase on the x-line is:

$$\Delta W(x)=\left[W(x+\delta)-W(x-\delta)\right]P(x+\delta)P(x-\delta), \qquad (5.6)$$

where $\Delta W(x)$ is the detected phase of the shearogram bounded by the overlapping region of the sheared wave-front aperture, $P(x+\delta)P(x-\delta)$, $M_{y=0}$ is a finite segment defined where the product of the lateral sheared pupil function $P(x+\delta)P(x-\delta)$ is different from zero. When no displacement is effected ($\delta=0$), $M_{y=0}$ is defined on the original wave-front aperture $M_{y=0}=2N$ where N is the half of the size of the original pupil. The displacement is equal to 2δ and δ can take any value in the domain $\left(0, {}^{M_{y=0}}\!/_{2}\right)$.

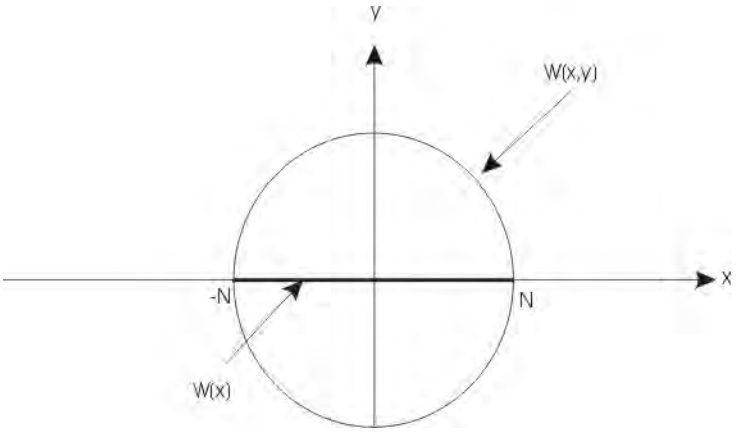


Figure 5.3 Original wave-front.

Similarly to two-dimensions will be considered an linear approximation to represent the discrete version of the lateral shearing

$$\Delta W'(x) = \frac{\delta'}{\delta}\Delta W(x). \tag{5.7}$$

The corrected field $\Delta W'(x)$ is then used instead of the observed field $\Delta W(x)$ to yield our estimation $\hat{W}(x)$ of the wave-front under analysis. The x-variable will be discrete. The equation (5.6) can be rewrite as:

$$\Delta W'(x) = \frac{\delta'}{\delta}[W(x+\delta') - W(x-\delta')]P(x+\delta')P(x-\delta'), \tag{5.8}$$

equation (5.8) show the discrete case for one-dimension. Where $\Delta W'(x)$ is the corrected phase of the shearogram bounded by the overlapping region of the sheared wave-front's aperture, $P(x+\delta')P(x-\delta')$. $M'_{y=0}$ is a finite one-dimension sequence defined where the product of the lateral sheared pupil function $P(x+\delta')P(x-\delta')$ is different from zero. When no displacement is effected ($\delta' = 0$), $M'_{y=0}$ is defined on the original wave-front's aperture $M'_{y=0} = 2N'$ where N' is the half of the size of the original pupil quantized. The total displacement is equal to $2\delta'$ and δ' can take any value in the sequence $\{0,1,2,\ldots,N'\}$.

5.3 Frequency analysis of the Lateral Shear Interferometer.

Let be consider that the extent of the aperture is large enough that $P(x)$ may be considered equal to one from $-\infty$ to ∞ domain. Taking the Fourier transform [see Appendix B for a reference of the Fourier Transform] of equation (5.6) we obtain the transfer function of the lateral shear interferometer,

$$H_{LSI}(\omega) = \frac{\Im\{\Delta W(x)\}}{\Im\{W(x)\}} = 2i\,\mathrm{sen}(\delta'\omega), \qquad -\pi \le \omega \le \pi \tag{5.9}$$

the spatial frequencies with zero frequency response are

$$\omega = \frac{n\pi}{\delta'}, \qquad n = 0,\pm1,\pm2,\pm3,... \tag{5.10}$$

At the spatial frequencies given by equation (5.10), the information about the wave-front under test is lost; this is the reason that the wave-front detection from a lateral sheared interferogram is an ill-posed problem. So the *simple* inverse transfer function is given by the following relation,

$$H^{-1}_{LSI}(\omega) = \frac{\Im\{W(x)\}}{\Im\{\Delta W(x)\}} = \frac{-i}{2\,\mathrm{sen}(\delta'\omega)}. \tag{5.11}$$

Solving for $\Im\{W(x)\}$ on the relation (5.11) and taking the inverse Fourier transform, the wave-front is recovered, but it have two main limitations: this *simple* inverse transfer function consider infinite pupils and it has poles at frequencies given by equation (5.10). In order to overcome this two limitations a quadratic cost functional with regularization is used to introduce *a priori* information.

5.4 Wave-front Estimation from Sheared Unwrapped Phase[1].

To recover the original wave-front $W(x)$ from $\Delta W'(x)$ one may choose an estimated function $\hat{W}(x)$, with the x discrete variable on the domain $\{-N',\ldots,-2,-1,0,1,2,\ldots,N'\}$, so that it minimizes the least squared error U, given by:

$$U = \sum_{x=-N'}^{N'} \left[\hat{W}(x+\delta') - \hat{W}(x-\delta') - \Delta W'(x)\right]^2 P(x+\delta')P(x-\delta'). \qquad (5.12)$$

Equation (5.12) is a functional that only have one global minimum because it is a positive quadratic cost functional. Taking its derivative with respect to $\hat{W}(x)$

$$\begin{aligned}&\left[\hat{W}(x) - \hat{W}(x-2\delta')\right]P(x-2\delta')P(x) - \left[\hat{W}(x+2\delta') - \hat{W}(x)\right]P(x)P(x+2\delta') = \\ &\Delta W'(x-\delta')P(x-2\delta')P(x) - \Delta W'(x+\delta')P(x)P(x+2\delta')\end{aligned}, \qquad (5.13)$$

$$(x = -N',\ldots,-2,-1,0,1,2,\ldots,N')$$

equating it to zero and solving for $\hat{W}(x)$, we end up with a linear set of equations, which solution is the global minimum of the functional. For facility in matrix representation the domain of $\hat{W}(x)$ has been translated to a positive domain getting the function displaced. This is realized with a change of variable $(x' = x + N')$ which translate the solution on the x' space. It is equivalent to take the derivative of equation (5.12) with respect to $\hat{W}(x-N')$ equating it to zero and solving the linear set of equations for $\hat{W}(x-N')$ on the space of x, then

$$\begin{aligned}&\left[\hat{W}(x'-N') - \hat{W}(x'-N'-2\delta')\right]P(x'-N'-2\delta')P(x'-N') \\ &-\left[\hat{W}(x'-N'+2\delta') - \hat{W}(x'-N')\right]P(x'-N')P(x'-N'+2\delta') = \\ &\Delta W'(x'-N'-\delta')P(x'-N'-2\delta')P(x'-N') - \Delta W'(x'-N'+\delta')P(x'-N')P(x'-N'+2\delta')\end{aligned} \qquad (5.14)$$

$$(x' = 0,1,2,\ldots,2N')$$

The equation (5.14) is a set of linear equations that can be represented in a matrix form. Lets consider an example, if $\delta' = 1$ pixel (remember the total displacement is $2\delta'$) then,

[1] In Appendix C can be seen different techniques to get unwrapped phase maps.

$$A \bullet \begin{pmatrix} \hat{W}(0) \\ \hat{W}(1) \\ \hat{W}(2) \\ \hat{W}(3) \\ \hat{W}(4) \\ \hat{W}(5) \\ \vdots \\ \hat{W}(2N'-5) \\ \hat{W}(2N'-4) \\ \hat{W}(2N'-3) \\ \hat{W}(2N'-2) \\ \hat{W}(2N'-1) \\ \hat{W}(2N') \end{pmatrix} = \begin{pmatrix} -\Delta W'(1-N') \\ -\Delta W'(2-N') \\ \Delta W'(1-N') - \Delta W'(3-N') \\ \Delta W'(2-N') - \Delta W'(4-N') \\ \Delta W'(3-N') - \Delta W'(5-N') \\ \Delta W'(4-N') - \Delta W'(6-N') \\ \vdots \\ \Delta W'(N'-6) - \Delta W'(N'-4) \\ \Delta W'(N'-5) - \Delta W'(N'-3) \\ \Delta W'(N'-4) - \Delta W'(N'-2) \\ \Delta W'(N'-3) - \Delta W'(N'-1) \\ \Delta W'(N'-2) \\ \Delta W'(N'-1) \end{pmatrix}, \qquad (5.15a)$$

where A is a squared matrix of size $2N$.

$$A = \begin{pmatrix} 1 & 0 & -1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 2 & 0 & -1 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 2 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 2 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & -1 & 0 & 2 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 0 & 2 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & -1 & 0 & 2 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & -1 & 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}.$$

The equation (5.15a) can be written in a compact form as:

$$A\,\hat{W}(x'-N') = b, \qquad (5.15b)$$

where b is the data, and is given by

$$b = \Delta W'(x'-N'-\delta')P(x'-N'-2\delta')P(x'-N') - \Delta W'(x'-N'+\delta')P(x'-N')P(x'-N'+2\delta').$$

Matrix A cannot be inverted because $\det[A]=0$ for $2\delta' \geq 1$ pixel, a direct method like Gauss and its variants cannot be used. Information *a priori* about the original wave-front $W(x)$ is introduced in the functional, equation (5.12), through regularization terms to make that its determinant be different from zero. It realizes that the matrix can be inverted. A simple kind of regularization term can be the first-order potentials or membrane potentials. These potentials assume a slow slope variation in the solution, this means that two discrete consecutive values of $\hat{W}(x)$ are approximately the same. The first-order potential function is:

$$R(x) = \left[\hat{W}(x) - \hat{W}(x-1)\right] P(x) P(x-1). \tag{5.16}$$

The α constant weights up the effect of the first-order potential introduced on the functional. Then the augmented functional becomes,

$$U = \sum_{x=-N'}^{N'} \left\{ \left[\hat{W}(x+\delta') - \hat{W}(x-\delta') - \Delta W'(x)\right]^2 P(x+\delta') P(x-\delta') + \alpha R^2(x) \right\}. \tag{5.17}$$

Taking the derivative of equation (5.17) with respect to $\hat{w}(x)$

$$\begin{aligned} &\left[\hat{W}(x) - \hat{W}(x-2\delta')\right] P(x-2\delta') P(x) - \left[\hat{W}(x+2\delta') - \hat{W}(x)\right] P(x) P(x+2\delta') \\ &+ \alpha\left[R(x) - R(x+1)\right] = \\ &\Delta W'(x-\delta') P(x-2\delta') P(x) - \Delta W'(x+\delta') P(x) P(x+2\delta') \end{aligned} \quad , \tag{5.18}$$

$$(x = -N', \ldots, -2, -1, 0, 1, 2, \ldots, N')$$

equating it to zero and making a change of variable $(x' = x + N')$ we end up with the set of linear equations represented by

$$\begin{aligned} &\left[\hat{W}(x'-N') - \hat{W}(x'-N'-2\delta')\right] P(x'-N'-2\delta') P(x'-N') \\ &- \left[\hat{W}(x'-N'+2\delta') - \hat{W}(x'-N')\right] P(x'-N') P(x'-N'+2\delta') + \alpha\left[R(x'-N') - R(x'-N'+1)\right] = . \\ &\Delta W'(x'-N'-\delta') P(x'-N'-2\delta') P(x'-N') - \Delta W'(x'-N'+\delta') P(x'-N') P(x'-N'+2\delta') \end{aligned}$$

$$(x' = 0, 1, 2, \ldots, 2N') \tag{5.19}$$

Showing this set of linear equations in matrix form, with $\delta' = 1$ pixel,

$$A \bullet \begin{pmatrix} \hat{W}(0) \\ \hat{W}(1) \\ \hat{W}(2) \\ \hat{W}(3) \\ \hat{W}(4) \\ \hat{W}(5) \\ \vdots \\ \hat{W}(2N'-5) \\ \hat{W}(2N'-4) \\ \hat{W}(2N'-3) \\ \hat{W}(2N'-2) \\ \hat{W}(2N'-1) \\ \hat{W}(2N') \end{pmatrix} = \begin{pmatrix} -\Delta W'(1-N') \\ -\Delta W'(2-N') \\ \Delta W'(1-N') - \Delta W'(3-N') \\ \Delta W'(2-N') - \Delta W'(4-N') \\ \Delta W'(3-N') - \Delta W'(5-N') \\ \Delta W'(4-N') - \Delta W'(6-N') \\ \vdots \\ \Delta W'(N'-6) - \Delta W'(N'-4) \\ \Delta W'(N'-5) - \Delta W'(N'-3) \\ \Delta W'(N'-4) - \Delta W'(N'-2) \\ \Delta W'(N'-3) - \Delta W'(N'-1) \\ \Delta W'(N'-2) \\ \Delta W'(N'-1) \end{pmatrix}, \qquad (5.20a)$$

where A is a square matrix of size $2N$

$$A = \begin{pmatrix} 1+\alpha & -\alpha & -1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ -\alpha & 1+2\alpha & -\alpha & -1 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ -1 & -\alpha & 2+2\alpha & -\alpha & -1 & \cdots & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -\alpha & 2+2\alpha & -\alpha & \cdots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -\alpha & 2+2\alpha & \cdots & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 2+2\alpha & -\alpha & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & -\alpha & 2+2\alpha & -\alpha & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & -1 & -\alpha & 2+2\alpha & -\alpha & -1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & -1 & -\alpha & 1+2\alpha & -\alpha \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & -1 & -\alpha & 1+\alpha \end{pmatrix}.$$

The equation (5.20a) can be written in a compact form as:

$$A\,\hat{W}(x'-N') = b, \qquad (5.20b)$$

where b is the data, and is given by

$$b = \Delta W'(x'-N'-\delta')P(x'-N'-2\delta')P(x'-N') - \Delta W'(x'-N'+\delta')P(x'-N')P(x'-N'+2\delta').$$

The matrix A can be inverted for whatever δ' in the range allowed ($\delta' = 0,1,2,\ldots,2N'$). This linear equation system can be resolved by direct methods like Gauss or its variants. The use of an indirect or iterative method like Conjugate Gradient

(CG) [see Appendix E], require the following two conditions: first, that its matrix to be symmetric i.e.: $A = A^T$, and second, that its matrix be positively defined, i.e.: $\hat{W}^T A\hat{W} > 0$.

To study the frequency response of the *regularized* inverse transfer function given by equation (5.20), assuming an aperture of infinite size. Taking the Fourier transform on both sides of equation (5.20), then

$$\left|H_R^{-1}(\omega,\alpha)\right| = \left|\frac{\Im\{\hat{W}(x)\}}{\Im\{\Delta W(x)\}}\right| = \frac{2sin(\delta'\omega)}{2-2\cos(2\delta'\omega)+\alpha[2-2\cos(\omega)]}. \qquad (5.21)$$

Solving for $\Im\{\hat{W}(x)\}$ on the relation (5.21) and taking the inverse Fourier transform, the wave-front is recovered, but it have one main limitations: this *regularized* inverse transfer function consider infinite pupils. When $\alpha = 0$ the equation (5.21) is reduced to the *simple* inverse transverse function, equation (5.11). A difference with equation (5.11) is that it has zeros instead of poles at frequencies given by equation (5.10).

5.5 Results.

5.5.1 Analysis of lost of information due to large displacements.

a. One dimension.

As has been indicated large lateral shear displacement have higher sensitivities, but, how large can be that displacements? As is known, in interferometry we consider that always exist the coherent superposition of two wave-fronts. In the case of LSI the interference occur when the wave-front is superposed with itself displaced. If for each point of the wave-front exist the superposition with another point of the displaced wave-front then fringes will appear due to the optical path difference and the interference will exist. Now from the figure 4 the loss of data will be analyzed. Suppose that as is show in the figure 5.4(a), $W(x)$ is any wave-front where its limits are the points a and b, and its center is the point c. In the figure 5.4(b) are showed the displacements of the wave-front in the range $2\delta < N$, in this case all the points into the wave-front are superposed with another point of the displaced wave-front at least one time. In the figure 5.4(c) is showed a critical displacement $2\delta = N$ where the point a is superposed with the point c on the displaced wave-front and similarly b is superposed with c, which means that yet in this case all the points into the wave-front are superposed with another point of the displaced wave-front at least one time. In figure 5.4(d) are showed the displacement of the wave-front in the range $2\delta > N$ in this case some points no are superposed, for example the point c is lost because is not superposed with any other point. The maximum displacement is $2\delta = 2N$ greater displacements produce any superposed point. Then the loss of information start in the middle of the wave-front, in other words the central elements start being zeros in the data vector (this vector is the right-hand side of the equation (5.15a) or (5.20a)). In the

functional formulation the explicit bound of the wave-front is given by the pupil function. The effect of loss of information on the matrix A is that the elements in the half of the principal diagonal and the two lateral diagonals start doing zero. As has been explained in one-dimension problem it is very easy determine when the information is lost. The loosing of data start when the displacement holds the following inequality $2\delta > N$.

In the figure 5.5 is shown the percent of loss of data as function of the percent of displacement. In the range from 0 to 50%, that correspond to a real displacement of N, there is not loss of information. But for displacements over 50% the lost of data is given by a line with constant slope equal to two.

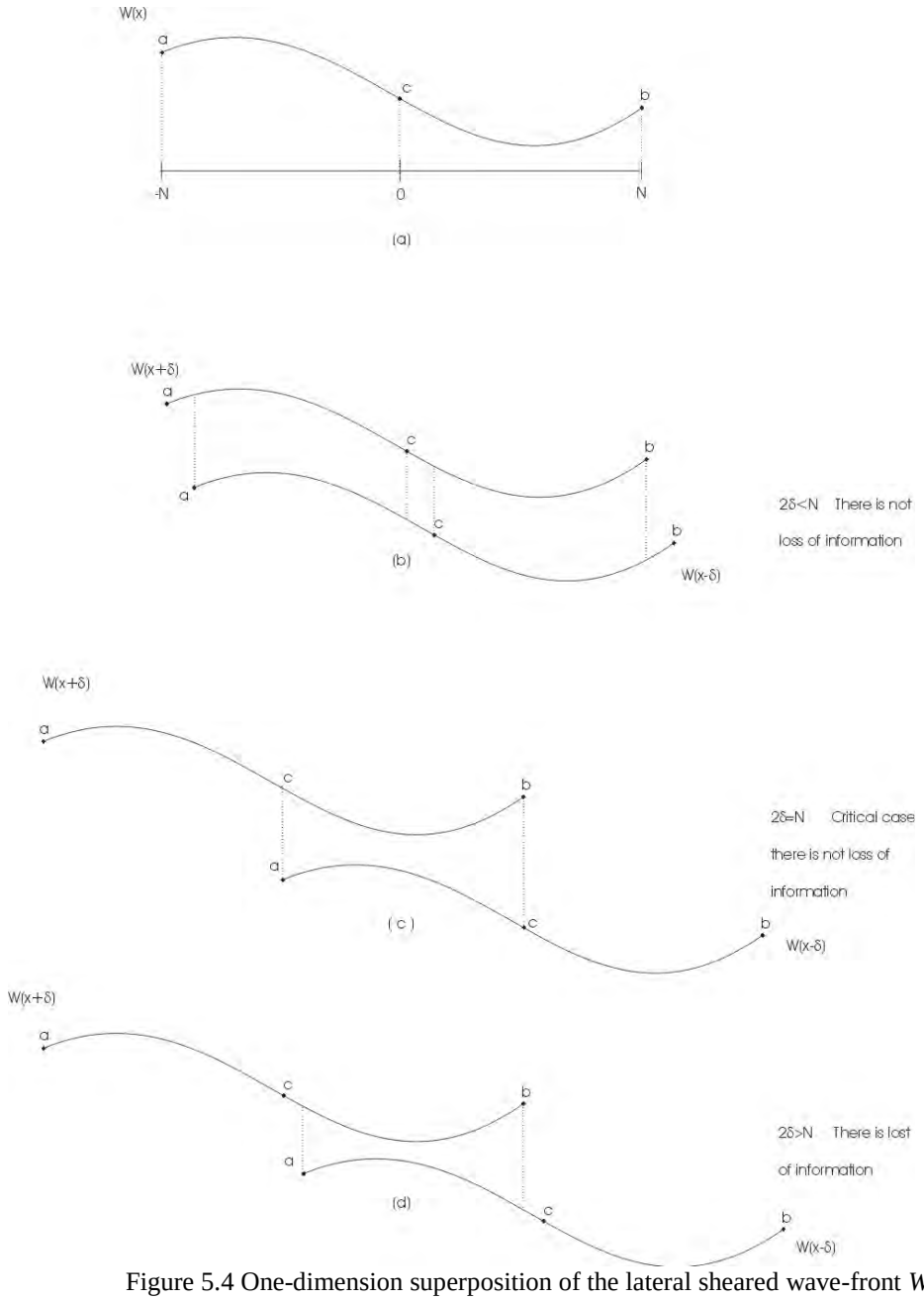


Figure 5.4 One-dimension superposition of the lateral sheared wave-front *W(x)*.

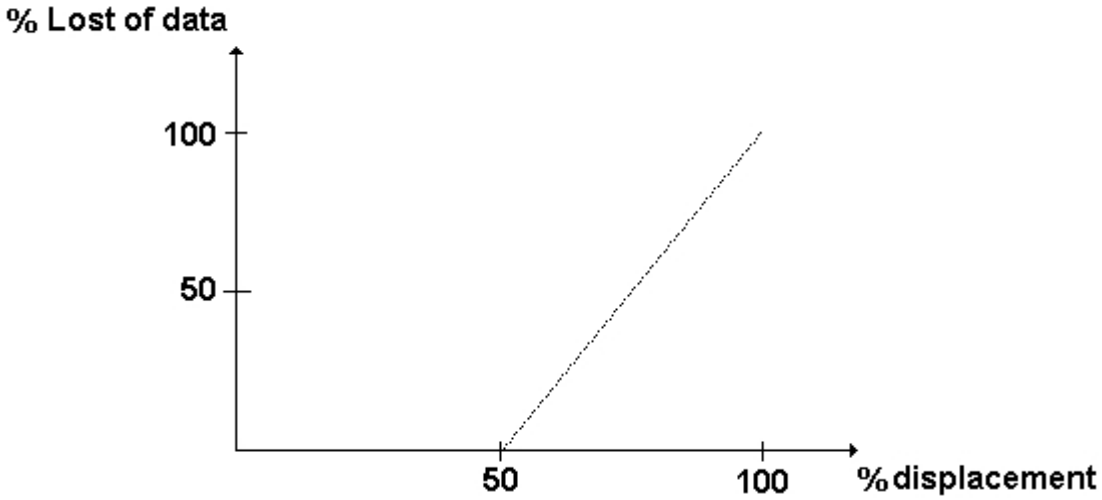


Figure 5.5 One-dimension curve of percent of lost of information vs percent of displacement.

b. Two dimensions.

In this section will be extend the analysis of one-dimension to two-dimensions for different geometry forms of the pupil function. In figure 5.6 (a), (b) and (c) is showed the region where loss of information exist in a square geometry for a lateral shear of 25%, 50% and 75% respectively. Black region represents the amount of loss of information. Its curve of percent of loss of information vs percent of displacement is shown in figure 5.7, it is identical to curve of loss of information in one-dimension. In figure 5.6 (d), (e) and (f) the same analysis is make for a circular geometry, in this case the loss of information begins since earlier that in the square case. An analytical treatment is made in order to estimate the loss of information,

$$A = A_T - A_R$$

where

$$A_T = 8\left[\frac{r^2}{2}\text{sen}^{-1}(1) - \frac{\delta'\sqrt{r^2-\left(\frac{\delta'}{2}\right)^2}}{4} - \frac{r^2}{2}\text{sen}^{-1}\left(\frac{\delta'}{2r}\right)\right] \qquad 0 < \delta' < 2r$$

$$A_R = \begin{cases} 4\left[\frac{r^2}{2}\text{sen}^{-1}(1) - \frac{\delta'\sqrt{r^2-\delta'^2}}{2} - \frac{r^2}{2}\text{sen}^{-1}\left(\frac{\delta'}{r}\right)\right] & 0 < \delta' \le r \\ 0 & r < \delta' \le 2r \end{cases}$$

and A is the area of the region with information, r is the radius of the pupil function. Its curve of percent of loss of information vs percent of displacement is shown in figure 5.8. In

figure 5.6 (g), (h) and (i) is shown the region of loss of information for a pupil like a primary mirror of Cassegrain telescope. Its curve of percent of loss of information vs percent of displacement is shown in figure 5.9. Finally in figure 5.6 (j), (k) and (l) is show the region of loss of information for an armour pupil and its curve of percent of loss of information vs percent of displacement is shown in figure 5.10.

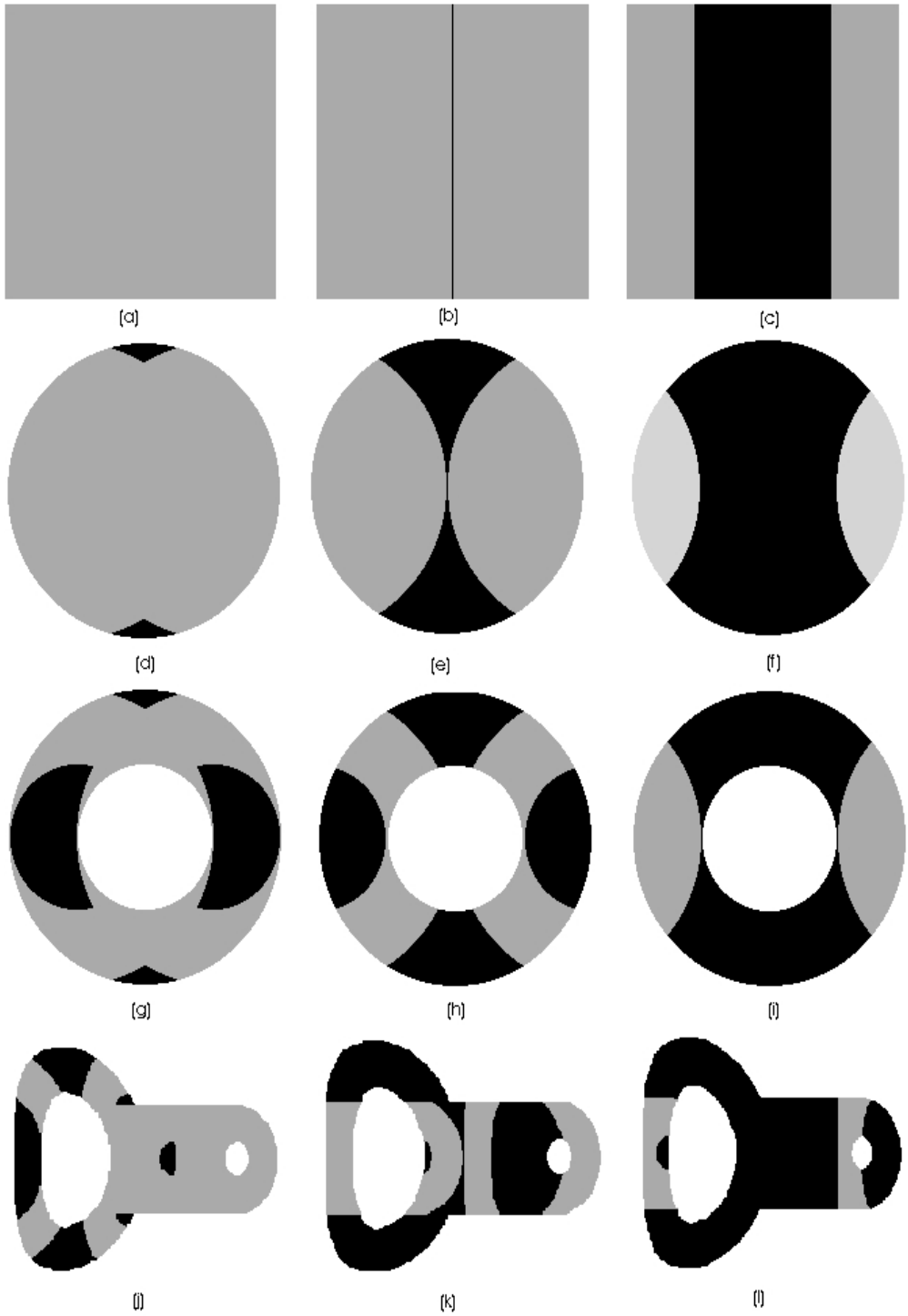


Figure 5.6 Regions of lost of information for different pupils function.

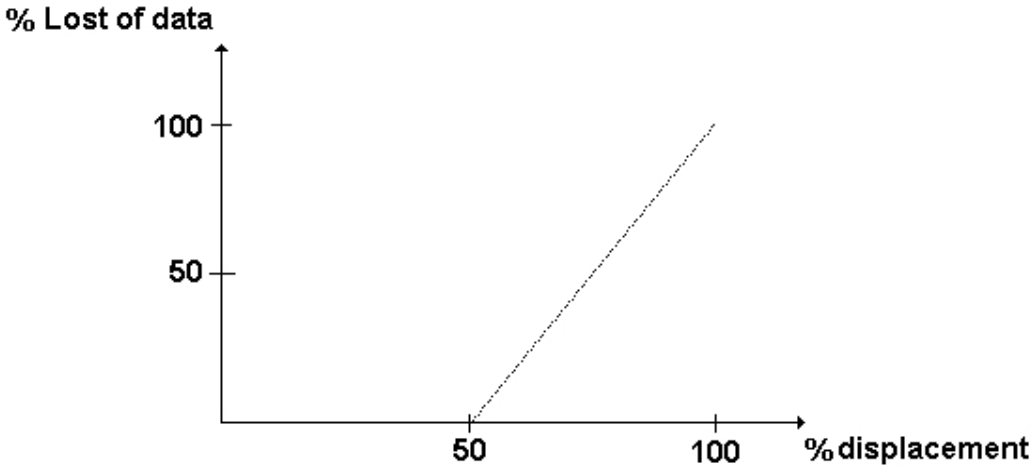


Fig. 5.7 Curve of percent of lost of information vs percent of displacement for a square aperture.

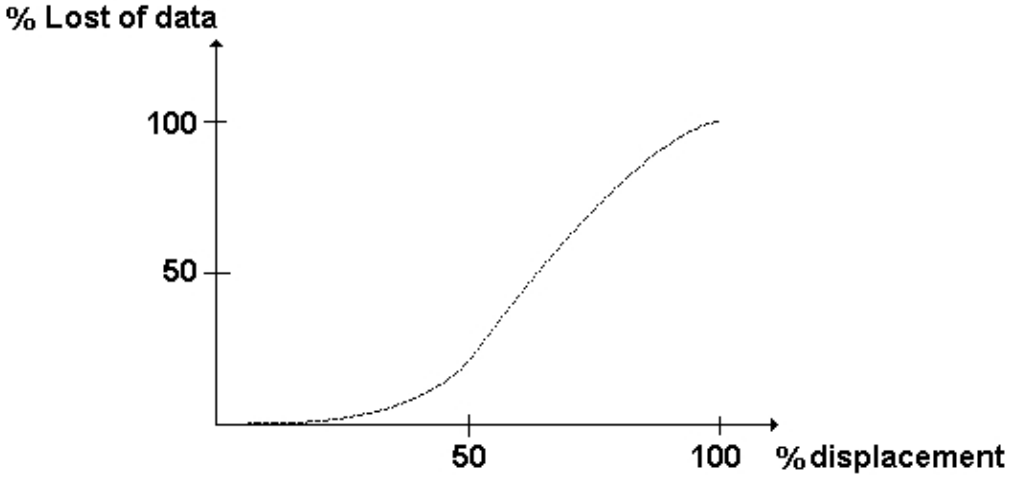


Figure 5.8 Curve of percent of lost of information vs percent of displacement for a circular aperture.

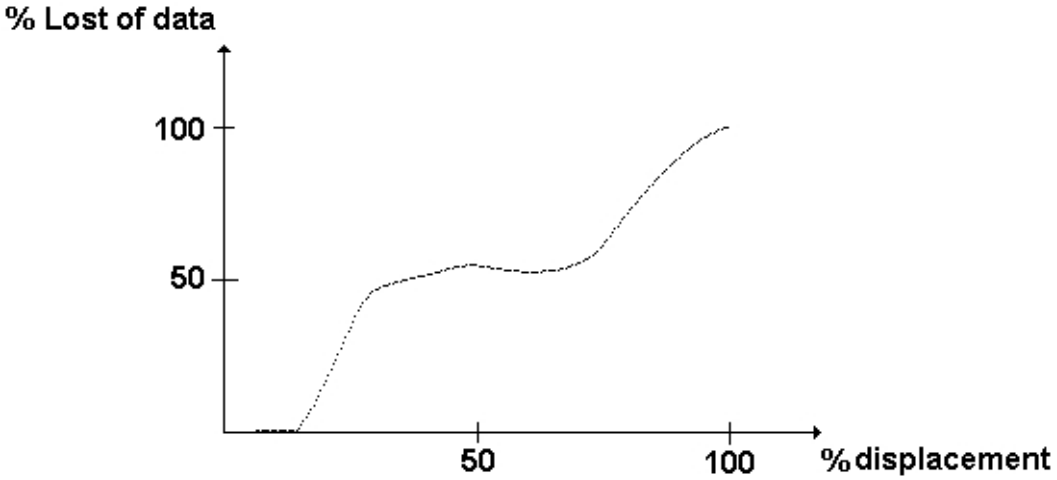


Figure 5.9 Curve of percent of lost of information vs percent of displacement for a primary mirror of a Cassegrain's telescope aperture.

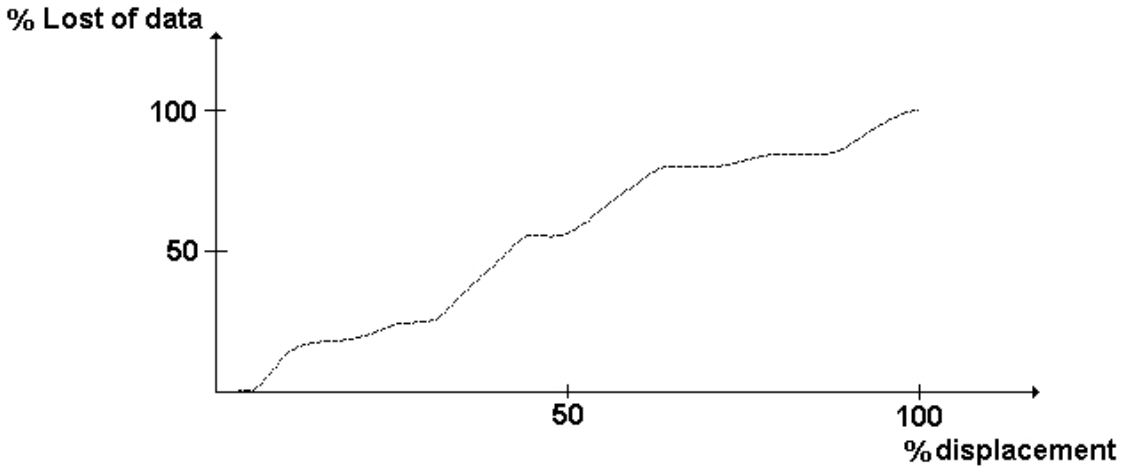


Figure 5.10 Curve of percent of lost of information vs percent of displacement for an armour aperture.

In summary the curves of loss of information for different geometric forms not follows a linear behavior. In the case of geometry of the pupil function like a telescope Cassegrain can be observed in figure 5.9 that higher losses exist with a displacement of 50% than a displacement of 60%.

5.5.2 Error produced in reconstruction of $W(x)$.

The amplitude of the entrance wave-front is strongly modified when the spatial frequencies of its frequency spectrum are equal to a multiple of $\frac{1}{2\delta'}$ as can be seen from equation (5.10), due zero frequency response of the regularized inverse transfer function. To calculate the square error of the estimation (σ), the following relation is used

$$\sigma^2(\omega;\alpha) = \sum_{x=-N'}^{N'} \left(W_\omega(x) - \hat{W}_\omega(x;\alpha)\right)^2 \qquad (5.22)$$

The percent of square error of the estimation is: $\%\sigma = \frac{\sigma}{\sigma_{max}} \times 100$.

In figure 5.11 are shown the effects of α (where α is the weight of regularization term or membrane potential) over the percent of square error of the estimation (σ). In this graph can be observed for a α given, a widening in the maximum error peaks at higher frequencies and an increment of the minimum error value. This is due to the membrane potentials try to maintain equal two consecutive values of the estimated wave-front, but for high frequencies is not satisfied. If α is incremented, σ is incremented too, then the estimation result obey mainly to regularization terms neither to data.

Therefore must be seek a minimum value of α that produce the regularization without introduce high error in the estimated result.

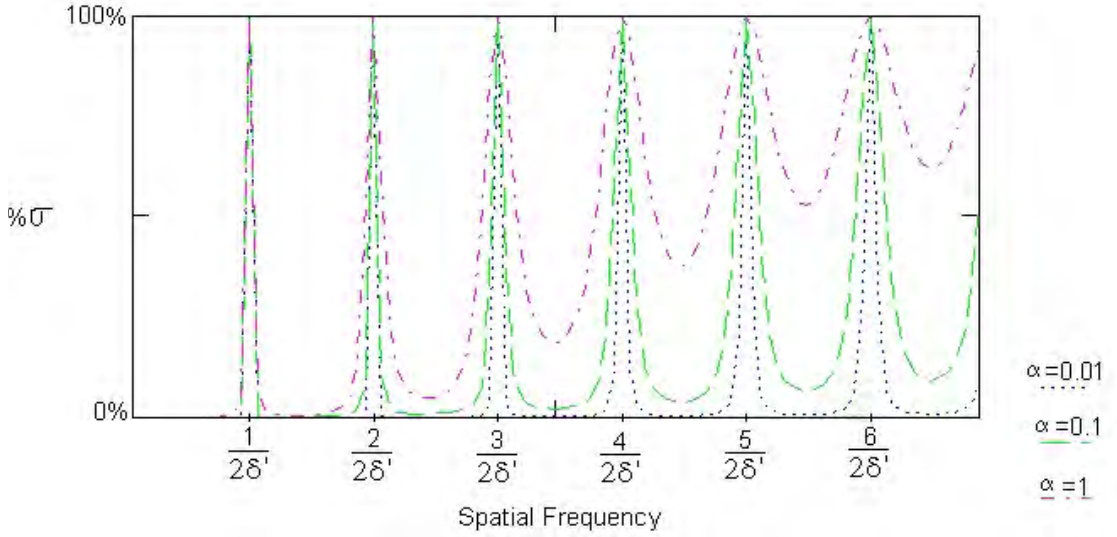


Figure 5.11 Graph of percent of quadratic error vs spatial frequency.

The membrane potentials also introduce interpolation in the elements in the middle of the wave-front that were make zero by the large displacements.

In figure 5.12, let be $W(x)$ the original wave-front and $m(x)$ a mask function which is zero in the range (A,B) and one otherwise. Therefore the product $W(x)m(x)$ produces that $W(x)$ be zero in the range (A,B), which will be the cut function. Let $\hat{W}(x)$ the estimated function of the cut function, and then in the range (A,B) the membrane potential will produce an interpolation between the points A and B joining them with a straight line.

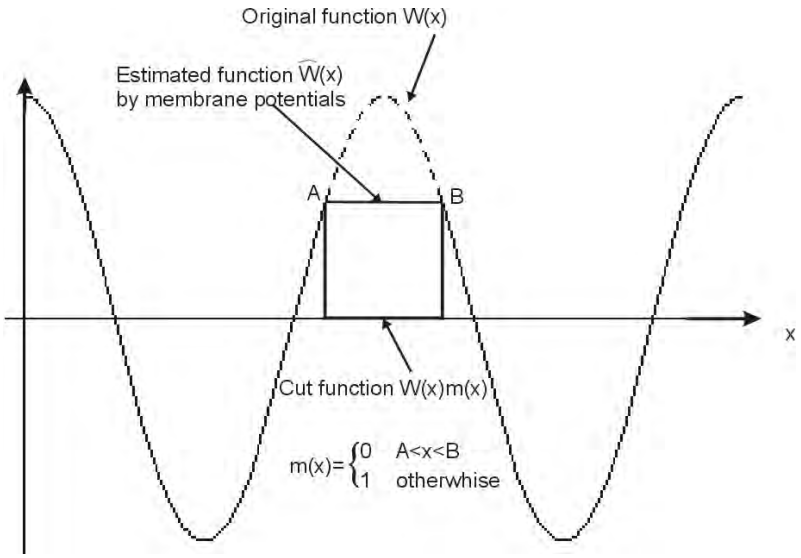


Figure 5.12 Graph of the estimated function $\hat{W}(x)$ from the cut function $W(x)m(x)$.

5.6 Conclusions.

In this work has been presented the analysis of the loss of information in a lateral shear interferometer. A method of solution by least square with regularization was presented in order to avoid the poles in the *simple* inverse transfer function. In this method is necessary to consider the weight of the regularization term α, due to if $\alpha = 0$ the *regularized* inverse transfer function is reduced to the *simple* inverse transfer function. But if α is higher then the effect of the terms of regularization dominate over the information. In lateral shear interferometry small lateral shear implies small sensitivity, then bigger displacement is convenient but it produce loss of information. Loss of information is analyzed to different geometry forms in order to consider how much can be the lateral shear. The loss of information curves are in general not linear, the only one is the curve of a square aperture, which is similar to the one-dimension curve. Analyzing the derivative of this curve with respect to the displacement can be estimated the displacements with higher ratio of loss of information. In the regions where loss of information is produced the membrane potentials acts as linear interpolators and in the regions with information act as smothers weighted by α.

References.

W. J. Bates, “A Wavefront Shearing Interferometer”, Proc. Phys. Soc. (London) 59, 940 (1947).
A. Cornejo-Rodriguez, “Ronchi test”, in Optical Shop Testing, D. Malacara, ed. (Wiley, New York, 1992), pp. 321-365.
A. S. DeVany, “Using a Murty Interferometer for testing the Homogeneity of Test Samples of Optional Materials”, Appl. Opt. 10, 1459 (1971).
Dickey, F. M. And T. M. Harder, “Shearing Plate Optical Alignment”, Opt. Eng., 17, 295 (1978).
R. L. Drew, “A Simplified Shearing Interferometer”, Proc. Phys. Soc. (London) B64, 1005 (1951).
D. L. Fried, “Least-squares fitting a wave-front distortion estimate to an array of phase-difference measurements”, J. Opt. Soc. Am. 67, 370-375 (1977).
D. C. Ghiglia and L.A. Romero, “Direct phase estimation from phase differences using fast elliptic partial differential equation solvers”, Opt. Lett. 14, 1107-1109 (1989).
J. Hadamard, “Sur les problems aux derives partielles et leur signification physique”, Princeton Unioversity Bulletin 13 (Princeton University, Princeton, N. J., 1902).
R. H. Hudgin, “Wave-front reconstruction for compensated imaging”, J. Opt. Soc. Am. 67, 375-378 (1977).
B. R. Hunt, “Matrix Formulation of the reconstruction of phase values from phase differences”, J. Opt. Soc. Am. 69, 393-399 (1979).
Kasana, R. S. And K. J. Rosenbruch, “Determination of the Refractive Index of a Lens Using the Murty Shearing Interferometer”, Appl. Opt., 22, 3526 (1983a).
M. V. Mantravadi, “Lateral shearing interferometers”, in Optical Shop Testing, D. Malacara, ed. (Wiley, New York, 1992), pp. 123-172.

M. V. R. K. Murty, “The Use of a Single Plane Parallel Plate as a Lateral Shearing Interferometer With a Visible Gas Laser Source”, Appl. Opt. 3, 531 (1964).
R. J. Noll, “Phase estimates from slope-type wave-front sensors”, J. Opt. Soc. Am. 68, 139-140 (1978).
M. P. Rimmer, and J. C. Wyant, “Evaluation of large aberrations using a lateral shear interferometer having variable shear”, Appl. Opt. 14, 142-150 (1975).
M. Servin, D. Malacara and J. L. Marroquin, “Wave-front recovery from two orthogonal sheared interferograms”, Appl. Opt., 35, 4343 (1996).
Sirohi, R. S. And M. P. Kothiyal, “Double Wedge Plate Shearing Interferometer for Collimation Test”, Appl. Opt., 26, 4054 (1987a).
H. Takajo and Takahashi, “Least-squares phase estimation from the phase difference”, J. Opt. Soc. Am. A 5, 416-425 (1988).
A. N. Thikonov, “Solution of incorrectly formulated problems and the regularization method”, Sov. Math. Dokl. 4, 1035-1038 (1963).
Wyant, J. C., “Double Frequency Grating Lateral Shear Interferometer”, Appl. Opt., 12, 2057 (1973).

6 Results and conclusions to the work.

6.0 Objectives of this Chapter.

Present the calculations of determinant on matrix [A] for different lateral shear. Show the results obtained using least squares minimization with regularization to estimate the original wave-front. We show the results obtained about the lost of information in a lateral shear interferometer for large shear and some examples of reconstruction of the wave-front.

6.1 Results.

As we show in chapter 4 the use of the regularizing potentials (membrane or plate) is a must, to yield a stable solution of the least squares integration for lateral displacements greater than one pixel. The method conjugate gradient [see appendix E] is used to solve the inverse problem given for example in equation (5.15), it exhibit regularization properties. The method has some peculiar filtering properties even if these properties cannot be described in terms of a global PSF because the method is basically non-linear (Bertero, 1998).

Large lateral displacements can produce that the matrix [A] in equation (5.20) has zeros on the main diagonal, so on it is possible that it's determinant to be zero. This can be observed in table 6.1.

The determinant is:		
δ	Membrane($\alpha = 0.1$)	Plate($\alpha = 0.1$)
1	$3.211X10^{-4}$	-0.11
2	$3.368X10^{-5}$	-0.027
3	$8.701X10^{-7}$	$-5.215X10^{-4}$
4	$1.289X10^{-8}$	$-4.354X10^{-6}$
5	$1.108X10^{-10}$	$-1.235X10^{-8}$
6	$8.153X10^{-13}$	$6.103X10^{-11}$
7	$2.214X10^{-15}$	$2.88X10^{-13}$
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0

Table 6.1. Calculation of det[A] in equation (5.20).

In figure 6.1 is shown an armour pupil function. The figure 6.2 show the lost of interfered data as function of lateral shear displacement in the armour pupil.

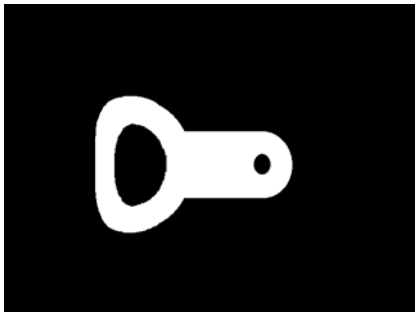


Figure 6.1 Armour pupil function.

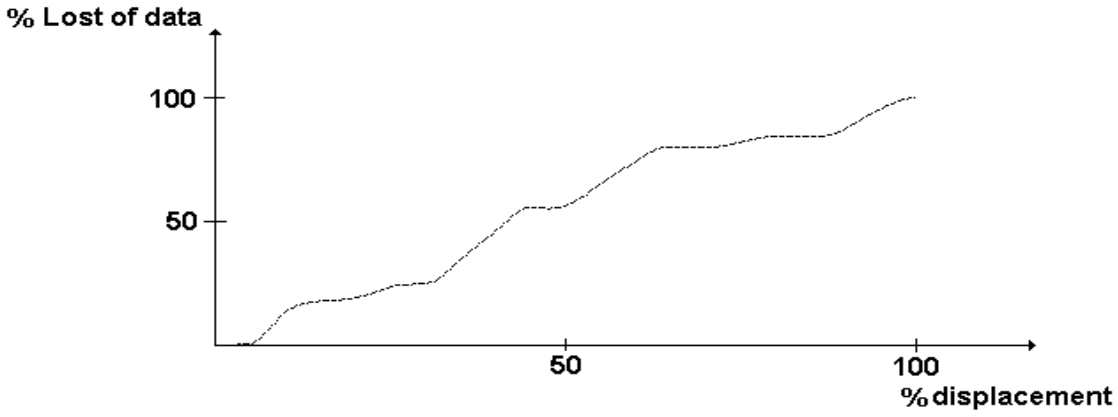


Figure 6.2 Percent of lost of data vs. lateral shear in an armour pupil.

The figure 6.3 show the lost of interfered data as function of lateral shear displacement in a circle pupil.

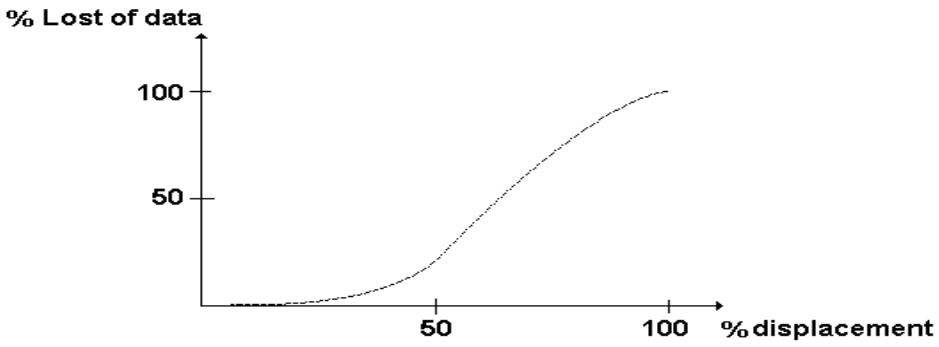


Figure 6.3 Percent of lost of data vs. lateral shear in a circle pupil.

The figure 6.4 show the lost of interfered data as function of lateral shear displacement in the Cassegrain primary mirror pupil, this pupil can be seen in the figure 5.6 of the chapter 5.

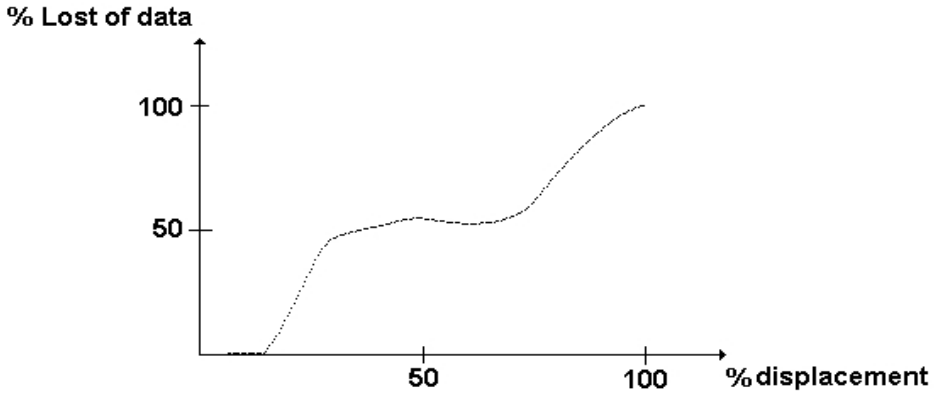


Figure 6.4 Percent of lost of data vs. lateral shear in a Cassegrain's primary mirror pupil.

The figure 6.5 show the lost of interfered data as function of lateral shear displacement in a square pupil.

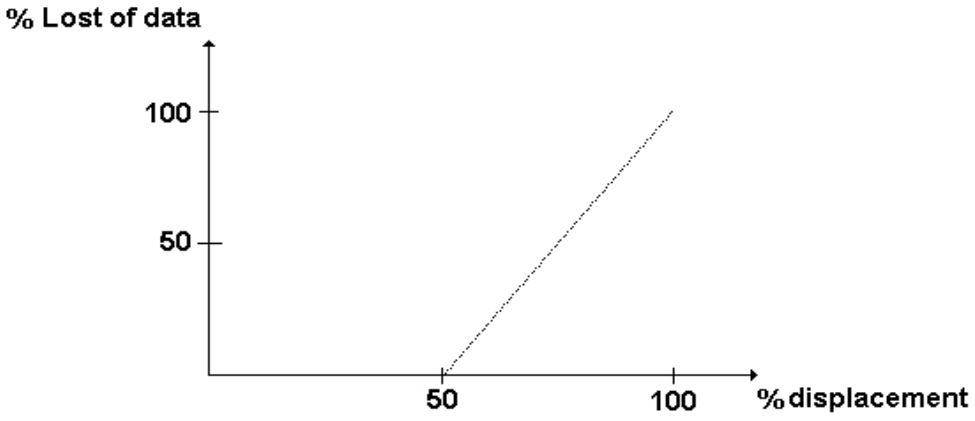


Figure Curve of lost of data vs displacement

Figure 6.5 Percent of lost of data vs. lateral shear in a square pupil.

Figure 6.6 show a numerical computation of the lost of some spectral components due to the lateral shear. It produce that the recovery of the wave-front from a lateral shear interferometer is an ill posed problem.

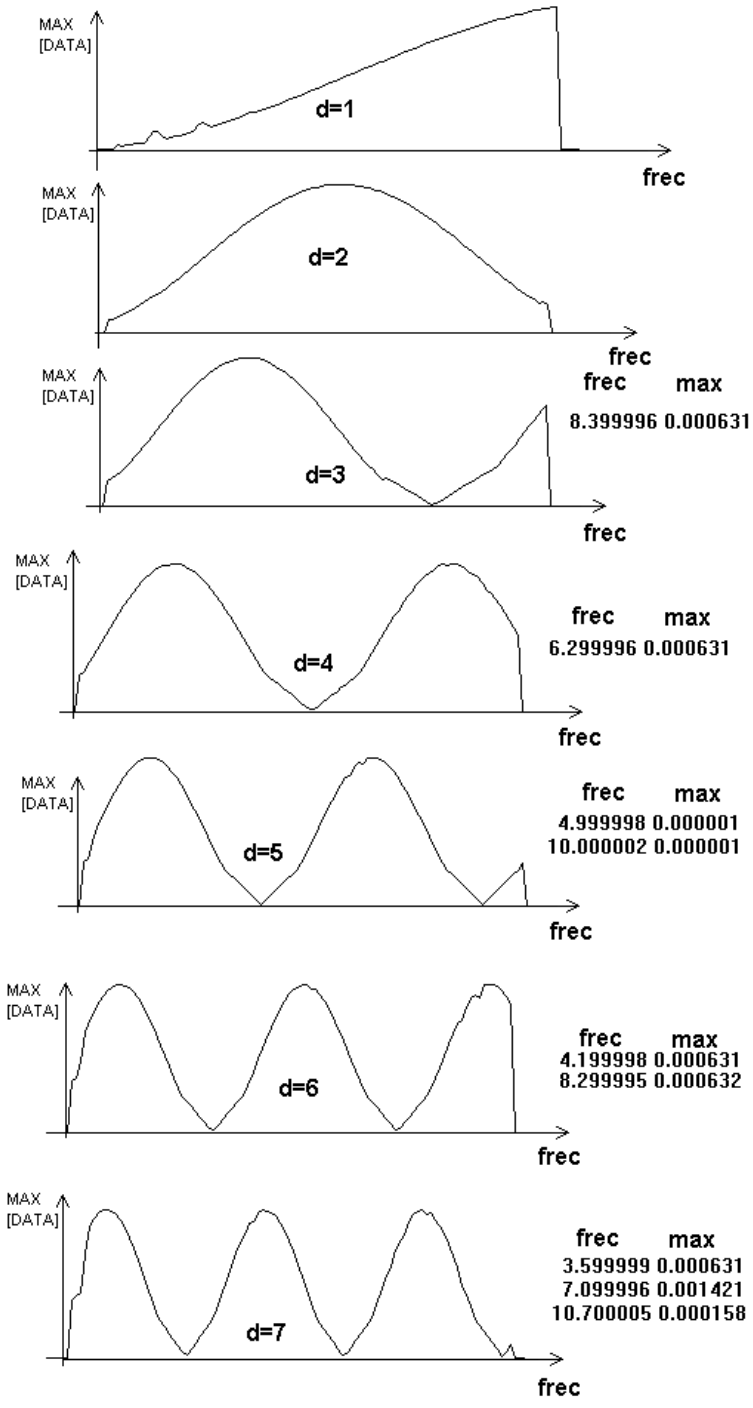


Figure 6.6 Numerical computation of frequency response of interferometer on observations.

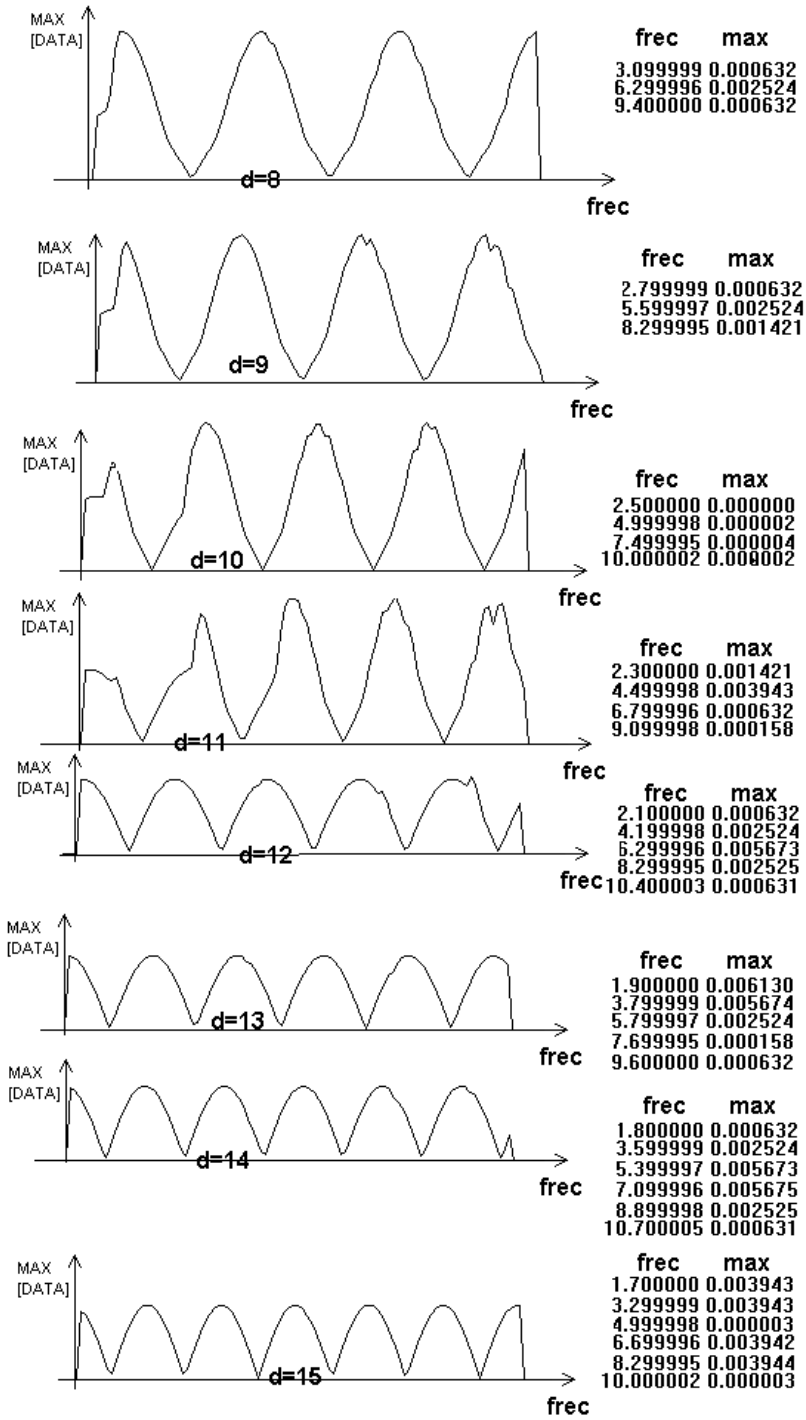


Figure 6.6 Numerical computation of frequency response of interferometer on observations (cont.).

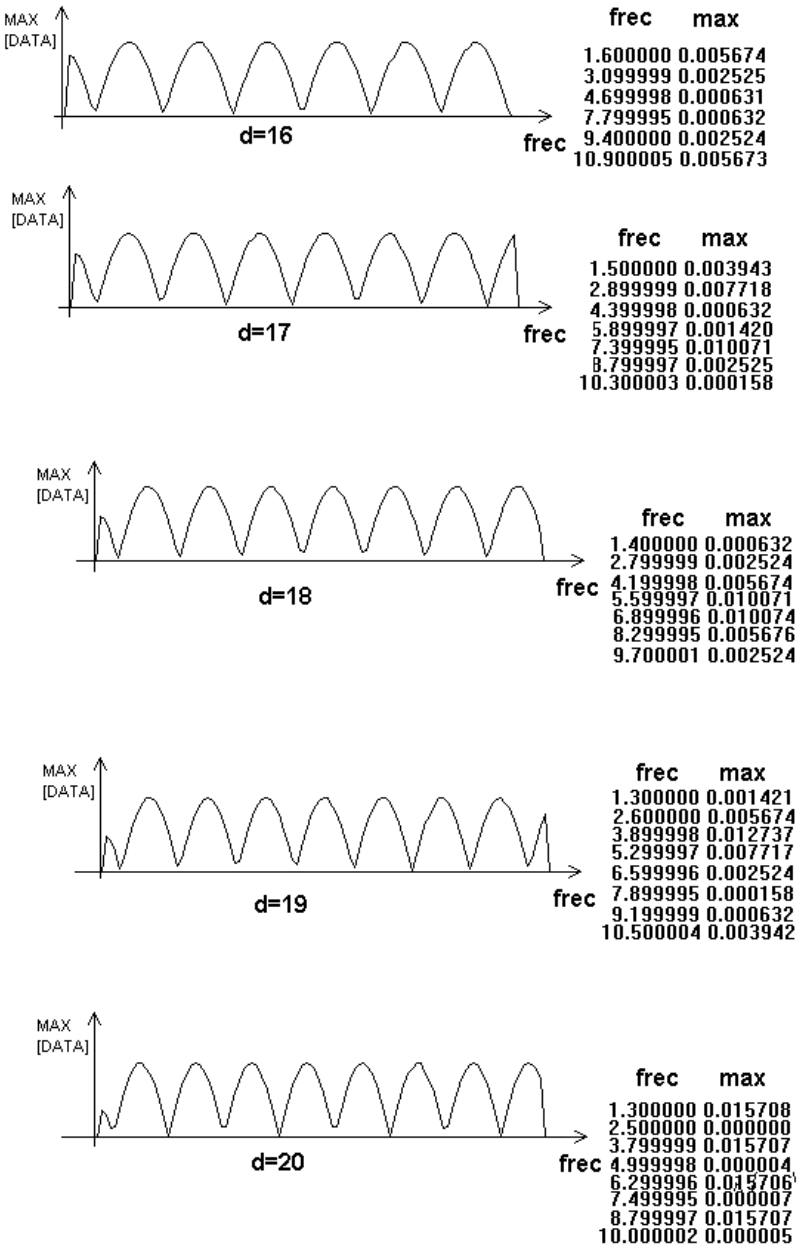


Figure 6.6 Numerical computation of frequency response of interferometer on observations (cont).

Figure 6.7 show a one-dimension wave-front recovery for plate regularizing term with lambda=Alfa=0.01 (unless another thing is indicated) and $\delta = 7$ pixels. The solution is reached by means of the conjugate gradient of the equation (5.20). It can be observe that we only need to introduce DC information in order to recover the original wave-front. The wave-front chosen is a cosine function, the idea behind this function is that the wave-front have a selective frequency f and the recovery in each case is attained.

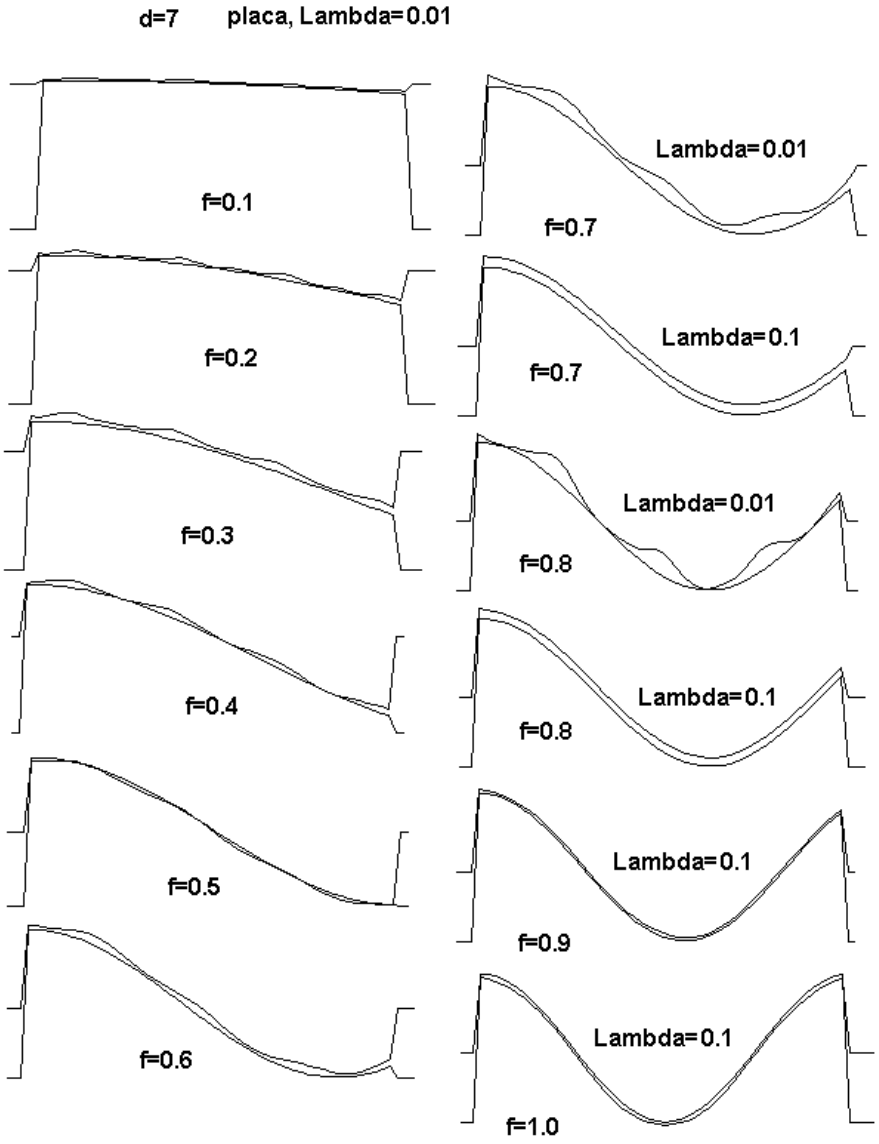


Figure 6.7 Wave-front recoveries for a lateral shear $\delta = 7$.

Figure 6.8 show a one-dimension wave-front recovery for plate regularizing term with lambda=Alfa=0.1 and $\delta = 7$ pixels. The solution is calculated by means the conjugate gradient of the equation (5.20). It can be observe that we only need to introduce DC information in order to recover the original wave-front. The wave-front chosen is a cosine function with a different frequency (*f*) and the recovery in each case is attained.

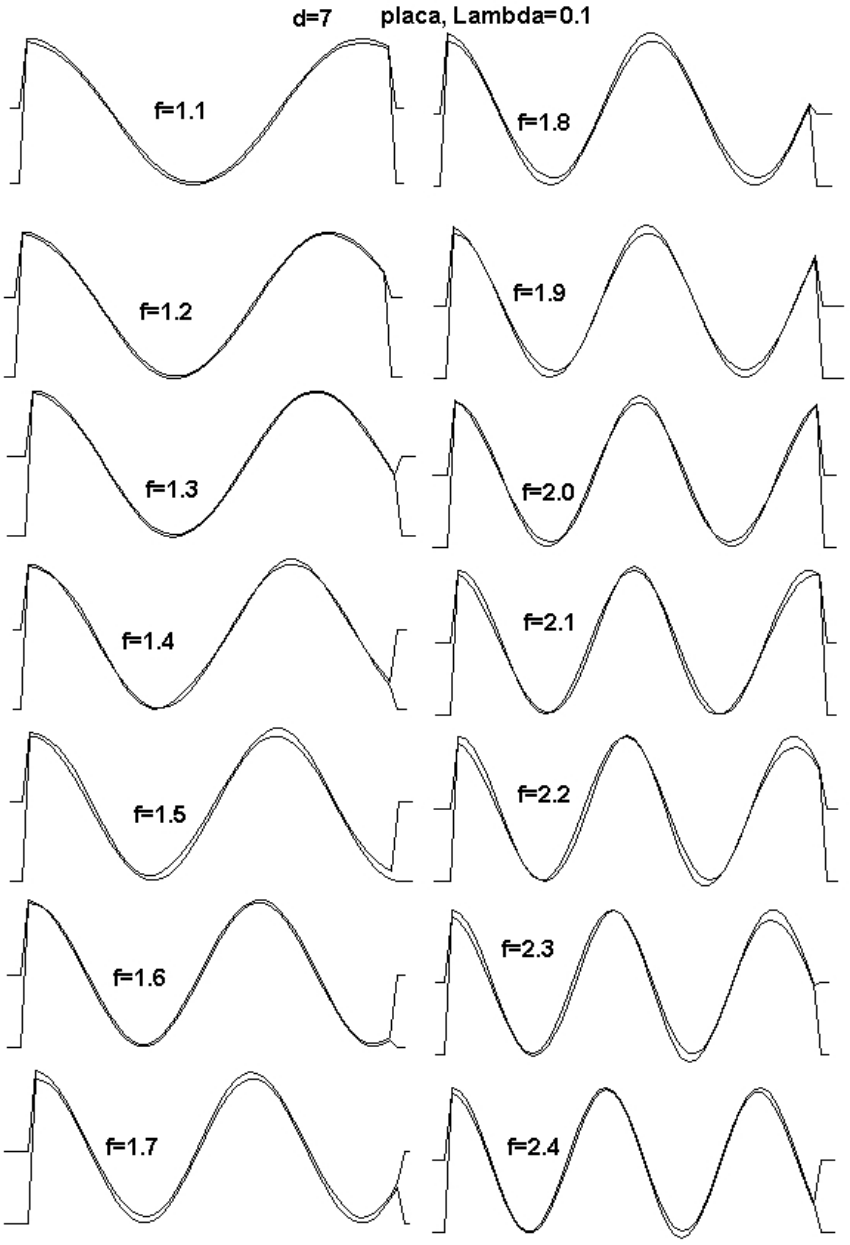


Figure 6.8 Wave-front recoveries for a lateral shear $\delta = 7$.

Figure 6.9 show a one-dimension wave-front recovery for plate regularizing term with lambda=Alfa=0.1 and different lateral shears. The solution is attained by means of computing the conjugate gradient of equation (5.20), i.e. S_i. Taking the inverse of the matrix [A] in the equation (5.20), i.e. E_i. The error between this two is calculated $E_i - S_i$. The wave-front chosen is a cosine function with a different frequency (*f*) and the recovery in each case is attained.

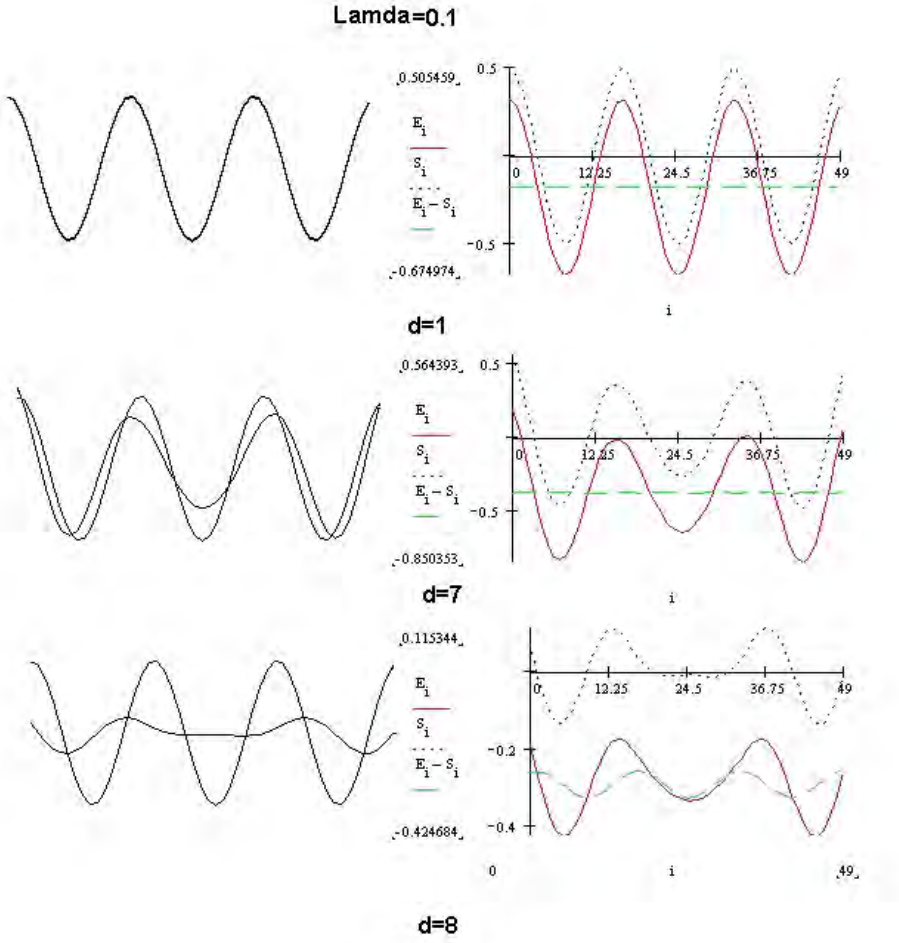


Figure 6.9 Wave-front estimation using conjugate gradient and inverse of the matrix.

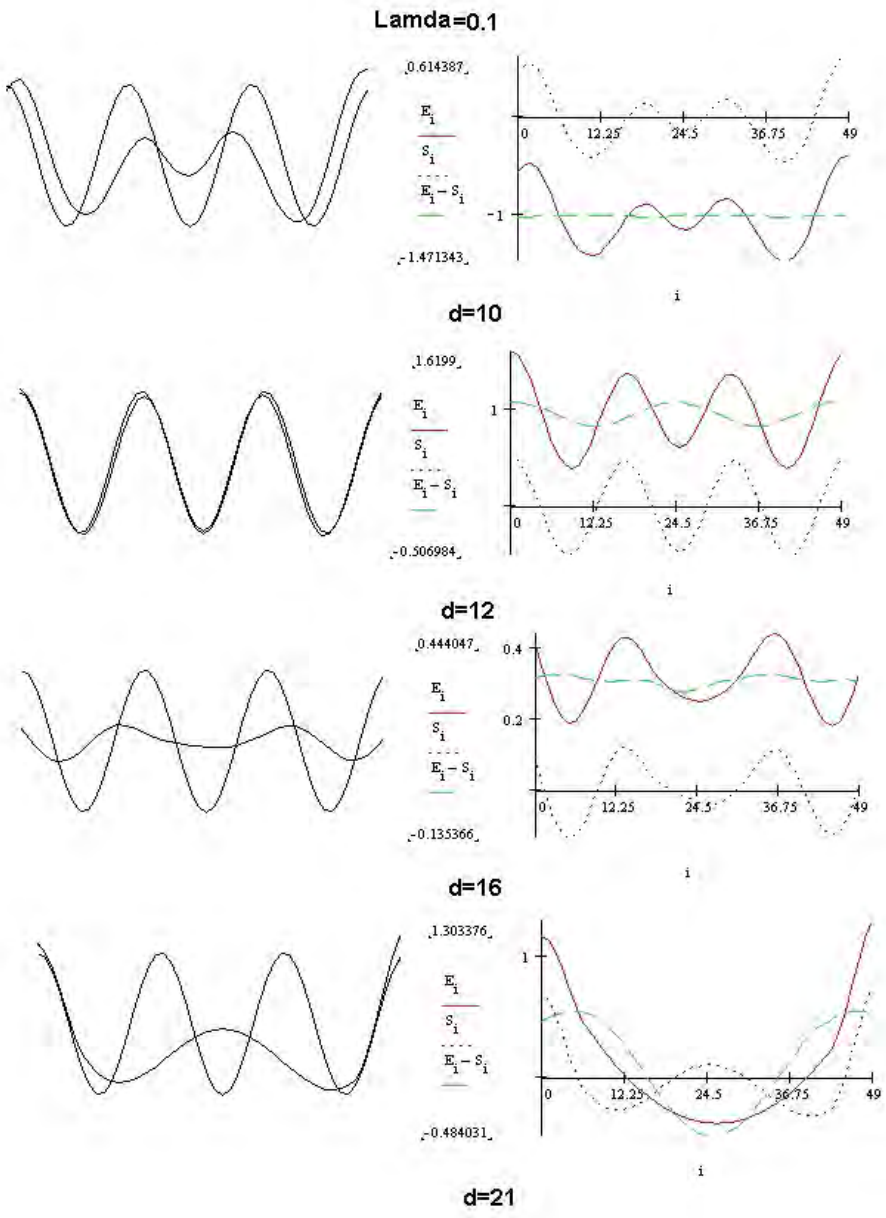


Figure 6.9 Wave-front estimation using conjugate gradient and inverse of the matrix (cont).

In figure 6.10 we can observe in the reconstruction using conjugate gradient in the solution of the equation (5.15) the effects of being close to the zeros of the frequency response of the lateral shear interferometer. Also we can observe for large lateral shear that we have lost of data in the middle of the domain function The wave-front is a cosine function with spatial frequency f=4.

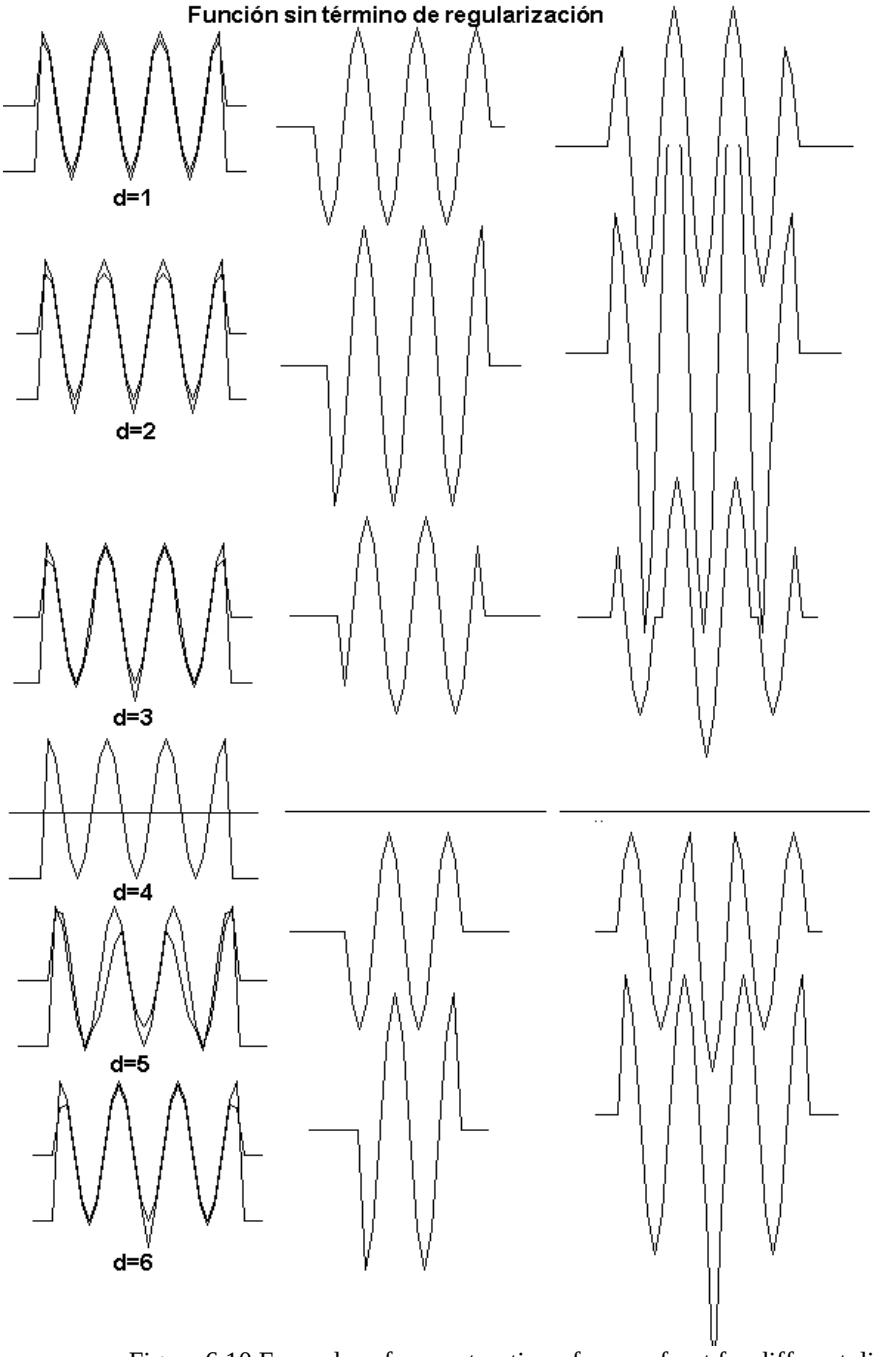


Figure 6.10 Examples of reconstruction of wave-front for different displacements.

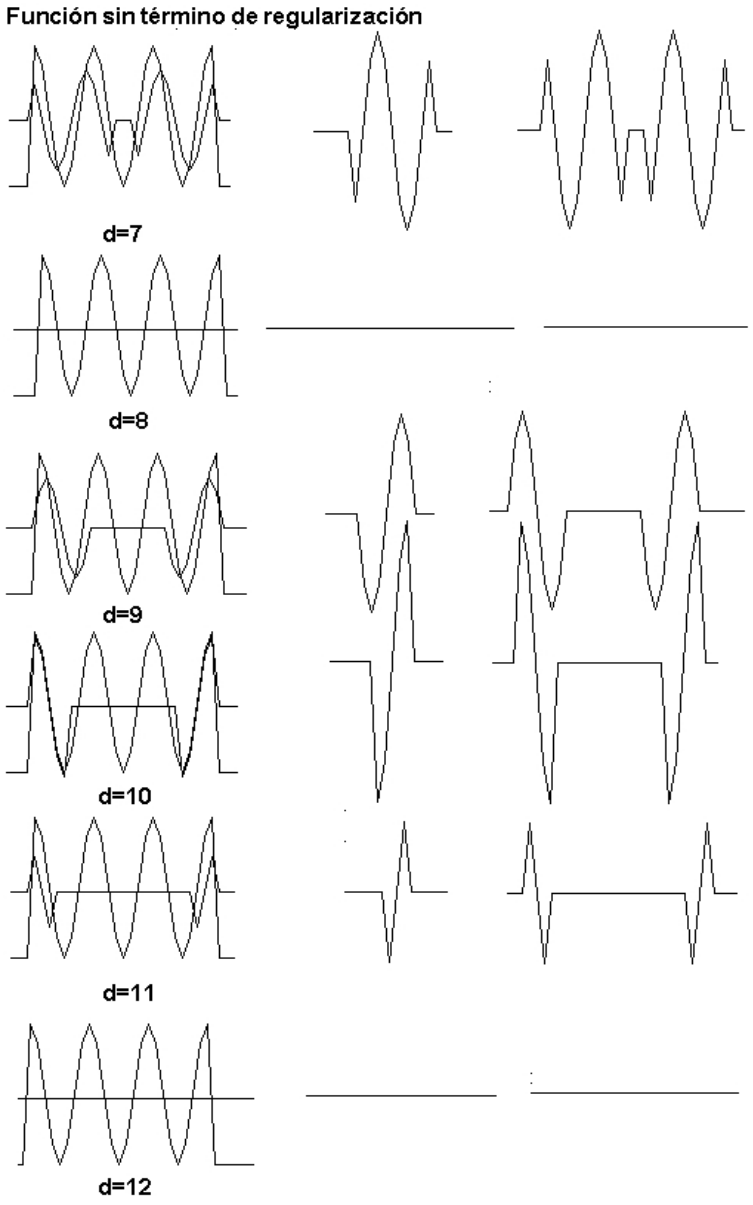


Figure 6.10 Examples of reconstruction of wave-front for different displacements (cont).

In figure 6.11 we can observe in the reconstruction using conjugate gradient in the solution of the equation (5.20) the effects of being close to the zeros of the frequency response of the lateral shear interferometer. Also we can observe for large lateral shear that we have lost of data in the middle of the domain function The wave-front is a cosine function with spatial frequency f=8.3333.

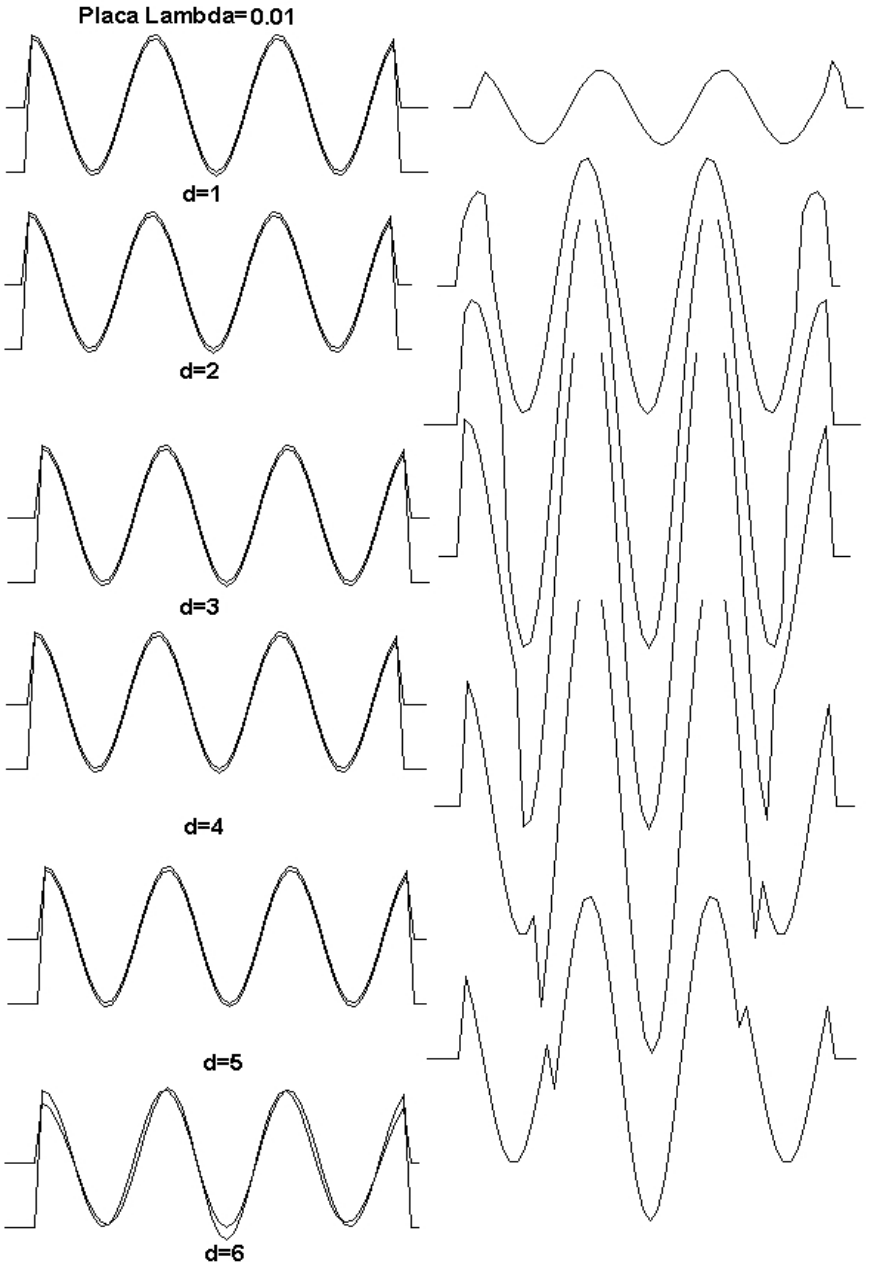


Figure 6.11 Examples of reconstruction of wave-front for different displacements.

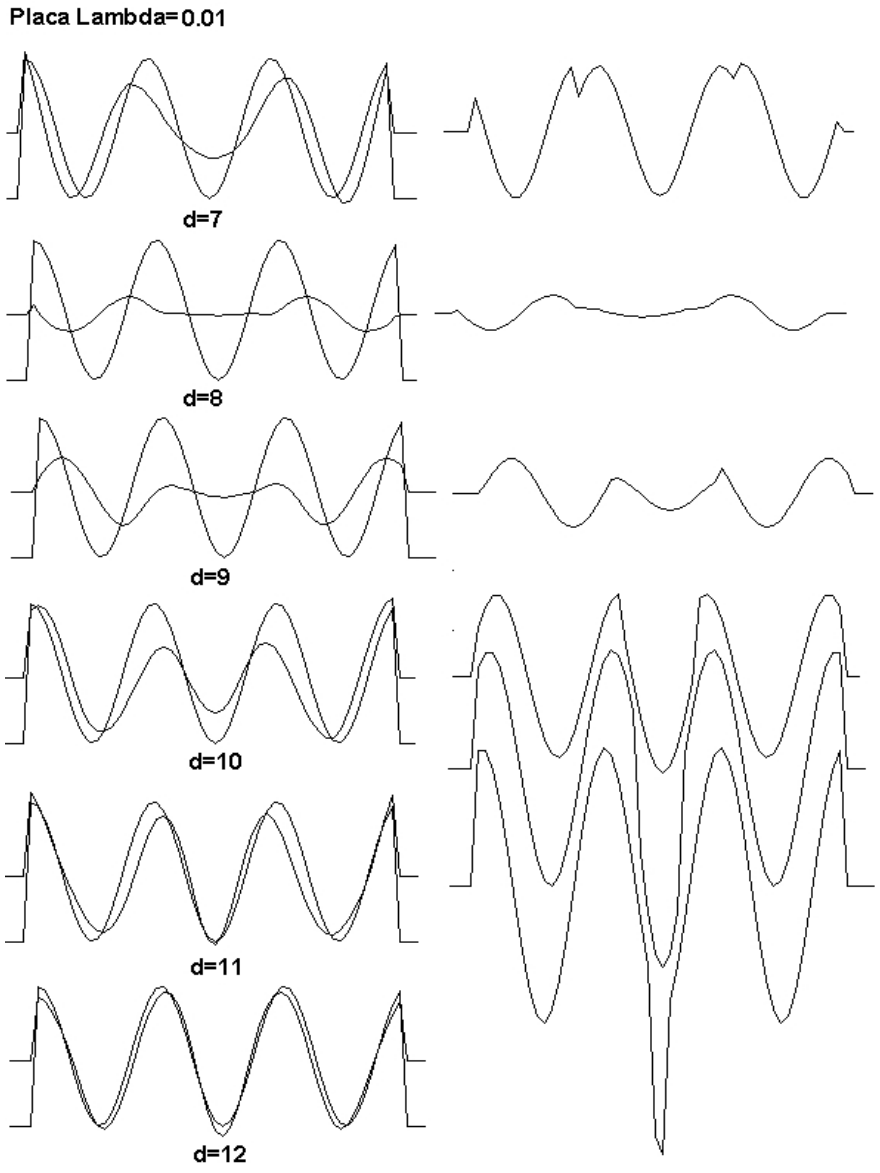


Figure 6.11 Examples of reconstruction of wave-front for different displacements (cont).

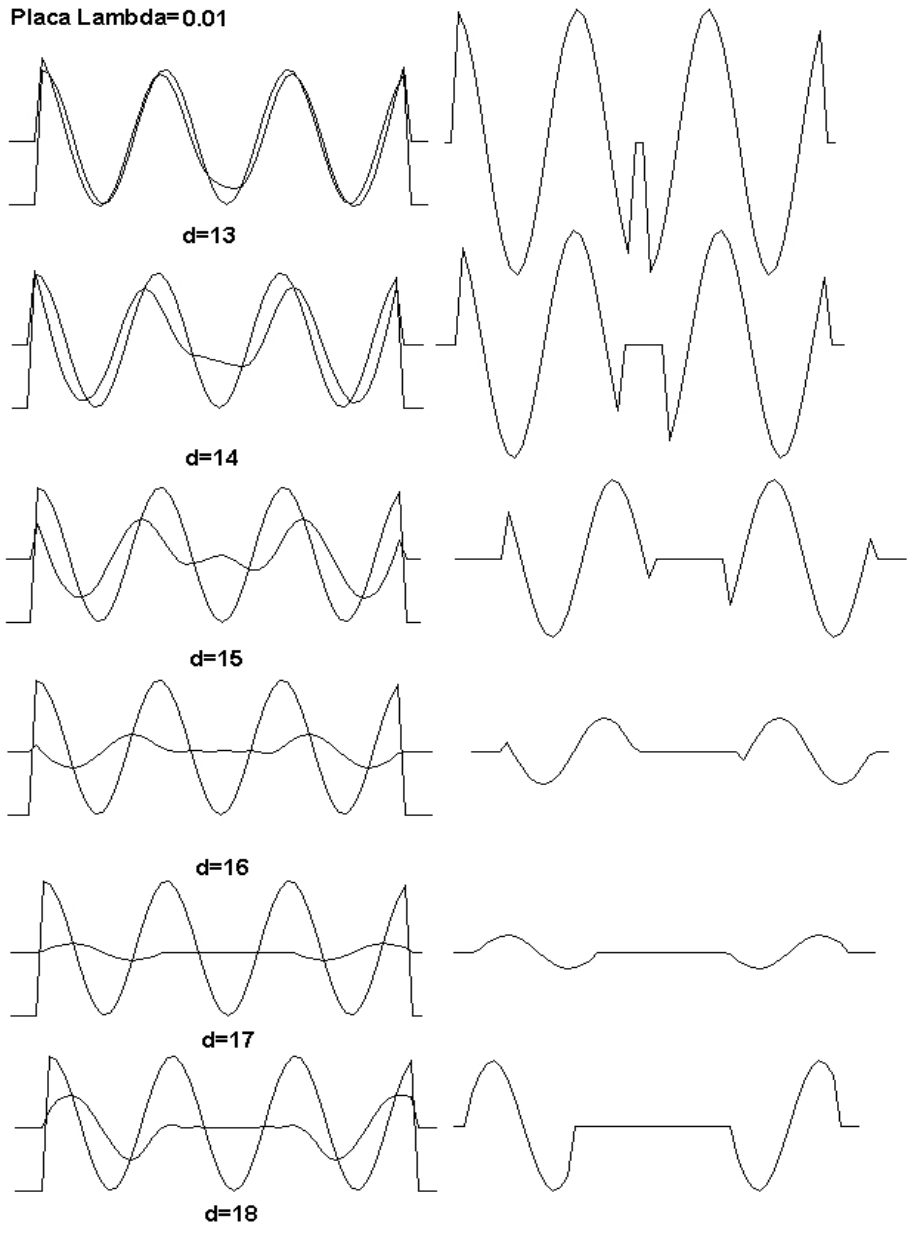


Figure 6.11 Examples of reconstruction of wave-front for different displacements (cont).

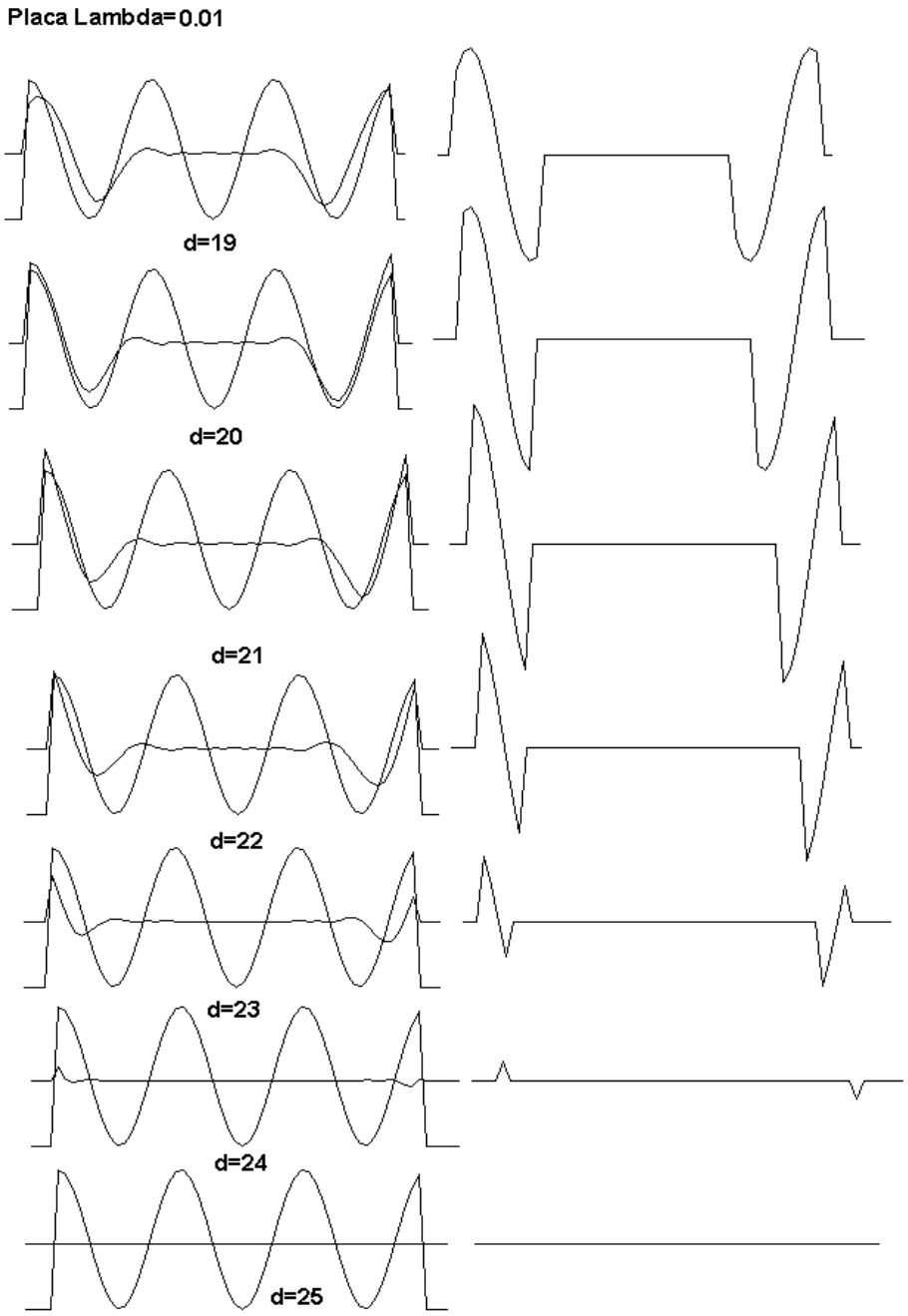


Figure 6.11 Examples of reconstruction of wave-front for different displacements (cont).

Figure 6.12 show a numerical computation of the impulse response; it can be observe that the impulse response is variable spatially due to the finite size of the pupil.

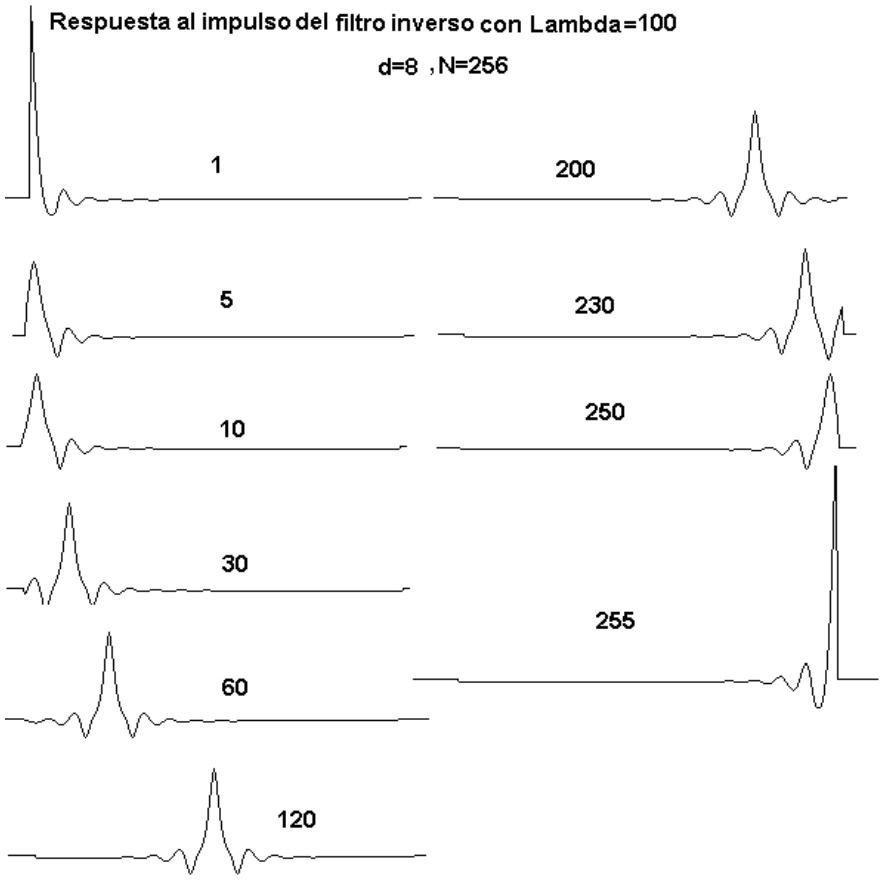


Figure 6.12 Numerical computation of impulse response of the CG applied to equation (5.20) to recover the wave-front with Alfa=lambda=100.

6.2 Conclusions to the work.

In this work we studied the lateral shear interferometer from different aspects, from it's physical construction integrating all the necessary information to make a right decision for the implementation of the interferometer based mainly in the Murtys's type interferometer. Other aspect is the sensitivity of the instrument used to optical proofs. We studied the theoretical limit in the lateral displacement, but is not always recommended use it. To get a better sensitivity of the optical proof we need to increase the lateral displacements and the third order aberrations in the LSI can't be represented as the partial derivatives as before. It is understood that the aberrations are represented by the difference of two lateral displaced wave-front. The reconstruction of the wave-front from this difference is estimated using integration of field of gradient (in the limit case) by least squares using regularization in order to solve in some soft way the lost of information due the nature of the LSI. As we can see from the graphs for numerical computation of frequency response on observations (data) the zeroes produce lost of information on specific frequencies.

In the case when the lateral shear is large, there is another source of lost of information "white space", i.e. regions in the pupil where there isn't interference of the wave-front with itself. We studied the curves of lost of "observations" useful for the reconstruction of the wave-front.

So on for large lateral shear there are two causes of lost of information one are the zeroes in the frequency response of the LSI that increase as increase the lateral shear and the other cause is due to the cut in the interference of the wave-front with itself by the pupil function that limit the extent of the interferogram (shearogram).

6.3 Work for the future.

- Another configuration of the lateral shear interferometer can be studied in detail.
- The effects of the filtering of the CG method to solve the inverse problem of reconstruction can be studied.
- Elaborate a chart of the main characteristics of the lateral shear interferometer used to optical proofs.
- Propose an alternate way to recover the wave-front of the LSI using a local conditioner for the parameter Alfa.
- Propose an alternate way to recover the wave-front of the LSI using the frequency domain.

References

Bertero M. and P. Boccacci, *Introduction to inverse problems in imaging*, Institute of physics publishing, England, 1998.

Appendix A
Introducción al procesamiento digital de señales.

A.1 Caracterización y clasificación de las señales.

El termino señal se aplica generalmente a algo que lleva información. Las señales llevan generalmente información sobre el comportamiento de un sistema físico. Aunque las señales se pueden representar de muchas formas la información está contenida en algún patrón de variaciones. Las señales se representan matemáticamente como funciones de una o más variables independientes. Además las señales pueden ser funciones de valor real o funciones de valor complejo.

A.1.1 Señales de tiempo continuo y señales de tiempo discreto.

Dependiendo de la naturaleza de las variables independientes y del valor de la función que define la señal, se pueden definir varios tipos de señales. Por ejemplo, las variables independientes pueden ser continuas o discretas. En el caso de las señales continuas la variable independiente es continua , por lo que estas señales se definen para una sucesión continua de valores de la variable independiente. Las señales en tiempo continuo se denominan frecuentemente señales analógicas. Por otra parte, las señales discretas solo están definidas en tiempos discretos, y en consecuencia para estas señales la variable independiente toma solo un conjunto discreto de valores, esto es una secuencia de números.

A.1.2 Señales de valor continuo y de valor discreto.

Los valores de una señal de tiempo continuo o de tiempo discreto pueden ser continuos o discretos. Si una señal toma todos los valores posibles sobre un rango finito o sobre un rango infinito, ésta se dice que es una señal de valor continuo. Alternativamente si la señal toma valores de un conjunto finito de posibles valores, se dice que es una señal de valor discreto. Usualmente, estos valores son equidistantes y de aquí que puedan ser expresados como un múltiplo entero de la distancia entre dos valores sucesivos. Una señal de tiempo discreto que tiene un conjunto de valores discretos es llamada una señal digital.

A.1.3 Señales determinísticas y señales aleatorias.

El análisis matemático y el procesamiento de señales requiere la posibilidad de una descripción matemática de la señal en si misma. Esta descripción matemática, se conoce como el modelo de señal y conduce a otra importante clasificación de las señales. Cualquier señal que puede ser descrita únicamente por una expresión matemática explicita, una tabla de datos, o una regla bien definida es llamada determinística. Este término se usa para enfatizar el hecho de que todos los valores pasados, presentes o futuros de la señal son conocidos precisamente, sin incertidumbre.

En muchas aplicaciones prácticas de cualquier forma, hay señales que no pueden ser descritas con un grado razonable de exactitud por formulas matemáticas explicitas, o tales descripciones son demasiado complicadas para ser utilizadas en la práctica. La falta de tales relaciones implican que tales señales evolucionan en el tiempo de una manera impredecible. Nos referiremos a estas señales como aleatorias.

A.2 Conversión de señales analógicas a señales digitales.

La mayoría de las señales de interés práctico como velocidad, señales biológicas, sísmicas, de radar, de sonar, y señales de comunicaciones como audio y video son señales analógicas. Para procesar señales analógicas por medios digitales, es necesario convertirlas en una forma digital, es decir convertirlas en una secuencia de números con una precisión finita. Este procedimiento se llama conversión analógica-digital (A/D), y los dispositivos correspondientes se denominan convertidores A/D.

Conceptualmente la conversión analógica-digital (A/D) es un proceso de tres pasos:

1. Muestreo. Es la conversión de una señal de tiempo continuo en una señal de tiempo discreto y se obtiene por medio de muestras de la señal continua a instantes de tiempo discreto. Por lo tanto, si $x_a(t)$ es la entrada a un muestreador, la salida es $x_a(nT) \equiv x[n]$ donde T es llamado intervalo de muestreo y n es una variable entera.
2. Cuantización. Es la conversión de una señal de tiempo discreto y de valor continuo en una señal de tiempo discreto y de valor discreto, señal digital. El valor de cada muestra de la señal esta representado por un valor seleccionado de un conjunto finito de posibles valores. La diferencia entre la muestra sin cuantizar $x[n]$ y la salida cuantizada $x_q[n]$ es llamado error de cuantización.
3. Codificación. En el proceso de codificación, cada valor discreto $x_q[n]$ es representado por una secuencia binaria de bits.

A.2.1 Muestreo de señales analógicas.

Existen muchas formas de muestrear una señal analógica. El tipo de muestreo más utilizado en la práctica es el muestreo periódico o uniforme. Se describe por la siguiente relación

$$\boxed{x[n] = x_a(t)\big|_{t=nT} = x_a(nT), \qquad n = ...,-2,-1,0,1,2,...} \tag{A.1}$$

donde $x[n]$ es la señal de tiempo discreto, que se obtiene "tomando muestras" de la señal analógica $x_a(t)$ cada T segundos. El intervalo de tiempo T entre muestras sucesivas se

denomina período de muestreo o intervalo de muestreo y su reciproco $1/T = f_s$ es denominado razón de muestreo (muestras por segundo) o frecuencia de muestreo (hertz).

El muestreo periódico establece una relación entre las variables de tiempo t y n para las señales de tiempo continuo y de tiempo discreto, respectivamente. Estas variables están linealmente relacionadas a través del periodo de muestreo T o, equivalentemente, a través de la frecuencia de muestreo, $f_s = 1/T$ como

$$\boxed{t = nT = \frac{n}{f_s}} \tag{A.2}$$

A.2.1.1 Teorema de muestreo

Dada una señal analógica, ¿Cómo puedo seleccionar el periodo de muestreo T, o equivalentemente la razón de muestreo f_s? Para contestar esta pregunta debemos tener cierta información sobre las características de la señal que será muestreada. En particular, debemos tener alguna información sobre el contenido frecuencial de la señal.

Del conocimiento del contenido frecuencial de la señal, en particular de $f_{mseñal}$ (frecuencia máxima de la señal), podemos seleccionar una frecuencia de muestreo apropiada. Sabemos que la más alta frecuencia de una señal analógica que puede ser reconstruida sin ambigüedad es cuando la señal muestreada a una razón $f_s = \frac{1}{T}$ es $f_s/2$. Cualquier frecuencia arriba de $f_s/2$ o debajo de $-f_s/2$ resulta en muestras que son idénticas con una frecuencia correspondiente en el rango $-f_s/2 \le f \le f_s/2$. Para evitar las ambigüedades que resultan del aliasing (traslape de componentes de frecuencia), debemos seleccionar la razón de muestreo suficientemente grande. Esto es, debemos seleccionar $f_s/2$ mayor de $f_{mseñal}$. Por lo tanto, para evitar el problema de aliasing f_s se selecciona de tal manera que

$$\boxed{f_s > 2 f_{mseñal}} \tag{A.3}$$

donde $f_{mseñal}$ es la componente de mayor frecuencia en la señal analógica. Con la razón de muestreo seleccionada de esta manera, cualquier componente en frecuencia en la señal analógica, f, que sea menor que la frecuencia máxima de la señal, $|f| < f_{mseñal}$, es mapeada en una senoide de tiempo discreto con una frecuencia

$$\boxed{-\frac{1}{2} \le f_{Norm} = \frac{f}{f_s} \le \frac{1}{2}} \tag{A.4}$$

o equivalentemente,

$$\boxed{-\pi \le \Omega = 2\pi f_{Norm} \le \pi} \qquad \text{(A.5)}$$

donde Ω es la frecuencia angular normalizada. Al intervalo de frecuencias normalizadas definido por (4) o bien al intervalo de frecuencias angulares normalizadas definidas por (5) se les conoce como intervalo fundamental.

Ya que $\left| f_{Norm} \right| = \frac{1}{2}$ ó $\left|\Omega\right| = \pi$ es la única frecuencia más alta en una señal de tiempo discreto, la elección de una frecuencia de muestreo de acuerdo con la equación (3) evita el problema de aliasing. En otras palabras, la condición $f_s > 2 f_{mseñal}$, asegura que todas las componentes sinusoidales en la señal analógica son mapeadas en sus correspondientes componentes en frecuencia de tiempo discreto con frecuencias en el intervalo fundamental. Así todas las componentes en frecuencia de la señal analógica son representadas en su forma muestreada sin ambigüedad, y de aquí que la señal analógica pueda ser reconstruida sin distorsión de los valores muestreados usando un método de interpolación "apropiado" (conversión digital a analógico). La fórmula de interpolación "apropiada" o ideal es especificada por el teorema del muestreo.

A.2.2 Cuantización de señales de amplitud continua.

Una señal digital es una secuencia de números (muestras) en la cual cada número está representado por un número finito de dígitos (precisión finita).

El proceso de convertir una señal de tiempo discreto y de valor continuo en una señal digital expresando cada valor de la muestra como un número finito (en lugar de infinito) de digitos, es llamado cuantización. El error introducido en la representación de señales de valor continuo por un conjunto finito de niveles de valor discreto es llamado error de cuantización o ruido de cuantización.

Se denota la operación de cuantización de las muestras $x[n]$ como $Q\{x[n]\}$ y sea $x_q[n]$ que denota la secuencia de muestras cuantizadas a la salida del cuantizador. Por lo tanto

$$x_q[n] = Q\{x[n]\}$$

Entonces el error de cuantización es una secuencia $e_q[n]$ definido como la diferencia entre el valor cuantizado y el valor verdadero de la muestra. Así

$$\boxed{e_q[n] = x_q[n] - x[n]} \qquad \text{(A.6)}$$

A.2.3 Codificación de muestras cuantizadas.

El proceso de codificación en un convertidor A/D asigna un número binario único a cada nivel de cuantización. Si se tienen L niveles se necesitan al menos L diferentes números binarios. Con una longitud de palabra de b bits pueden ser representados 2^b diferentes números binarios. De aquí que $2^b \geq L$, o equivalentemente, $b \geq \log_2 L$. Así el número de bits requeridos en el codificador es el entero más pequeño mayor que o igual a $\log_2 L$.

A.3 Ventajas de procesamiento de señales digitales sobre señales analógicas.

Hay muchas razones del porque el procesamiento digital de señales de una señal analógica puede ser preferible al procesamiento de la señal directamente en el dominio analógico. Primero, un sistema digital programable tiene flexibilidad para reconfigurar las operaciones del procesamiento digital de la señal simplemente cambiando el programa. La reconfiguración de un sistema analógico usualmente implica un rediseño del hardware seguido por pruebas y verificaciones para ver que este opere apropiadamente.

Las consideraciones de precisión también juegan un papel importante en la determinación de la forma del procesador de señales. Las tolerancias en los componentes de circuitos analógicos hacen extremadamente difícil para el diseñador del sistema controlar la precisión de un sistema de procesamiento de señales analógico. Por otro lado, un sistema digital provee un mejor control de los requerimientos de precisión. Tales requerimientos, a su vez, resultan en la especificación de los requerimientos de precisión en el convertidor A/D y el procesador de señales digitales, en términos de longitud de palabra, aritmética de punto flotante o aritmética de punto fijo, y factores similares.

Las señales digitales son fácilmente almacenadas en medios magnéticos sin deterioro o pérdida en la fidelidad de la señal más allá que el introducido en la conversión A/D. Como consecuencia, las señales pueden transportarse y pueden ser transportadas “fuera de línea” en un laboratorio remoto. El método de procesamiento digital de señales también permite la implementación de algoritmos más sofisticados de procesamiento de señales. Usualmente es mas difícil realizar operaciones matemáticas precisas sobre señales analógicas pero estas mismas operaciones pueden ser implementadas sobre una computadora digital usando software.

En algunos casos una implementación digital de un sistema de procesamiento de señales es más barato que su contraparte analógica.

Sin embargo las implementaciones digitales tienen sus limitaciones. Una limitación práctica es la velocidad de operación de los convertidores A/D y los procesadores de señales digitales.

A.4 Conversión digital a Analógico.

Para convertir una señal digital a una señal analógica puede ser utilizado un convertidor digital a analógico (D/A). La tarea de un convertidor D/A es interpolar entre muestras. Todos los convertidores D/A "conectan los puntos" en la señal digital realizando algún tipo de interpolación, cuya precisión dependen de la calidad del proceso de conversión D/A.

El más simple convertidor D/A es un mantenedor de orden cero o aproximación escalera, la cual simplemente mantiene constante el valor de una muestra hasta que se recibe la próxima muestra. Se pueden obtener mejoras adicionales usando interpolación lineal que conecta muestras sucesivas con segmentos de línea recta. Aún mejores interpolaciones pueden ser logradas utilizando técnicas de interpolación de alto orden más sofisticadas.

Referencias bibliográficas.

Mitra, Sanjit K., *Digital Signal Processing*, a computer-based Approach, Mc Graw Hill, New York, 2001.
Oppenheim, Alan V., Schafer, Ronald W., *Tratamiento de señales en tiempo discreto*, Prentice Hall, España, 1999.
Proakis, John G., Manolakis, Dimitris G., *Digital Signal Processing, principles, algorithms, and applications*, Prentice Hall, USA, 1996.

Appendix A

A Introducción al procesamiento digital de señales.
- **A.1** Caracterización y clasificación de las señales.
  - **A.1.1** Señales de tiempo continuo y señales de tiempo discreto.
  - **A.1.2** Señales de valor continuo y de valor discreto.
  - **A.1.3** Señales determinísticas y señales aleatorias.
- **A.2** Conversión de señales analógicas a señales digitales.
  - **A.2.1** Muestreo de señales analógicas.
    - **A.2.1.1** Teorema de muestreo.
  - **A.2.2** Cuantización de señales de amplitud continua.
  - **A.2.3** Codificación de muestras cuantizadas.
- **A.3** Ventajas de procesamiento de señales digitales sobre señales analógicas.
- **A.4** Conversión digital a Analógico.

Referencias bibliográficas.

Appendix B.1
Relaciones entre serie y transformada de Fourier en dominio (tiempo) continuo y discreto

B.1.1 Serie de Fourier de señales periódicas de tiempo continuo.

Para cualquier señal periódica que satisfaga las condiciones de Dirichlet, puede calcularsele los coeficientes de la serie de Fourier.

$$\boxed{c_k = \frac{1}{T_p}\int_{-T_p/2}^{T_p/2} x(t)e^{-jk\omega_o t}dt} \qquad \text{(B1.1)}$$

T_p es el periodo de la señal.

$$T_p = \frac{2\pi}{\omega_o} \qquad \text{(B1.2)}$$

Calculando su inversa:

$$\boxed{x_p(t) = \sum_{k=-\infty}^{\infty} c_k e^{jk\omega_o t}} \qquad \text{(B1.3)}$$

de (B1.2)

$$\frac{1}{T_p} = \frac{\omega_o}{2\pi}$$

B.1.2 Transformada de Fourier de señales no periódicas de tiempo continuo.

$$lim_{T_p\to\infty}\frac{1}{T_p} \to 0 \qquad \therefore \qquad \omega_o \to d\omega$$

entonces

$$lim_{T_p\to\infty}\frac{1}{T_p} = \frac{d\omega}{2\pi} \qquad \text{(B1.4)}$$

por tanto, sustituyendo el límite $T_p \to \infty$ y sustituyendo la ecuación (B1.4) en (B1.1)

$$C(\omega) = \frac{d\omega}{2\pi}\int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt \qquad \text{(B1.5)}$$

reescribiendo (B1.5)

$$2\pi\frac{C(\omega)}{d\omega} = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt \qquad \text{(B1.6)}$$

definiendo

$$X(\omega) = 2\pi\frac{C(\omega)}{d\omega} \qquad \text{(B1.7)}$$

$$\boxed{X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt} \qquad \text{(B1.8)}$$

sustituyendo (B1.7) en (B1.3), aplicando el límite $T_p \to \infty$ en (B1.3) y como $k\omega_o \to \omega$

$$x(t) = \int_{-\infty}^{\infty} \frac{X(\omega)}{2\pi}e^{j\omega t}d\omega \qquad \text{(B1.9)}$$

$$\boxed{x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)e^{j\omega t}d\omega} \qquad \text{(B1.10)}$$

B.1.3 Serie de Fourier de señales periódicas de tiempo discreto.

Con las *N* muestras tomadas en un periodo y de (B1.1), se obtiene el cálculo de los coeficientes de la serie de Fourier para señales periódicas de tiempo discreto. Nótese que las muestras están centradas respecto al origen (no es forzosa esta condición). Haciendo $t = nT_s$ donde T_s es el periodo de muestreo, esto es, $x[n]$ es periódica con periodo fundamental *N*.

Si N - impar.	Si N - par.
$c_k = \frac{1}{N}\sum_{n=-N/2}^{N/2} x(nT_s)e^{-jk\omega_o nT_s}$	$c_k = \frac{1}{N}\sum_{n=-N/2}^{\frac{N}{2}-1} x(nT_s)e^{-jk\omega_o nT_s}$

(B1.11a)

donde

N es un entero.

$$\omega_o = \frac{2\pi}{N}$$

$$N = \frac{2\pi}{\omega_o}$$

Nota: Obsérvese que debido a que N es un número entero, la razón $\frac{2\pi}{\omega_o}$ debe ser un número racional $\frac{p}{q}$, con p y q siendo números enteros también. Como N es el periodo fundamental, se simplifican los factores comunes entre p y q.

Equivalentemente para una señal de tiempo discreta $x[n]$ (i.e., que puede ser expresada como una secuencia), las relaciones en (B1.11a) pueden ser re-expresadas como:

Si N - impar.	Si N - par.
$c_k = \frac{1}{N}\sum_{n=-N/2}^{N/2} x[n]e^{-jkn\Omega_o}$	$c_k = \frac{1}{N}\sum_{n=-N/2}^{\frac{N}{2}-1} x[n]e^{-jkn\Omega_o}$

(B1.11b)

donde

N es un entero y es el periodo fundamental de la señal de tiempo dicreto.

$N = \frac{2\pi}{\Omega_o}$.

Relacionamos la frecuencia de la señal de tiempo continuo ($f_{señal\ periodica\ de\ tiempo\ continuo}$) con la frecuencia de muestreo (f_s) por medio de la frecuencia normalizada (F) o equivalentemente, mediante un factor de escala de 2π, a la frecuencia angular normalizada (Ω_o) con la señal de tiempo discreto $x[n]$.

$$F = \frac{f_{señal\ periodica\ de\ tiempo\ continuo}}{f_s}$$

f_s es la frecuencia de muestreo.

$\Omega_o = 2\pi F$.

De (B1.3) haciendo $t = nT_s$ donde T_s es el periodo de muestreo.

$$x(nT_s) = \sum_{k=-\infty}^{\infty} c_k e^{jk\omega_o nT_s} \qquad \text{(B1.12)}$$

Determinemos el caso cuando dos muestras consecutivas tienen la misma fase de la señal periódica.

$$k\omega_o T_s(n+1) - k\omega_o T_s n = 2\pi m \qquad m = 0,\pm1,\pm2,...$$

$$k\omega_o T_s n + k\omega_o T_s - k\omega_o T_s n = 2\pi m \qquad m = 0,\pm1,\pm2,...$$

$$k\omega_o T_s = 0,\pm2\pi,\pm4\pi,... \tag{B1.13}$$

$$f_s = \frac{1}{T_s} \qquad \text{y} \qquad f_s = \frac{\omega_s}{2\pi}$$

$$T_s = \frac{2\pi}{\omega_s} \tag{B1.14}$$

sustituyendo (B1.14) en (B1.13)

$$k\omega_o \frac{2\pi}{\omega_s} = 0,\pm2\pi,\pm4\pi,...$$

dividiendo ambos lados de la ecuación anterior entre 2π

$$k\frac{\omega_o}{\omega_s} = 0,\pm1,\pm2,...$$

con $k = 0,\pm1,\pm2,...$

de lo cual se observa que

$$\frac{\omega_o}{\omega_s} = 1$$
$$\omega_o = \omega_s$$
$$\frac{2\pi}{T_o} = \frac{2\pi}{T_s}$$
$$T_o = T_s \tag{B1.14a}$$

$\therefore$ La *respuesta en frecuencia* de una señal discreta tiene un periodo T_s, como $T_s = \frac{1}{f_s}$:

$$-2\pi\left(\frac{f_s}{2}\right) \le k\omega_o \le 2\pi\left(\frac{f_s}{2}\right)$$

$$-\pi f_s \le k(2\pi f_o) \le \pi f_s$$
$$-\frac{f_s}{2} \le k f_o \le \frac{f_s}{2} \qquad \text{con } k = 0,\pm1,\pm2,...$$

$$\therefore \quad k = 0,\pm1,\pm2,...,\pm\frac{N}{2} \text{ mientras se cumpla que } \left|kf_o\right| \le \frac{f_s}{2},$$

de (B1.12) y la periodicidad mostrada en (B1.14a),

Si N - impar.	Si N - par.
$x(nT_s) = \sum_{n=-N/2}^{N/2} c_k e^{jk\omega_o nT_s}$	$x(nT_s) = \sum_{n=-N/2}^{\frac{N}{2}-1} c_k e^{jk\omega_o nT_s}$

(B1.15a)

donde, N es un entero y el número total de componentes frecuenciales no repetidos $\left(\left|k\right| \le \frac{N}{2}\right)$. Debido a que la serie de Fourier esta expresada por medio de exponenciales complejas, se puede observar la simetría de k con respecto al origen.

Equivalentemente para la señal de tiempo discreta $x[n]$ (i.e., que puede ser expresada como una secuencia), las relaciones en (B1.15a) pueden ser re-expresadas como:

Si N - impar.	Si N - par.
$x[n] = \sum_{n=-N/2}^{N/2} c_k e^{jkn\Omega_o}$	$x[n] = \sum_{n=-N/2}^{\frac{N}{2}-1} c_k e^{jkn\Omega_o}$

(B1.15b)

B.1.4 Transformada de Fourier de señales no periódicas de tiempo discreto.

Haciendo $t = nT_s$ en (B1.8)

$$X(\omega) = \int_{-\infty}^{\infty} x(nT_s) e^{-j\omega nT_s} dt$$

o bien, $x_s(t) = \sum_{n=-\infty}^{\infty} x(t)\delta(t - nT_s)$

$$X(\omega) = \int_{-\infty}^{\infty} x_s(t) e^{-j\omega t} dt$$

$$= \int_{-\infty}^{\infty} \sum_{n=-\infty}^{\infty} x(t)\delta(t - nT_s) e^{-j\omega t} dt$$

$$X(\omega) = \sum_{n=-\infty}^{\infty} \int_{-\infty}^{\infty} x(t)\delta(t - nT_s) e^{-j\omega t} dt$$

$$= \sum_{n=-\infty}^{\infty} x(nT_s) e^{-j\omega nT_s}$$

re-escribiendo:

$$\boxed{X(\omega)=\sum_{n=-\infty}^{\infty}x(nT_s)e^{-j\omega T_s n}}$$

Alternativamente, (B1.16)

$$\boxed{X(\Omega)=\sum_{n=-\infty}^{\infty}x[n]e^{-j\Omega n}}$$

Similarmente, haciendo $t=nT_s$ en (B1.10)

$$x(nT_s)=\frac{1}{2\pi}\int_{-\infty}^{\infty}X(\omega)e^{j\omega nT_s}d\omega$$

reescribiendo

$$x(nT_s)=\frac{1}{2\pi}\int_{-\infty}^{\infty}X(\omega)e^{j\omega T_s n}d\omega \qquad \text{(B1.16a)}$$

Debido a que el espectro en frecuencia $X(\omega)$ contiene toda la información de $x(nT_s)$ en el intervalo[1]:

$$-2\pi\frac{f_s}{2}\le\omega\le 2\pi\frac{f_s}{2}$$

$$-\pi f_s\le\omega\le\pi f_s$$

$$-\frac{\pi}{T_s}\le\omega\le\frac{\pi}{T_s}$$

$$x(nT_s)=\frac{1}{2\pi}\int_{-\pi/T_s}^{\pi/T_s}X(\omega)e^{j\omega T_s n}d\omega$$

con $\Omega=2\pi\dfrac{f}{f_s}$

$$d\Omega=2\pi\frac{df}{f_s}$$

o bien, $x_s(t)=\sum_{n=-\infty}^{\infty}x(t)\delta(t-nT_s)$

que produce un espectro periódico $X_p(\omega)$,

$$X_p(\omega)=\frac{2\pi}{T_s}\sum_{n=-\infty}^{\infty}X\left(\omega-\frac{2\pi n}{T_s}\right)$$

como se puede apreciar $X_p(\omega)$ es periódica con periodo $\dfrac{2\pi}{T_s}$ y que su periodo central es: $\dfrac{2\pi}{T_s}X(\omega)$.

Se puede obtener, ya sea $x(nT_s)$ o bien $x[n]$ del resultado de la transformada inversa,

$$x(nT_s)=\frac{T_s}{(2\pi)^2}\int_{-\pi/T_s}^{\pi/T_s}X_p(\omega)e^{j\omega nT_s}d\omega$$

con $\Omega=\dfrac{\omega}{f_s}$

$$d\Omega=\frac{d\omega}{f_s}$$

$$x(nT_s)=\frac{T_s}{(2\pi)^2}\int_{-\pi}^{\pi}X_p(\omega)e^{j\Omega n}\frac{d\Omega}{T_s}$$

[1] Véase la sección B.1.3 Serie de Fourier de señales periódicas de tiempo discreto.

$$d\Omega = \frac{d\omega}{f_s}$$

$$d\Omega = T_s d\omega$$

se puede apreciar que para la frecuencia angular normalizada Ω, el intervalo es:

$$-\pi \le \Omega \le \pi$$

$$x(nT_s) = \frac{1}{2\pi}\int_{-\pi}^{\pi} X\left(\frac{\Omega}{T_s}\right) e^{j\Omega n}\frac{d\Omega}{T_s}$$

de la propiedad de escalamiento en el tiempo

$$x\left(n\frac{T_s}{T_s}\right) = \frac{1}{2\pi}\int_{-\pi}^{\pi} X\left(\frac{\Omega}{T_s}T_s\right) e^{j\Omega n}\frac{d\Omega}{T_s}|T_s|$$

$$x(nT_s) = \frac{1}{(2\pi)^2}\int_{-\pi}^{\pi} X_p(\omega) e^{j\Omega n} d\Omega$$

$$x(nT_s) = \frac{1}{(2\pi)^2}\int_{-\pi}^{\pi} X_p(\Omega f_s) e^{j\Omega n} d\Omega$$

de la propiedad de escalamiento en el tiempo

$$x\left(n\frac{T_s}{T_s}\right) = \frac{|T_s|}{(2\pi)^2}\int_{-\pi}^{\pi} X_p\left(\Omega\frac{T_s}{T_s}\right) e^{j\Omega n} d\Omega$$

$$x[n] = \frac{|T_s|}{(2\pi)^2}\int_{-\pi}^{\pi} X_p(\Omega) e^{j\Omega n} d\Omega$$

$$x[n] = \frac{|T_s|}{(2\pi)^2}\int_{-\pi}^{\pi} \frac{2\pi}{T_s} X(\Omega) e^{j\Omega n} d\Omega$$

$x[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(\Omega) e^{j\Omega n} d\Omega$	$x[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(\Omega) e^{j\Omega n} d\Omega$

(B1.17)

B.1.5 Transformada discreta de Fourier (de señales no periódicas de tiempo discreto.)

Efectuando un ventaneo (windowing) sobre las muestras en la ecuación (B1.16)

$$X(\omega) = \sum_{n=0}^{N-1} x(nT_s) e^{-j\omega T_s n} \qquad \text{(B1.18)}$$

El periodo de la ventana es igual a: NT_s, su frecuencia angular es: $\frac{\omega_s}{N}$.

La separación entre los fasores (incremento entre componentes de la frecuencia angular, i.e.; $\Delta\omega$) es igual a la frecuencia angular de la ventana,

$$\Delta\omega = \frac{\omega_s}{N} \qquad \text{(B1.19)}$$

$$\omega_s = N\Delta\omega, \qquad \text{(B1.20)}$$

con esta ecuación, se puede discretizar el dominio de la frecuencia de la transformada de Fourier, tal que el espectro pueda ser escrito en términos de k en vez de ω como:

$$\omega = k\Delta\omega \tag{B1.21}$$

haciendo $k = N \quad \Rightarrow \quad \omega = \omega_s$

sustituyendo (B1.21) en (B1.18)

$$X(k\Delta\omega) = \sum_{n=0}^{N-1} x(nT_s) e^{-jk\Delta\omega T_s n} \tag{B1.22}$$

de (B1.19) y de $T_s = \dfrac{2\pi}{\omega_s}$ en (B1.22)

$$X(k\Delta\omega) = \sum_{n=0}^{N-1} x(nT_s) e^{-jk\left(\frac{\omega_s}{N}\right)\left(\frac{2\pi}{\omega_s}\right)n}$$

expresando la dependencia única y exclusivamente con la variable k, no mostrando la constante $\Delta\omega$, esto es, haciendo a $X_N[\bullet]$ como una secuencia.

$$X_N[k] = \sum_{n=0}^{N-1} x(nT_s) e^{-jk\left(\frac{2\pi}{N}\right)n} \tag{B1.23}$$

haciendo $$W_N = e^{-j\frac{2\pi}{N}} \tag{B1.24}$$

de (B1.23) y (B1.24)

$$X_N[k] = \sum_{n=0}^{N-1} x(nT_s) W_N^{kn} \tag{B1.25}$$

Expresando la dependencia exclusivamente con la variable n, no mostrando la constante T_s, esto es, haciendo a $x[\bullet]$ una secuencia.

$$X_N[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \qquad \text{para} \quad 0 \le k \le N-1 \tag{B1.26}$$

Efectuando un ventaneo (windowing) sobre las muestras en la ecuación (B1.16a)

$$x(nT_s) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\omega) e^{j\omega T_s n} d\omega \tag{B1.26a}$$

por analogía, al desarrollo anterior, el periodo de la ventana es: NT_s, su frecuencia angular es: $\frac{\omega_s}{N}$.

La separación entre los fasores (incremento entre componentes de la frecuencia angular, i.e. $\Delta\omega$)

Es igual a la frecuencia angular de la ventana,

$$\Delta\omega = \frac{\omega_s}{N} \tag{B1.27}$$

$$\omega_s = N\Delta\omega \tag{B1.28}$$

Con la ecuación (B1.27) se puede discretizar el dominio de la frecuencia de la transformada inversa de Fourier (IFT, por sus siglas en inglés) tal que el espectro pueda ser escrito en términos de k en vez de ω como:

$$\omega = k\Delta\omega \tag{B1.29}$$

Haciendo $k = N \quad \Rightarrow \quad \omega = \omega_s$.

Sustituyendo (B1.29) en (B1.26a)

$$x(nT_s) = \frac{1}{N\Delta\omega}\sum_{k=0}^{N-1} X(k\Delta\omega)e^{jk\Delta\omega T_s n}\Delta\omega \tag{B1.30}$$

de (B1.27) y de $T_s = \frac{2\pi}{\omega_s}$ en (B1.30)

$$x(nT_s) = \frac{1}{N}\sum_{k=0}^{N-1} X(k\Delta\omega)e^{jk\left(\frac{\omega_s}{N}\right)\left(\frac{2\pi}{\omega_s}\right)n}$$

expresamos la dependencia únicamente con la variable k, no mostrando la constante $\Delta\omega$, esto es, haciendo $X_N[\bullet]$ una secuencia

$$x(nT_s) = \frac{1}{N}\sum_{k=0}^{N-1} X_N[k]e^{jk\left(\frac{2\pi}{N}\right)n} \tag{B1.31}$$

haciendo

$$W_N = e^{-j\frac{2\pi}{N}} \tag{B1.32}$$

de (B1.31) y (B1.32)

$$x(nT_s) = \frac{1}{N}\sum_{k=0}^{N-1} X_N[k] W_N^{-kn} \tag{B1.33}$$

expresando la dependencia exclusivamente con la variable n, no mostrando la constante T_s, esto es, haciendo a $x[\bullet]$ como una secuencia

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X_N[k] W_N^{-kn} \qquad \text{para} \qquad 0 \le n \le N-1 \tag{B1.34}$$

Referencias bibliográficas.

TMS320C5x Digital Signal Processing. Teaching Kit, Instructor guide, Texas Instruments, USA, 1997.

Appendix B.2
Utilizando la FFT

Las muestras de una señal constituyen su representación en el dominio temporal o en el dominio espacial. Esta representación da las amplitudes de la señal en instantes de tiempo o puntos específicos en el espacio 2D o 3D durante o donde el muestreo ha sido efectuado. De cualquier manera, en muchos casos se quiere conocer el contenido de frecuencias de una señal más que las amplitudes de las muestras individuales. La representación de una señal en términos de sus componentes de frecuencia individuales se conoce como representación en el dominio de frecuencia y podría dar más entendimiento de la señal y del sistema del cual ésta fue generada.

El algoritmo usado para transformar muestras de datos (secuencias de datos) desde el dominio temporal o espacial al dominio de la frecuencia se le conoce como *transformada discreta de Fourier* o DFT. La DFT establece la relación entre las muestras de una señal en el dominio temporal o espacial y su representación en el dominio de la frecuencia. La DFT es ampliamente usada en los campos del análisis espectral, mecánica aplicada, acústica, imágenes médicas, análisis numérico, instrumentación y telecomunicaciones (LabVIEW, 1998).

Los sistemas digitales frecuentemente usan una frecuencia digital, la cual es la razón entre la frecuencia analógica y la frecuencia de muestreo:

frecuencia digital = frecuencia analógica / frecuencia de muestreo

Esta frecuencia digital se conoce como frecuencia normalizada y sus unidades son [ciclos / muestra].

Suponga que ha obtenido f_s muestras de una señal. Si aplicamos la DFT a las N muestras de esta representación en el dominio temporal o espacial de la señal, el resultado también tiene N muestras, pero la información que contiene es de la representación en el dominio frecuencial. La relación entre las N muestras en el dominio del tiempo o del espacio y las N muestras del domino de la frecuencia es la siguiente:

Si la señal es muestreada a una razón de muestreo de f_s [Hz] o el inverso de la unidad de longitud, entonces el intervalo en tiempo o en espacio de las muestras (esto es, el intervalo de muestreo o período de muestreo) es

(tiempo)

$$\boxed{\Delta t = \frac{1}{f_s}}$$

o equivalentemente, (B2.1)

(espacio, una dimensión)

$$\boxed{\Delta d = \frac{1}{f_s}}$$

Las señales muestreadas son denotada por $x[n],\ 0 \le n \le N-1$ (esto es, se tiene un total de N muestras). Cuando la transformada discreta de Fourier, dada por

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi mk/N} \qquad k = 0,1,2,\ldots,N-1 \tag{B2.2}$$

se aplica a estas N muestras, la salida resultante ($X[k],\ 0 \le k \le N-1$) es la representación en el dominio de la frecuencia de $x[n]$. Note que ambos el dominio temporal o espacial x y el dominio de frecuencias X tienen un total de N muestras. Análogamente al *espaciamiento en tiempo o en espacio* Δt o Δd entre las muestras de x en el dominio del tiempo o del espacio, se tiene un espaciamiento de las frecuencias de

$$\Delta f = \frac{f_s}{N} = \frac{1}{N\Delta(t \text{ ó } d)} \tag{B2.3}$$

entre las componentes de X en el dominio de la frecuencia, Δf es también conocida como la resolución en frecuencia. Para incrementar la resolución en frecuencia (menor Δf) se debe incrementar también el número de muestras N (con f_s constante) o disminuir la frecuencia de muestreo f_s (con N constante).

Si introducimos la información muestreada en una matriz, cada elemento en la matriz tiene una muestra (este es el caso más común) en este caso la frecuencia máxima que la matriz puede mantener es f_s; pero, si por ejemplo, introducimos las muestras únicamente en los elementos pares de la matriz y llenamos los elementos intermedios nones con cero, se tiene que la máxima frecuencia que la matriz puede mantener es de dos veces la frecuencia de muestreo. Se puede ver que las muestras se pueden introducir de una manera más espaciada, por ejemplo una muestra cada cuatro u ocho elementos de la matriz entonces la máxima frecuencia que la matriz puede mantener es de $4f_s$ y $8f_s$ respectivamente. Haciendo esto, se puede mostrar una interesante propiedad de la DFT, esto es, la periodicidad de los datos (muestras) y de la secuencia transformada.

Representación de secuencia de datos	Representación de la secuencia transformada
$x[n+lN]$ $l = \ldots,-1,0,1,\ldots$	$X[k+mN]$ $m = \ldots,-1,0,1,\ldots$

donde N para el caso general representa el número de muestras, no el tamaño de la matriz.

Para evitar aliasing debemos satisfacer el teorema de Nyquist, el cual dice que la máxima frecuencia de la señal (es decir, la componente de frecuencia más alta) debe ser menor o igual que la mitad de la frecuencia de muestreo, esto es

$$f_{msignal} \leq \frac{f_s}{2} \qquad \text{(B2.4)}$$

donde $f_{msignal}$ es la máxima frecuencia de la señal. Entonces la frecuencia de Nyquist se define como:

$$f_{Nyq} = \frac{f_s}{2} \qquad \text{(B2.5)}$$

Observamos que N muestras de la señal de entrada resultan en N muestras de la DFT. Esto es, el número de muestras en el dominio del tiempo o del espacio es el mismo que en el dominio de la frecuencia. De la definición de DFT, observamos que sin importar si la señal de entrada $x[n]$ es real o compleja, $X[k]$ es siempre complejo (aún cuando la parte imaginaria sea cero). Así, debido a que la DFT es compleja, esta contiene dos partes de información – la magnitud y la fase (Aboites, 1998). De esto resulta que para una señal $x[n]$ real, la DFT es simétrica con las siguientes propiedades:

$$|X[k] = |X[N-1-k] \qquad \text{(B2.6)}$$

y

$$fase(X[k]) = -fase(X[N-1-k])$$

Los términos usados para describir esta simetría son que la magnitud de $X[k]$ es simétrica par, y la fase ($X[k]$) es simétrica impar. Una señal simétrica par, es aquella que es simétrica sobre el eje y, mientras que una señal simétrica impar es simétrica sobre el origen.

El efecto total de esta simetría es que existe repetición de información contenida en las N muestras de la DFT. Debido a esta repetición de información, únicamente la mitad de las muestras de la DFT necesitan ser calculadas o desplegadas, la otra mitad se puede obtener de esta repetición. Si la señal de entrada es compleja la DFT será no simétrica y no podremos usar este truco.

Si el intervalo de muestreo es de Δt segundos o Δd unidad de longitud, y la primera muestra de datos ($k = 0$) es a 0 segundos o cero unidades de longitud, entonces la k^{th} muestra de datos ($k > 0$, k entero) esta a $k(\Delta t \; o \; \Delta d)$ segundos o unidades de longitud. De manera similar, si la resolución en frecuencia es Δf (Hz o 1/unidades de longitud) ($\Delta f = \frac{f_s}{N}$) entonces la k^{th} muestra de la DFT ocurre a una frecuencia de $k\Delta f$ (Hz o 1/unidades de longitud). Como se verá más adelante, esto es válido solo para la primera mitad de las componentes de frecuencia. La otra mitad representa componentes de

frecuencia negativas. Dependiendo de si el número de muestras, N, es par o impar, se puede tener una diferente interpretación de la frecuencia correspondiente a la k^{th} muestra de la DFT.

Por ejemplo, el caso más común en donde el número de muestras N es igual al tamaño de la matriz, suponga que N es par y sea $p = \frac{N}{2}$. La siguiente tabla muestra las frecuencias que corresponden a cada uno de los elementos de la secuencia de salida X.

Note que el p^{th} elemento, $X[p]$, corresponde a la frecuencia Nyquist. Las entradas negativas en la segunda columna más allá de la frecuencia de Nyquist representan frecuencias negativas. Si $N = 8$, $p = \frac{N}{2} = 4$, entonces

X[0]	DC
X[1]	Δf
X[2]	$2\Delta f$
X[3]	$3\Delta f$
X[4]	$4\Delta f$ (frecuencia Nyquist)
X[5]	$-3\Delta f$
X[6]	$-2\Delta f$
X[7]	$-\Delta f$

Aquí, $X[1]$ y $X[7]$ tendrán la misma magnitud, $X[2]$ y $X[6]$ tendrán la misma magnitud, y $X[3]$ y $X[5]$ tendrán la misma magnitud. La diferencia es que mientras que $X[1]$, $X[2]$, y $X[3]$ corresponden a componentes de frecuencia positiva, $X[5]$, $X[6]$, y $X[7]$ corresponden a componentes de frecuencia negativa. Note que $X[4]$ está a la frecuencia de Nyquist. Esta representación donde se pueden ver tanto frecuencias positivas como negativas, se conoce como *trasformada bilateral*.

Note que cuando N es impar, no hay componente a la frecuencia de Nyquist.

Si $N = 7$, $p = \frac{N-1}{2} = \frac{7-1}{2} = 3$, y se tiene

X[0]	DC
X[1]	Δf
X[2]	$2\Delta f$
X[3]	$3\Delta f$
X[4]	$-3\Delta f$
X[5]	$-2\Delta f$
X[6]	$-\Delta f$

Ahora $X[1]$ y $X[6]$ tienen la misma magnitud, $X[2]$ y $X[5]$ tienen la misma magnitud, y $X[3]$ y $X[4]$ tienen la misma magnitud. De cualquier manera, mientras $X[1]$, $X[2]$, y $X[3]$ corresponden a componentes de frecuencia positiva $X[4]$, $X[5]$, $X[6]$ corresponden a componentes de frecuencia negativa. Debido a que N es impar no existe componente a la frecuencia de Nyquist. Esta también es una transformada *bilateral*, debido a que se tienen frecuencias positivas y negativas.

Para obtener la frecuencia de DC en la mitad de la matriz, se usa la propiedad de traslación de la DFT,

$$\boxed{e^{i2\pi t f_{Nyq}} x[t] \rightarrow X[f + f_{Nyq}]}$$

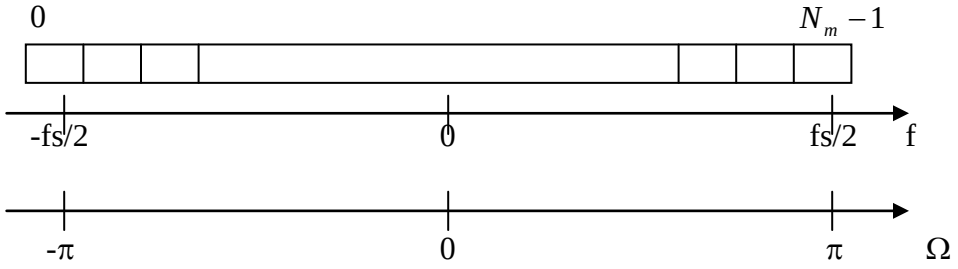
o equivalentemente, (B2.7)

$$\boxed{e^{i2\pi d f_{Nyq}} x[d] \rightarrow X[f + f_{Nyq}]}$$

obteniendose un corrimiento de Nyquist, que produce un corrimiento en las componentes de frecuencia como se muestra enseguida,

X[0]	$-3\Delta f$
X[1]	$-2\Delta f$
X[2]	$-\Delta f$
X[3]	DC
X[4]	Δf
X[5]	$2\Delta f$
X[6]	$3\Delta f$

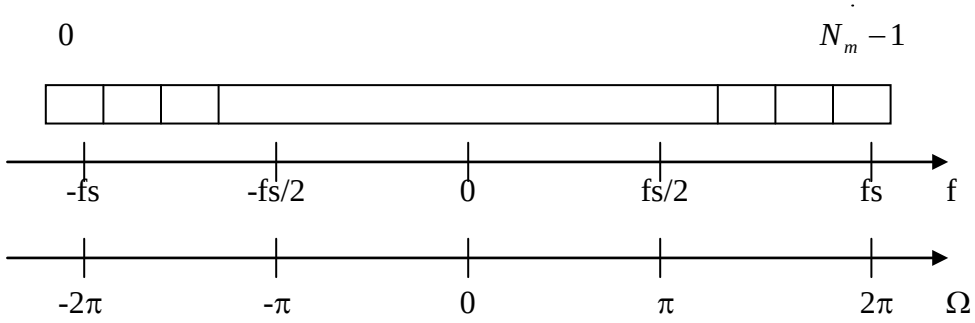
Como podemos observar, las N muestras de datos se cargan directamente en N_m elementos de la matriz, cuando aplicamos el corrimiento de Nyquist y después DFT se obtiene la transformada de Fourier de la secuencia de datos dentro de los siguientes intervalos



con Ω es la frecuencia angular normalizada $\left[\frac{radianes}{muestra}\right]$.

donde N_m es el número de elementos en la matriz. En este caso $N_m = N$.

Si por ejemplo introducimos en los elementos pares de la matriz las muestras y llenamos los otros elementos entre muestras con ceros, tenemos que la máxima frecuencia que puede contener la matriz es dos veces la frecuencia de muestreo:



donde N_m es el número de elementos en la matriz. En este caso $N_m = 2N$.

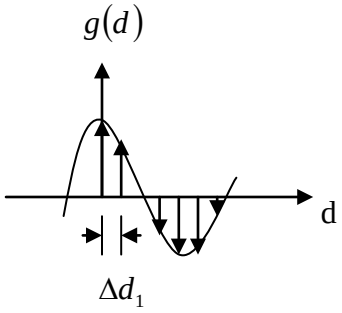
Para tener un claro entendimiento de la relación entre el tamaño de la matriz y la frecuencia de muestreo, implementamos el siguiente ejemplo (en dos casos):

Sea la señal continua (esta se ilustra con una variable espacial pero es equivalente usar una variable temporal):

$$g(d) = \cos(2\pi f_{signal} d)$$

Escogimos esta debido a que el valor inicial corresponde a una cresta, esta señal será muestreada y si su frecuencia f_{signal} es igual a la frecuencia Nyquist f_{Nyq} entonces en este caso las muestras en un período serán un máximo y un mínimo.

Si introducimos la información muestreada en una matriz, cada elemento en la matriz tiene una muestra obtenida en un intervalo de espacio Δd_1 (este es el caso más común) en este caso la frecuencia máxima que la matriz puede tener es f_s. El muestreo de $g(d)$ se ilustra en la siguiente figura:



$\Delta d_1 = 1$ [pixel / muestras] y $f_s = \frac{1}{\Delta d_1} = 1$ [muestras / pixel]

si $f_{signal} = \frac{10}{256}$ [ciclos / pixel] y $T_{signal} = \frac{256}{10}$ [pixeles / ciclo]

$$f_{Norm} = \frac{f_{signal}}{f_s} = \frac{\frac{10}{256}}{1} \text{ [ciclos / pixel] / [muestras / pixel]}$$

$$f_{Norm} = \frac{10}{256} \text{ [ciclos / muestra]}$$

recordando,

$$\Delta f = \frac{f_s}{N}$$

en este caso si el número de muestras es $N = 256$ [muestras]

$$\Delta f = \frac{1}{256} \quad \text{[1 / pixel]}$$

después de aplicar DFT cada elemento en la matriz representa una componente de frecuencia $G(f)$, usamos

$$f_n = (\Delta f)n \qquad n = 0,1,2,3,\ldots, N_m - 1$$

donde n es un índice entero que va de cero a $N_m - 1$.

Como se ha establecido en este ejemplo, el número de muestras es igual al número de elementos de la matriz.

$$N_m = N$$

donde N_m es el número de elementos en la matriz.

N es el número de muestras.

entonces el intervalo en frecuencia que se obtiene después de aplicar DFT a la secuencia de datos es:

$$-\frac{N_m(\Delta f)}{2} \le f \le \frac{N_m(\Delta f)}{2}$$

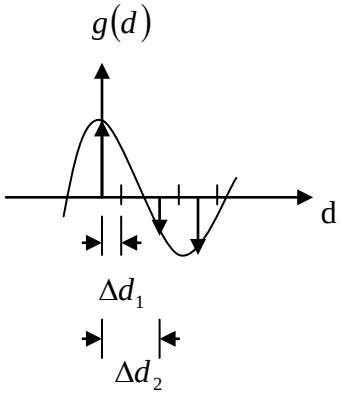
reemplazando N_m y Δf , obtenemos

$$-\frac{N}{2}\left(\frac{f_s}{N}\right) \le f \le \frac{N}{2}\left(\frac{f_s}{N}\right)$$

$$\boxed{-\frac{f_s}{2} \le f \le \frac{f_s}{2}} \qquad \text{(B2.8)}$$

Ahora, si realizamos el muestreo como se ilustra en la siguiente figura y "alimentamos" solamente los elementos pares de la matriz con las muestras y llenamos los elementos impares con cero, entonces tenemos que la máxima frecuencia que la matriz puede contener es dos veces la frecuencia de muestreo (este es el caso menos común)

El muestreo de $g(d)$ se muestra en la siguiente figura:



$\Delta d_2 = 2\Delta d_1$ [pixeles / muestra] y $f_s = \frac{1}{\Delta d_2}$ [muestras / pixel]

$\Delta d_2 = 2$ [pixeles / muestra] y $f_s = \frac{1}{2}$ [muestras / pixel]

si $f_{signal} = \frac{10}{256}$ [ciclos / pixel] y $T_{signal} = \frac{256}{10}$ [pixeles / ciclo]

$$f_{Norm} = \frac{f_{signal}}{f_s} = \frac{\frac{10}{256}}{\frac{1}{2}} \text{ [ciclos / pixel] / [muestras / pixel]}$$

$$f_{Norm} = \frac{20}{256} \text{ [ciclos / muestra]}$$

recordando,

$$\Delta f = \frac{f_s}{N}$$

en este caso si el número de muestras es $N = 128$ [muestras]

$$\Delta f = \frac{\frac{1}{2}}{128} = \frac{1}{256} \qquad [1 / \text{pixel}]$$

después de aplicar DFT cada elemento en la matriz representa una componente en frecuencia de $G(f)$, usamos

$$f_n = (\Delta f)n \qquad n = 0,1,2,3,\ldots, N_m - 1$$

donde n es un índice entero que va de cero a $N_m - 1$.

Como hemos establecido en este ejemplo, el número de elementos en la matriz es dos veces el número de muestras.

$$N_m = 2N$$

donde N_m es el número de elementos en la matriz.

N es el número de muestras.

entonces el intervalo en frecuencia que se obtiene después de aplicar DFT a la secuencia de datos es:

$$-\frac{N_m(\Delta f)}{2} \le f \le \frac{N_m(\Delta f)}{2}$$

reemplazando N_m y Δf, obtenemos

$$-\frac{2N}{2}\left(\frac{f_s}{N}\right) \le f \le \frac{2N}{2}\left(\frac{f_s}{N}\right)$$

$$\boxed{-f_s \le f \le f_s} \qquad \text{(B2.9)}$$

La implementación directa de la DFT en N muestras de datos requiere aproximadamente N^2 operaciones complejas por lo que es un proceso que consume bastante tiempo. De cualquier manera, cuando el tamaño de la secuencia es una potencia de 2,

$$N = 2^m \qquad \text{para} \qquad m = 1,2,3,\ldots$$

se puede implementar el calculo de la DFT con aproximadamente $N\log_2(N)$ operaciones. Esto hace el cálculo de la DFT mucho más rápido, existe un algoritmo conocido como F*ast Fourier Transforms* (FFT), transformada rápida de Fourier. La FFT es solo un algoritmo rápido para calcular la DFT cuando el número de muestras (N) es una potencia de 2.

Las ventajas de la FFT incluyen velocidad y eficiencia en memoria. El tamaño de la secuencia de entrada, de cualquier modo debe ser potencia de 2. La DFT puede procesar eficientemente una secuencia de cualquier tamaño, pero es más lenta y usa más memoria que la FFT.

Una técnica empleada para hacer el tamaño de la secuencia de entrada sea igual a una potencia de 2 es agregar ceros al final de las secuencias, así el número total de muestras es igual a la potencia de 2 más próxima.

El agregar estos ceros a la señal en el dominio temporal no afecta el espectro de la señal. Además, el hacer el número total de muestras una potencia de dos para hacer los cálculos más rápidos mediante la FFT, la adición de ceros también ayuda a incrementar la resolución en frecuencia (recuerde que $\Delta f = \frac{f_s}{N}$, por el incremento en el número de muestras, N).

Todos los resultados previos se pueden extender a dos o más dimensiones debido a la propiedad de linealidad de la DFT,

$$\boxed{ax[t] + by[t] \rightarrow aX[f] + bY[f]}$$

o equivalentemente (B2.10)

$$\boxed{ax[d] + by[d] \rightarrow aX[f] + bY[f]}$$

Escribimos la definición de la DFT para una función bidimensional $x[m,n]$ donde $0 \le m \le M-1$, $0 \le n \le N-1$, con m y n siendo índices enteros.

$$\boxed{X_{kl} = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x_{mn} e^{-i2\pi(nk+ml)/N}}$$

(B2.11)

Si nos preguntamos porque el espectro se reproduce cada f_m, esto es debido al proceso de muestreo, esto puede ser modelado matemáticamente como (esto se ilustra con una variable temporal pero es equivalente usar una variable espacial):

$$x(t)\cdot T_s comb\left(\frac{t}{T_s}\right) \rightarrow T_s\left[X(f) * comb(T_s f)\right]$$

donde usamos la propiedad de modulación (multiplicación en el tiempo) de la DFT. Recordamos la definición de una función *comb*

$$comb(f) = \sum_{n=-\infty}^{\infty} \delta(f-n)$$

$$\Im\left\{x(t)\cdot T_s comb\left(\frac{t}{T_s}\right)\right\} = T_s\left(X(f) * \sum_{n=-\infty}^{\infty}\delta(T_s f - n)\right)$$

$$= T_s\left(X(f) * \sum_{n=-\infty}^{\infty}\delta\left(T_s\left[f - \frac{n}{T_s}\right]\right)\right),$$

usando la propiedad de escalamiento de la función delta de Dirac,

$$\delta(at) = \frac{1}{|a|}\delta(t)$$

se tiene:

$$= T_s\left(X(f) * \frac{1}{T_s}\sum_{n=-\infty}^{\infty}\delta\left(f - \frac{n}{T_s}\right)\right)$$

$$= X(f) * \sum_{n=-\infty}^{\infty}\delta(f - f_s n)$$

$$\boxed{\Im\left\{x(t)\cdot T_s comb\left(\frac{t}{T_s}\right)\right\} = \sum_{n=-\infty}^{\infty} X(f - f_s n)} \qquad \text{(B2.12)}$$

donde f_s es la frecuencia de muestreo.

$T_s = \frac{1}{f_s}$ es el período de muestreo.

La última ecuación muestra que $X(f)$ se reproduce en el dominio de la frecuencia cada múltiplo entero de f_s.

La dualidad también es cierta, como se muestra a continuación:

$$F_{fs}\left[x(t) * comb(F_{fs}t)\right] \rightarrow X(f) \cdot F_{fs} comb\left(\frac{f}{F_{fs}}\right)$$

donde usamos la propiedad de convolución de la DFT y $F_{fs} = \Delta f$ la cual representa el intervalo entre dos elementos consecutivos en el dominio de la frecuencia, y es el intervalo de muestreo en el dominio de la frecuencia. Recordando la definición de una función $comb$.

$$comb(t) = \sum_{n=-\infty}^{\infty} \delta(t-n)$$

$$\Im^{-1}\left\{X(f) \cdot F_{fs} comb\left(\frac{f}{F_{fs}}\right)\right\} = F_{fs}\left(x(t) * \sum_{n=-\infty}^{\infty} \delta(F_{fs}t - n)\right)$$

$$= F_{fs}\left(x(t) * \sum_{n=-\infty}^{\infty} \delta\left(F_{fs}\left[t - \frac{n}{F_{fs}}\right]\right)\right)$$

usando la propiedad de escalamiento de la función delta de Dirac,

$$\delta(at) = \frac{1}{|a|}\delta(t)$$

$$= F_{fs}\left(x(t) * \frac{1}{F_{fs}} \sum_{n=-\infty}^{\infty} \delta\left(t - \frac{n}{F_{fs}}\right)\right)$$

$$= x(t) * \sum_{n=-\infty}^{\infty} \delta(t - T_{fs}n)$$

$$\boxed{\Im^{-1}\left\{X(f) \cdot F_{fs} comb\left(\frac{f}{F_{fs}}\right)\right\} = \sum_{n=-\infty}^{\infty} x(t - T_{fs}n)} \qquad \text{(B2.13)}$$

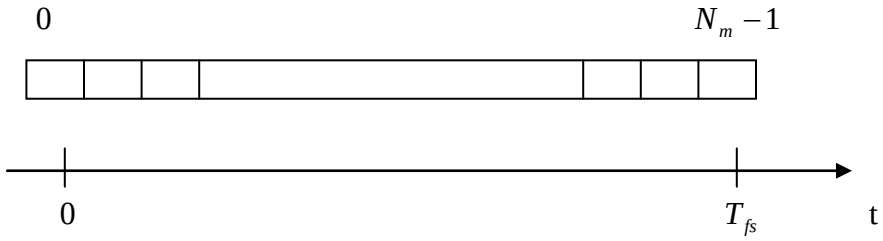
donde F_{fs} es el intervalo de muestreo en el dominio de la frecuencia.

$$T_{fs} = \frac{1}{F_{fs}} = \frac{1}{\Delta f} = \frac{N}{f_s} = N\Delta t$$

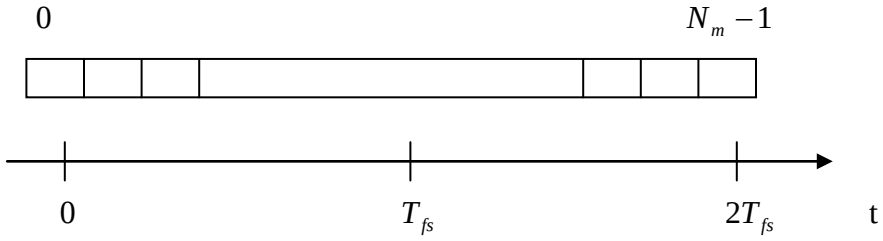
con T_{fs} es el período de muestreo en frecuencia.

La última ecuación muestra que $x(t)$ se reproduce en el tiempo o en el dominio espacial cada múltiplo entero de T_{fs}, o equivalentemente cada intervalo de tiempo $N\Delta t$ (esto corresponde a una secuencia muestreada).

Cuando en el dominio de la frecuencia cargamos en los elementos pares o impares de la matriz para calcularle su DFT inversa. Obtenemos similarmente la reproducción de las secuencias muestreadas en el dominio temporal o especial, como se explicó anteriormente. En este caso cuando $N_m = N$



En el caso cuando $N_m = 2N$



Referencias bibliográficas.

Aboites, V., Casillas, M.A., *Apuntes de Métodos Matemáticos 1*, CIO, León Gto., 1998.
LabVIEW User Manual, National Instruments, USA, 1998.

Appendix B.3
Generación y transformada de Fourier de señales digitales con LabVIEW

La frecuencia normalizada esta definida por:

$$\boxed{f_{norm} = \frac{f_{signal}}{f_s}} \qquad \text{(B3.1)}$$

Es conveniente trabajar con la frecuencia normalizada debido a que una forma alterna a la relación (B3.1) de obtener esta frecuencia (f_{norm}) es mediante "máximos absolutos", esto es, el número de ciclos máximo que se quieren en el número de muestras totales. Con f_{norm} conocido, de (B3.1) solo es necesario definir la frecuencia de la señal f_{signal} o la frecuencia de muestreo f_s, para conocer la frecuencia desconocida.

Como ejemplo supóngase que se desean N=128 muestras en 30 ciclos de una señal senoidal[1]:

$$f_{norm} = \frac{30[ciclos]}{128[muestras]} = 0.234375 \quad \frac{[ciclos]}{[muestra]} \qquad \text{(B3.2)}$$

Se puede seleccionar la frecuencia de muestreo como:

$$T_s = 1\frac{[seg]}{[muestra]} \quad \therefore$$

$$f_s = \frac{1}{T_s} = \frac{1[muestra]}{1[seg]} \qquad \text{ó} \qquad 1[Hz] \text{ de muestreo.} \qquad \text{(B3.3)}$$

faltando por conocer de (B3.1) la frecuencia de la señal,

$$f_{signal} = f_{norm} f_s \qquad \text{(B3.4)}$$

Sustituyendo (B3.2) y (B3.3) en (B3.4)

$$f_{signal} = \left(\frac{30[ciclos]}{128[muestra]}\right)\left(1\left[\frac{muestra}{seg}\right]\right) = \frac{30}{128}\left[\frac{ciclos}{seg}\right] \qquad \text{ó} \qquad [Hz]$$

[1] En este caso la señal senoidal es un seno que es una función fundamental que no tiene más que una componente en frecuencia, cuando se trate a otro tipo de señales no fundamentales, f_{signal} corresponderá a la frecuencia máxima de la señal.

Al obtener la transformada discreta de Fourier de la señal seno digital de este ejemplo, su resolución en frecuencia (Δf) , esta dada como:

$$\boxed{\Delta f = \frac{f_s}{N}} \qquad \text{(B3.5)}$$

por lo tanto de (B3.5)

$$\Delta f = \frac{1\left[\frac{muestras}{seg}\right]}{128[muestras]} = \frac{1}{128}\left[\frac{1}{seg}\right] \quad \text{ó}$$

$$\Delta f = \frac{1}{128} \quad [Hz] \quad \text{ó} \quad \Delta f = 0.0078125 \quad [Hz].$$

Por otra parte si ahora lo que se conoce es la frecuencia de la señal senoidal, que es de $60[Hz]$ ó $\left[\frac{ciclos}{seg}\right]$.

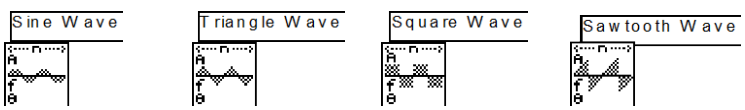
De (B3.1) tenemos que:

$$f_s = \frac{f_{signal}}{f_{norm}} = \frac{60\left[\frac{ciclos}{seg}\right]}{0.234375\left[\frac{ciclos}{muestra}\right]} = 256\left[\frac{muestras}{seg}\right].$$

Para el cálculo de la resolución en frecuencia de (B3.5) se tiene:

$$\Delta f = \frac{256\left[\frac{muestras}{seg}\right]}{128[muestras]} = 2 \quad \left[\frac{1}{seg}\right] \text{ ó} \quad 2 \quad [Hz].$$

La generación de señales digitales mediante LabVIEW se obtiene con funciones tales como[2]:



donde f en estas funciones es la frecuencia normalizada (f_{norm}).

[2] Estas funciones se encuentran en: las funciones del diagrama a bloques>> Signal Processing >> Signal Generation en LabVIEW ver. 6i.

Existen diferentes razones que pueden satisfacer la misma f_{norm} pero tienen diferentes f_{signal} y f_s. Tal como se mostró en el ejemplo anterior, sintetizando:

Caso I.

$$f_{norm} = \frac{f_{signal}}{f_s} = \frac{30\left[\frac{ciclos}{seg}\right]}{1\left[\frac{muestra}{seg}\right]} = 0.234375\left[\frac{ciclos}{muestra}\right]$$

$$\Delta f = 0.0078125 \qquad [Hz]$$

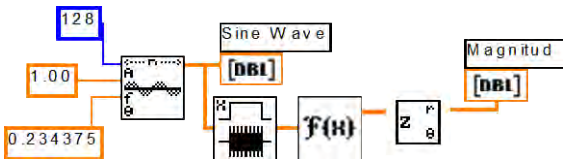
como se observa de este resultado, cuando se utilizan "máximos absolutos" para determinar la frecuencia normalizada (f_{norm}), en realidad se esta implicando una frecuencia de muestreo $f_s = 1\left[\frac{muestra}{seg}\right]$. Aunque por otro lado pueden existir infinitas razones de frecuencias de la señal (f_{signal}) a frecuencias de muestreo (f_s) que den la misma frecuencia normalizada $f_{norm} = 0.234375\left[\frac{ciclos}{muestra}\right]$, tal como:

Caso II.

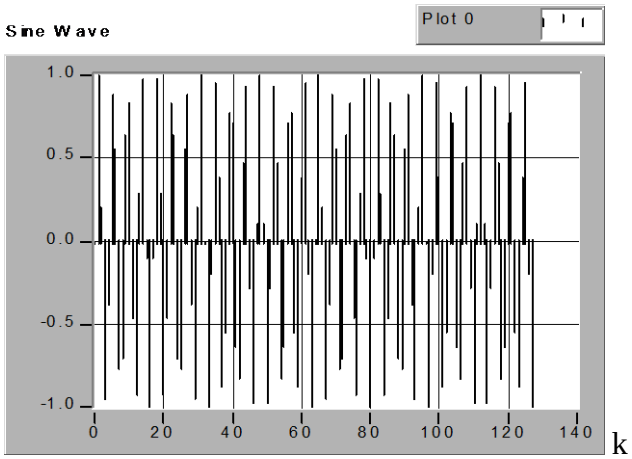
$$f_{norm} = \frac{f_{signal}}{f_s} = \frac{60\left[\frac{ciclos}{seg}\right]}{256\left[\frac{muestras}{seg}\right]} = 0.234375\left[\frac{ciclos}{muestra}\right]$$

$$\Delta f = 2 \qquad [Hz]$$

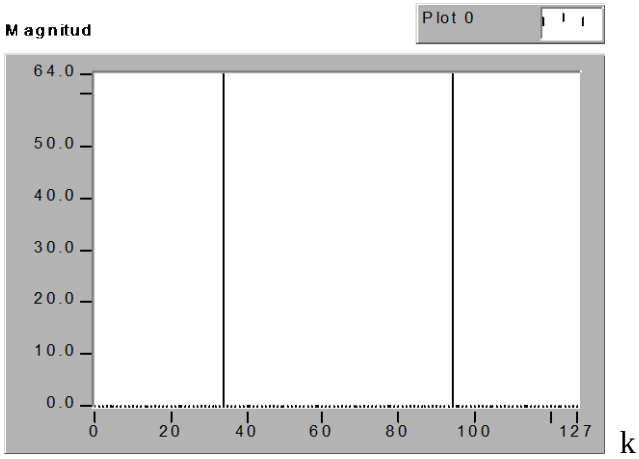
Si generamos una señal seno digital de $f_{norm} = 0.234375 \left[\frac{ciclos}{muestra}\right]$, el programa toma la forma:



Se obtiene la siguiente gráfica:



Graficamos la magnitud de la transformada de Fourier discreta:

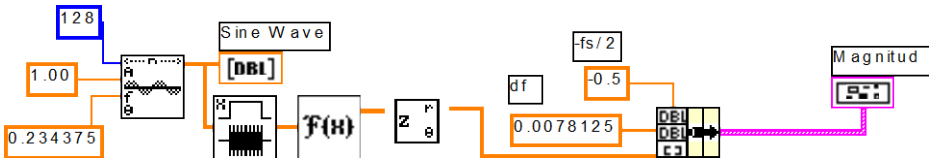


$$0 \qquad \frac{N-1}{2} \qquad N-1$$

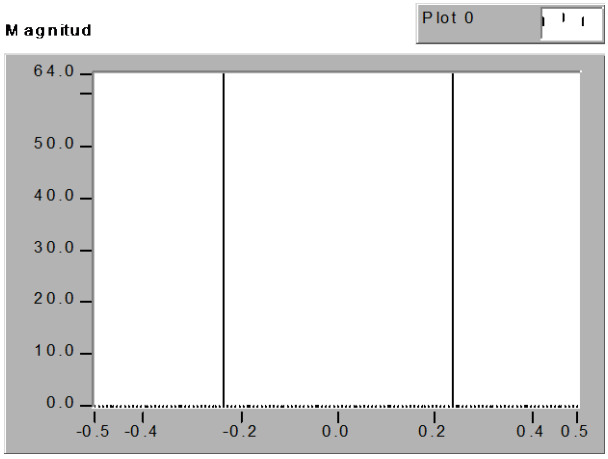
con N siendo el número de elementos (datos) en el arreglo (vector).

Como se observa en el eje de las abscisas de esta gráfica, los valores indicados corresponden simplemente al índice del arreglo (vector) que contiene la información de la DFT o FFT.

Considerando el Caso I. El programa se modifica para mostrar en el eje de las abscisas del espectro en potencia las frecuencias adecuadas para el caso, el programa es:



La gráfica de la magnitud de la transformada de Fourier discreta es:



f [Hz]

$-\frac{f_s}{2}=-\frac{1}{2}$ -0.2343 0 0.2343 $\frac{f_s}{2}=\frac{1}{2}$

Para ubicar en que elemento del arreglo (vector) se encuentran las deltas mostradas en la figura anterior, con la frecuencia del seno $f_{signal}=0.234375[Hz]$ y la resolución en frecuencia de la transformada de la señal $\Delta f=0.0078125[Hz]$. El número de elementos en el arreglo, con resolución en frecuencia Δf, necesarios para representar una frecuencia de señal f_{signal} es:

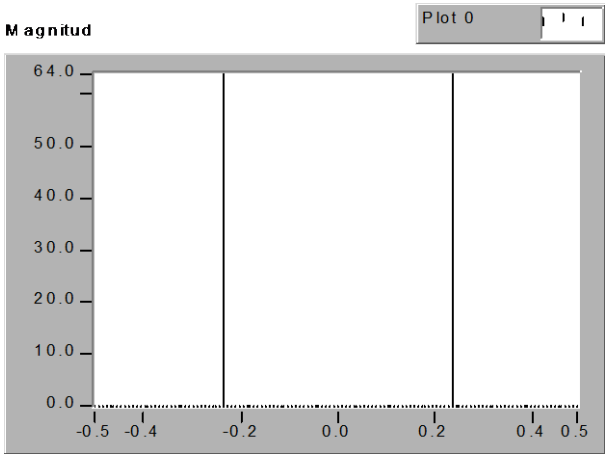
$$\#elementos\ del\ arreglo=\frac{f_{signal}}{\Delta f} \qquad \text{(B3.6)}$$

por lo tanto para este caso I, se tiene de (B3.6):

$$\#elementos\ del\ arreglo=\frac{0.234375[Hz]}{0.0078125[Hz/elemento]}=30[elementos]$$

Debido al corrimiento Nyquist aplicado en el programa, se ha recorrido el cero en frecuencia a la mitad (en el centro) del arreglo, lo cual implica que el cero en frecuencia se encuentra a la mitad ($N/2$) del total de elementos del arreglo (N), este desplazamiento lo denominaremos “offset”. En este ejemplo el offset es:

$$offset=\frac{N}{2}=\frac{128}{2}=64 \qquad [elementos]$$



f [Hz]

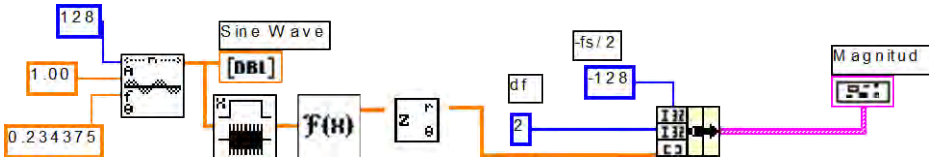
0 34 64 94 127

offset - #elementos

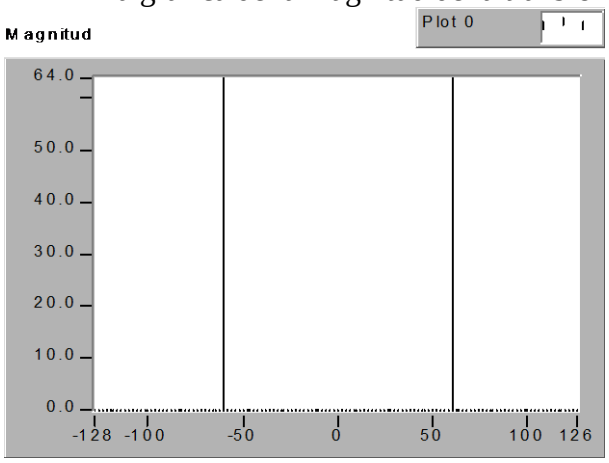
offset

offset + #elementos

Considerando el Caso II. El programa se modifica para mostrar en el eje de las abscisas del espectro en potencia las frecuencias adecuadas para el caso, el programa es:



La gráfica de la magnitud de la transformada de Fourier discreta es:



f [Hz]

$-\frac{f_s}{2} = -128$ -60 0 60 $\frac{f_s}{2} = 126$

Para ubicar en que elemento del arreglo (vector) se encuentran las deltas mostradas en la figura anterior, con la frecuencia del seno $f_{signal} = 60[Hz]$ y la resolución en

frecuencia de la transformada de la señal $\Delta f = 2[Hz]$. El número de elementos en el arreglo, con resolución en frecuencia Δf, necesarios para representar una frecuencia de señal f_{signal} es:

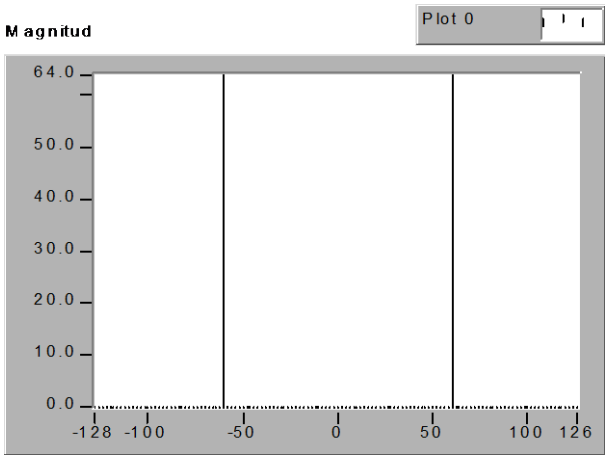
$$\#elementos\ del\ arreglo = \frac{f_{signal}}{\Delta f} \qquad \text{(B3.6)}$$

Ahora para el caso II, se tiene de (B3.6):

$$\#elementos\ del\ arreglo = \frac{60[Hz]}{2[Hz/elemento]} = 30[elementos]$$

Debido al corrimiento Nyquist aplicado en el programa, se ha recorrido el cero en frecuencia a la mitad (en el centro) del arreglo, lo cual implica que el cero en frecuencia se encuentra a la mitad ($N/2$) del total de elementos del arreglo (N), este desplazamiento lo denominaremos "offset". En este ejemplo el offset es:

$$offset = \frac{N}{2} = \frac{128}{2} = 64 \qquad [elementos]$$



0 34 64 94 127

offset - #elementos

offset

offset + #elementos

El almacenamiento típico de las muestras en el arreglo (vector) lo efectuamos como:

$$k_{almacenamiento} = 1\left[\frac{elementos\ de\ almacenamiento}{muestra}\right] \tag{B3.7}$$

$$f_{sc} = k_{almacenamiento} f_s \tag{B3.8}$$

donde f_{sc} es la frecuencia de muestreo computacional.

$k_{almacenamiento}$ es el factor de almacenamiento.
f_s es la frecuencia de muestreo.

Para calcular la resolución en frecuencia computacional (Δf_c), tenemos:

$$\Delta f_c = \frac{f_{sc}}{N_m} = \frac{k_{almacenamiento} f_s}{N_m} \tag{B3.9}$$

con N_m siendo el número de elementos en el arreglo (vector), y N es el número de muestras.

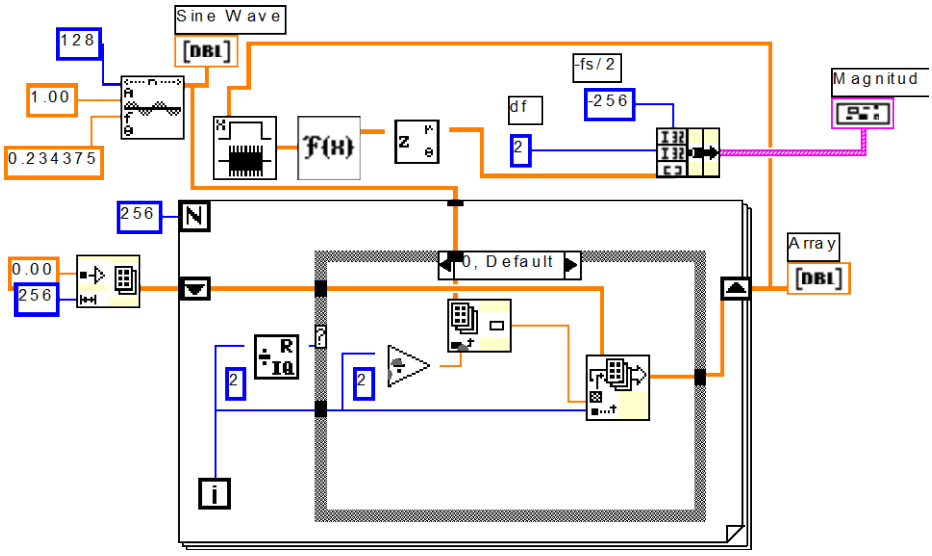
Por ejemplo si $k_{almacenamiento} = 2$, esto implica que: $N_m = 2N$

$$\Delta f_c = \frac{2 f_s}{N_m} = \frac{2 f_s}{2N} = \frac{f_s}{N} \equiv \Delta f \tag{B3.10}$$

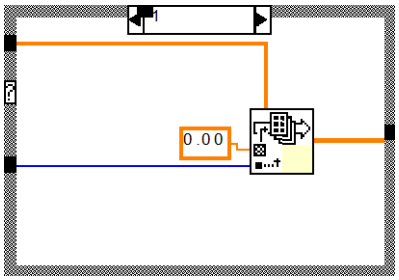
Nota: Para el caso en que se añaden ceros intercalados con las muestras, sin importar la cantidad de ceros que se añadan, modificándose así el factor de almacenamiento $k_{almacenamiento}$ deberá satisfacerse que $\Delta f_c = \Delta f$. Una alternativa a aumentar ceros, es dejar $N_m = N$ y seleccionar ya sea los elementos pares o los nones y hacerlos cero, lo cual implicará en este caso que $\Delta f_c = 2\Delta f$.

Se muestra la reproducción de espectros para el seno del caso II:

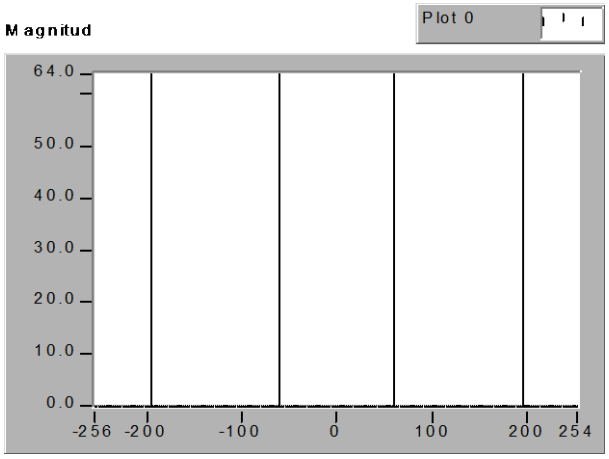
El programa se modifica para mostrar en el eje de las abscisas del espectro en potencia su “reproducción”, el programa es:



El otro caso en el case es:



La gráfica de la magnitud de la transformada de Fourier discreta donde se observa la reproducción del espectro en potencia es:



$-\frac{2f_s}{2} = -256$ -60 0 60 $\frac{2f_s}{2} = 254$

Para ubicar en que elemento del arreglo (vector) se encuentran las deltas mostradas en la figura anterior, con la frecuencia del seno $f_{signal} = 60[Hz]$ y la resolución en frecuencia de la transformada de la señal $\Delta f_c = \Delta f = 2[Hz]$. El número de elementos en el arreglo, con resolución en frecuencia Δf, necesarios para representar una frecuencia de señal f_{signal} es:

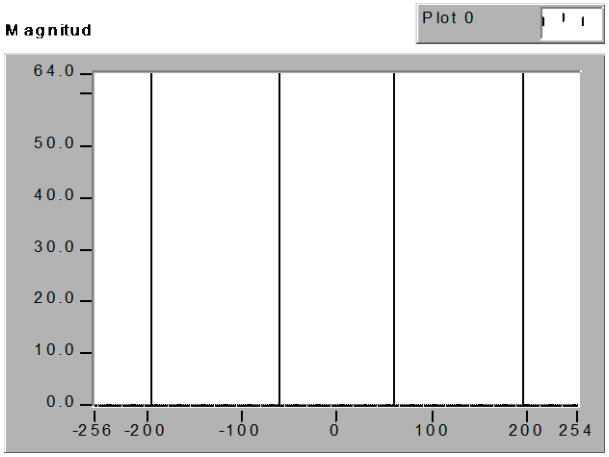
$$\#elementos\ del\ arreglo = \frac{f_{signal}}{\Delta f_c} \tag{B3.11}$$

Se tiene de (B3.11):

$$\#elementos\ del\ arreglo = \frac{60[Hz]}{2[Hz/elemento]} = 30[elementos]$$

Debido al corrimiento Nyquist aplicado en el programa, se ha recorrido el cero en frecuencia a la mitad (en el centro) del arreglo, lo cual implica que el cero en frecuencia se encuentra a la mitad ($N/2$) del total de elementos del arreglo (N), este desplazamiento lo denominaremos “offset”. En este ejemplo el offset es:

$$offset = \frac{N_m}{2} = \frac{2N}{2} = \frac{2(128)}{2} = 128 \quad [elementos]$$



0 98 128 158 255

offset - #elementos

offset

offset + #elementos

Referencias bibliográficas.

LabVIEW User Manual, National Instruments, USA, 1998.

Appendix B

B Análisis de Fourier.

B.1 Relaciones entre serie y transformada de Fourier en dominio (tiempo) continuo y discreto.

B.1.1 Serie de Fourier de señales periódicas de tiempo continuo.

B.1.2 Transformada de Fourier de señales no periódicas en tiempo continuo.

B.1.3 Serie de Fourier de señales periódicas de tiempo discreto.

B.1.4 Transformada de Fourier de señales no periódicas en tiempo discreto.

B.1.5 Transformada discreta de Fourier (de señales no periódicas de tiempo discreto).

Referencias bibliográficas.

B.2 Utilizando la FFT.

Referencias bibliográficas.

B.3 Generación y transformada de Fourier de señales digitales con LabVIEW.

Referencias bibliográficas.

Appendix C
Algunas técnicas para obtención de la fase de un interferograma.

C.0 Synchronous and asynchronous phase detection algoritms.

C.0.1 Synchronous phase detection algorithms.

Suponemos que la frecuencia de la señal detectada y los escalones (steps) de fase tomados durante las mediciones son conocidos.

C.0.2 Asynchronous phase detection algorithms.

Puede suceder, de cualquier forma, que los escalones de fase o la frecuencia de la señal medida sean desconocidos. En ese caso, antes de calcular la fase, se debe determinar la frecuencia de la señal.

C.1 Space domain phase detecting techniques (spatial synchrony of phase).

C.1.1 Three-step algorithm to measure the phase (phase shifting [step]).

Para determinar la fase sin ninguna ambigüedad, son necesarios un mínimo de tres muestras por punto (x,y) del mapa del interferograma.

$$I_1(x,y) = a(x,y) + b(x,y)\cos[\phi(x,y) + \alpha_1]$$

$$I_2(x,y) = a(x,y) + b(x,y)\cos[\phi(x,y) + \alpha_2]$$

$$I_3(x,y) = a(x,y) + b(x,y)\cos[\phi(x,y) + \alpha_3]$$

La expresión para algoritmos de tres muestras por punto del mapa del interferograma, en particular, para $\alpha_1 = 0^o$, $\alpha_2 = 120^o$, $\alpha_3 = -120^o$.

$$\phi(x,y) = \text{tg}^{-1}\left\{\sqrt{3}\frac{I_2 - I_3}{2I_3 - I_2 - I_1}\right\}$$

C.1.2 Space phase demodulation with a linear carrier.

Un análisis de interferometría síncrona típica con frecuencia de portadora lineal, asistido por computadora, requiere de los siguientes cuatro pasos (Carpio,1994): (a) multiplicación de las muestras del interferograma por las funciones seno y coseno con frecuencias iguales a la frecuencia de la portadora lineal de la señal, obteniendose de esta forma dos imágenes; (b) estas multiplicaciones son seguidas por un proceso de suavizado sobre las dos imágenes obtenidas anteriormente para eliminar la señal no deseada que tiene

el doble de la frecuencia de la portadora, obtenida como una consecuencia de las multiplicaciones por el seno y el coseno; (c) para obtener la fase del frente de onda con módulo 2π a partir de las dos imágenes suavizadas anteriores, se utiliza la función arcotangente como un detector de fase, esta fase envuelta es normalmente el mapa de fase del objeto, finalmente, (d) finalmente, el último paso, es el proceso de desenvolvimiento de la fase, el cual consiste en eliminar las discontinuidades cada 2π del mapa de fase.

C.1.2.1 Quadrature phase detection of a sinusoidal signal (direct method).

$$I(x,y) = a(x,y) + b(x,y)\cos[2\pi fx + \phi(x,y)]$$

$$I_c(x,y) = I(x,y)\cos(2\pi fx)$$
$$= a(x,y)\cos(2\pi fx) + \frac{b(x,y)}{2}\cos[\phi(x,y) + 4\pi fx] + \frac{b(x,y)}{2}\cos[\phi(x,y)]$$

$$I_s(x,y) = I(x,y)\operatorname{sen}(2\pi fx)$$
$$= a(x,y)\operatorname{sen}(2\pi fx) + \frac{b(x,y)}{2}\operatorname{sen}[\phi(x,y) + 4\pi fx] + \frac{b(x,y)}{2}\operatorname{sen}[-\phi(x,y)]$$
$$= a(x,y)\operatorname{sen}(2\pi fx) + \frac{b(x,y)}{2}\operatorname{sen}[\phi(x,y) + 4\pi fx] - \frac{b(x,y)}{2}\operatorname{sen}[\phi(x,y)]$$

introduciendo un filtro pasa bajos,

$$M_1(x,y) = \frac{b(x,y)}{2}\cos[\phi(x,y)]$$
$$M_2(x,y) = -\frac{b(x,y)}{2}\operatorname{sen}[\phi(x,y)]$$

$$\phi(x,y) = \operatorname{tg}^{-1}\left(-\frac{M_2(x,y)}{M_1(x,y)}\right)$$

C.1.2.2 Phase-Locked Loop (PLL).

La demodulación por phase-locked loop es otro método (Servin,1993) para análisis de interferograma con portadora lineal. Un PLL puede considerarse como un filtro pasa banda angosto adaptivo cuya frecuencia central se amarra a la frecuencia del patrón de franjas instantáneo a lo largo de la línea de rastreo.

El principio básico en este lazo de amarre de fase es el siguiente: Los cambios de fase de la señal de entrada modulada en fase son comparados con la salida de un oscilador controlado por voltaje (VCO, por sus siglas en inglés) mediante la utilización de un multiplicador. El PLL trabaja de tal forma que la diferencia de fase entre la señal de entrada modulada y la señal de salida del VCO eventualmente disminuye hasta desaparecer. Este

seguimiento en la fase es logrado por medio de un lazo cerrado que alimenta la entrada al VCO con la señal de salida, la cual es proporcional con la señal moduladora. Para la evaluación de interferogramas este VCO no es un dispositivo electrónico sino un VCO simulado por software. Ventajas del método del PLL: Todos los pasos necesarios para obtener el frente de onda continuo demodulado, incluyendo el proceso de desenvolvimiento de fase son efectuados simultáneamente en el PLL. Desventaja: Se requiere conocer la frecuencia de la portadora de la señal. No se conoce la curvatura global del mapa de fase.

C.2 Frequency domain phase detecting techniques.

Es posible calcular la fase de un interferograma con portadora lineal mediante un procedimiento que utilice la transformada de Fourier .

C.2.1 Takeda's method (Fourier transform method).

$$g(x,y) = a(x,y) + b(x,y)\cos[2\pi f_o x + \phi(x,y)]$$

$$\tilde{g}(x,y) = a(x,y) + \frac{b(x,y)}{2}\left[e^{i2\pi f_o x + i\phi(x,y)} + e^{-i2\pi f_o x - i\phi(x,y)}\right]$$

$$\tilde{g}(x,y) = a(x,y) + \frac{b(x,y)}{2}e^{i2\pi f_o x + i\phi(x,y)} + \frac{b(x,y)}{2}e^{-i2\pi f_o x - i\phi(x,y)}$$

$$\tilde{g}(x,y) = a(x,y) + \frac{b(x,y)}{2}e^{i2\pi f_o x}e^{i\phi(x,y)} + \frac{b(x,y)}{2}e^{-i2\pi f_o x}e^{-i\phi(x,y)}$$

making

$$\tilde{c}(x,y) = \frac{b(x,y)}{2}e^{i\phi(x,y)}$$

$$\tilde{g}(x,y) = A(x,y) + \tilde{c}(x,y)e^{i2\pi f_o x} + \tilde{c}^*(x,y)e^{-i2\pi f_o x}$$

realizing the Fourier transform

$$\tilde{G}(x,y) = A(\varsigma,y) + \tilde{C}(\varsigma - f_o, y) + \tilde{C}^*(\varsigma + f_o, y)$$

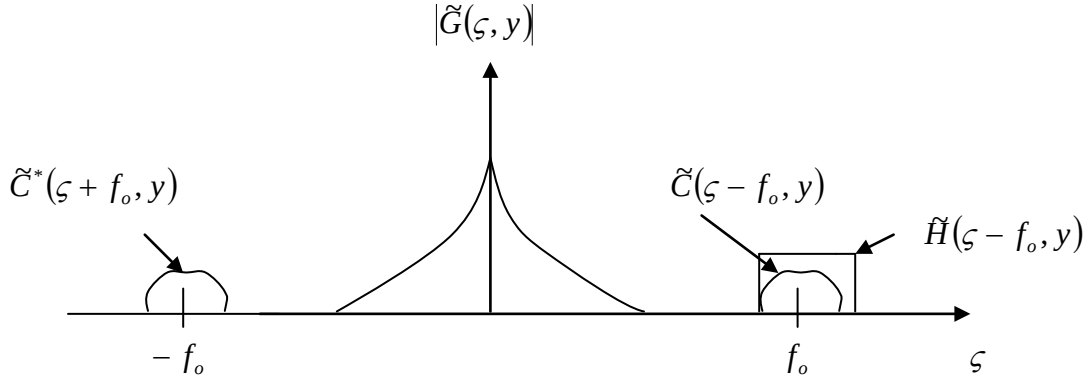


Figura C.1 Espectro del interferograma en la dirección del eje *x*.

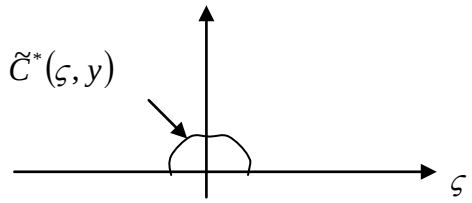


Figura C.2 Espectro de la señal filtrado y desplazado al origen.

Obtaining the inverse Fourier transform, we get $\tilde{c}(x, y)$

$$\phi(x, y) = arctan \frac{\mathrm{Im}\{\tilde{c}(x, y)\}}{\mathrm{Re}\{\tilde{c}(x, y)\}}$$

Referencias bibliográficas.

Carpio Martin , M. Servin, D. Malacara, "Direct phase detection of lateral shear interferograms using a phase-locked loop," *Opt. Comm.*, 225-229 (1994)
Malacara Daniel, M. Servin, Z. Malacara, *Interferogram analysis for optical testing*, Marcel Dekker, New York, 1998.
Servín M. and R. Rodríguez-Vera, "Two-dimensional Phase Locked Loop Demodulation of Interferogram," *J. Mod. Opt.*, **40**, 2087-2094 (1993).

Appendix C

C Algunas técnicas para obtención de la fase de un interferograma.

C.0 Synchronous and Asynchronous phase detection algorithms.

C.0.1 Synchronous phase detection algorithms.

C.0.2 Asynchronous phase detection algorithms.

C.1 Space domain phase detecting techniques (spatial synchrony of phase).

C.1.1 Three-step algorithm to measure the phase (phase shifting [step]).

C.1.2 Space phase demodulation with a linear carrier.

C.1.2.1 Quadrature phase detection of a sinusoidal signal (direct method).

C.1.2.2 Phase-Locked Loop (PLL).

C.2 Frequency domain phase detecting techniques.

C.2.1 Takedas's method (Fourier transform method)

Referencias bibliográficas.

Appendix D
Manual Básico del Lenguaje Java.
Una Introducción

Convenciones tipográficas.

`Este tipo de letra se usa para las palabras clave o reservadas en java.`

`Este tipo de letra se utiliza para escribirlo en la computadora, aunque se sustituye por lo que el usuario necesite.`

Cuando se utiliza subrayado es para indicar que hay un puntop importante para recordar.

Se utiliza para las palabras en inglés, y no se traducen porque son bien reconocidas en su lengua materna.

Curso Introductorio de Java.

Índice

El sitio *Web* de *Sun* es: http://www.sun.com/

El sitio Web de JavaSoft es: http://www.javasoft.com/

1. Introducción.

Java representa el futuro de los lenguajes de programación orientados a objetos.

En 1995 Java estuvo en versiones Alfa y Beta. Durante este tiempo Java estuvo limitado a ambientes Solaris y Widows NT. En 1996, Sun Microsystems liberó oficialmente Java 1.0, y se podían obtener kits para desarrolladores sin costo para cualquier sistema operativo importante incluyendo Solaris, AIX, Windows 95/NT, OS/2, y Macintosh.

Para un no programador, utilizar un lenguaje de programación como Java puede parecer imposible, pero nada esta más alejado de la verdad. Si es capaz de crear un documento HTML (*Hyper Text Markup Language*)(HTTP protocolo para recuperar y desplegar documentos HTML fácil y rápidamente), un documento SGML (*Standard Generalized Markup Language*), o un documento VRML (*Virtual Reality Modeling Language*) deberá ser capaz de crear poderosos documentos y aplicaciones Java.

La ventaja de Java es que aún los programas más básicos pueden tener características multimedia. Esta es la principal razón por la que los fundamentos de la programación del lenguaje Java son fáciles de aprender y utilizar.

1.1 ¿Qué es Java?

JavaSoft es una compañía de Sun Microsystems, dedico años desarrollando un lenguaje de programación muy poderoso para los 90's y más. Java cumple con la promesa de ser el lenguaje de programación más robusto, fácil de utilizar y versátil al momento. Incluye los mejores aspectos de los lenguajes C y C++, permite crear poderosas aplicaciones, tiene características multimedia internas (*built-in*), y deja fuera cosas de C y C++ como multiple herencia, sobrecarga de operadores y apuntadores.

Las mejores noticias a cerca de Java es que es orientado a objetos y de arquitectura neutral. La principal característica de los lenguajes orientados a objetos *Object-Oriented Programming* (OOP) es su capacidad de reusar código. Con Java, por otro lado, se pueden obtener los beneficios de reusar el código inmediatamente. Con Java se puede desarrollar una única aplicación que es inmediatamente utilizable en múltiples plataformas (Imagínese la cantidad de horas que se ahorrarán siendo capaz de desarrollar una única aplicación utilizable en sistemas Windows, UNIX, Macintosh.

La independencia de Java de la plataforma le permite desarrollar poderosas aplicaciones para sistemas operativos con los que nunca ha trabajado.

Los desarrolladores de Java pensaron muy cuidadosamente a cerca de la estructura de su nuevo lenguaje y no fue por mera coincidencia que modelaron al Java a partir del C y C++. C es el lenguaje favorito de los programadores que hacen programación procedural. C++ es el lenguaje favorito para programadores que escriben programas orientados a objetos. (Un buen número de programadores ya utiliza C y C++).

Para asegurar que Java sea fácil de entender y usar, Java es modelado a partir del C y C++. También tomó las extensiones de *Objective C*. Estas extensiones permiten la resolución de métodos dinámicamente (durante la corrida, *Runtime*).

Java se encarga de los apuntadores, maneja automáticamente la memoria, y también tiene una rutina de recolección de basura que corre en el fondo (en un ambiente multi-tasking un programa que corre en el fondo es otro programa que no tiene ventanas de despliegue, pero que sí usa recursos del sistema).

En ambientes distribuidos (que no es igual que sistemas operativos distribuidos), tal como la *World Wide Web*, estrictos mecanismos de seguridad son esenciales, Java fue desarrollado como el más seguro ambiente de programación encontrado en cualquier parte. Java no repara los "hoyos" de seguridad, los elimina, lo cual hace a Java el lenguaje perfecto para programación en el *Web*.

Los apuntadores son el más grande problema de seguridad de los programas C/C++. Los *crackers* pueden usar programas aparentemente benignos para falsificar apuntadores a memoria, robar información del sistema, y dañar el sistema permanentemente. En Java, en Java no se pueden falsificar apuntadores a memoria porque no hay apuntadores. Java elimina muchos de los otros agujeros de seguridad también. Por ejemplo, cuando los programas Java son compilados y corridos, ellos son comprobados con un verificador de código dinámico para asegurar que el programa no tiene codificación maliciosa. Además, el ambiente de corrida de Java (*Java Runtime Environment*, JRE) provee de estrictas reglas para programas comenzados desde *hosts* (anfitriónes) remotos. Estos programas no pueden acceder a la red local, no pueden acceder archivos en el sistema local y no pueden comenzar programas en el sistema local tampoco.

2. Historia.

2.1 La evolución de Java.

El lenguaje de programación Java es el resultado del esfuerzo concertado de un grupo de pensadores con visión. Pero lo que puede resultar sorprendente a cerca de Java es que este desarrollo tecnológico verdaderamente ingenioso no es el resultado de un proyecto que le fue bien desde el primer día. Sorprendentemente, si el proyecto que comenzó con el nombre Green en la primavera de 1991 hubiera procedido de acuerdo al plan, Sun Microsystems estaría en el negocio de la electrónica comercial y el mundo no hubiera conocido Java.

En 1991 Sun Microsystems, Inc. Estaba tomando la delantera en la producción de *workstations* UNIX. La compañía tuvo impuestos por USD 2.5 billones, contra solo USD 210 millones en 1986, estaba creciendo con un ritmo que parecía imparable. El crecimiento de Sun se debió a sus esfuerzos pioneros en los sistemas abiertos que posibilitaban a los negocios construir y mantener ambientes de computación de redes abiertas. Las corporaciones "corrieron a abrazar" los sistemas abiertos de Sun por que estaban hartos de los altos costos y los honorarios asociados con los ambientes de redes cerrados.

El proyecto Green nació como parte de un proyecto más grande para desarrollar software avanzado para el mercado de electrónica comercial, a fin de posicionar a Sun Microsystems en este mercado. Para lograr este objetivo los ingenieros y desarrolladores de Sun buscaron los microprocesadores que funcionaran en una variedad de máquinas, particularmente en sistemas distribuidos de tiempo real que fueran confiables y portátiles.

La clave del éxito de Sun en este mercado sería la fácil portabilidad del sistema a múltiples plataformas. El plan era desarrollar en proyecto en C++, pero, los desarrolladores tuvieron muchos problemas cuando intentaron extender el compilador C++. Tuvieron muchos problemas cuando intentaron trabajar en la estructura restrictiva de C++. James Gosling comenzó a trabajar en un nuevo lenguaje que él llamó Oak. Después el lenguaje tuvo que ser llamado Java porque Oak era una marca registrada de investigación.

El grupo Green tuvo una serie de errores. Con Time-Warner fallaron al intentar unas "cajas" que serían usadas para la televisión interactiva y vídeo por demanda. El lanzamiento al público del proyecto fue cancelado y mucha de la gente del equipo dejó el grupo Green.

Para 1994 hubo una migración masiva del National Center for Supercomputing Applications (NCSA) a Silicon Valley. Un grupo de la NCSA creó la compañía ahora llamada Netscape Communications Corporation. Silicon Valley fue inundada con pensamientos de ciberespacio y esa cosa llamada World Wide Web comenzó a esparcirse como fuego.

No fue una coincidencia que los desarrolladores del grupo Green pusieran sus ojos en el Internet y en la Web como la respuesta a sus problemas. El ambiente distribuido y de multiplataformas de Internet fue perfecto como "cama" de pruebas para su proyecto.

2.2 La evolución lógica de C a C++

El lenguaje de programación C ha sido muy popular desde que éste fue introducido a principios de 1970. La popularidad de C radica fuertemente en su relativa fácil utilización, amigabilidad, y características avanzadas, las cuales con el tiempo atrjeron a los programadores y a los no programadores.

Para disminuir los costos y el tiempo de desarrollo, los desarrolladores de software y los programadores buscaron formas de reusar el código existente en los proyectos futuros.

Aunque se pueden tomar segmentos de código C de un proyecto y usarlo en otro proyecto, la reutilización de código en C es dificil y no siempre es posible.

Un equipo creativo en los laboratorios Bell de AT&T desarrollaron una posible solución a los problemas de reutilización de código. Ellos llamaron a esta solución C++. C++ es una extensión del C que permite crear objetos. Los Objetos son módulos de código definidos separadamente que son utilizados para realizar funciones específicas. Bjarne Stroustrup fue la mente maestra quien tuvo la visión creativa de desarrollar esta extensión de C que permite a los programadores separar las funciones especificadas en una aplicación en módulos Objeto generalizados y reutilizables. Una década después de la introducción de C++, podemos ver claramente que éste como una evolución lógica de C.

2.3 El gran adelanto de C/C++ a Java.

Aunque el C++ es indudablemente un poderoso y popular lenguaje de programación, este tiene sus fallas. Estas fallas tienen sus raíces en su pasado al ser una extensión de C. Como C, el aspecto más difícil de entender y utilizar de C++ son el manejo de memoria y apuntadores. Además, debido a que C++ es una extensión de C, es muy fácil para los programadores de C++ mantener hábitos de codificación procedural que aprendieron en otros lenguajes, especialmente en C. Cuando se codifica una aplicación C++ usando métodos de programación procedural, se pierden los beneficios que los lenguajes de programación orientada a objetos ofrecen.

Java es el próximo progreso lógico para los lenguajes de programación y está fuertemente basado en el lenguaje de programación más popular de hoy. Java tiene el poder de los lenguajes diseñados específicamente para la programación orientada a objetos. A diferencia de C++ corta completamente las ataduras con la codificación procedural y obliga a seguir los conceptos de la programación orientada a objetos.

Las características del lenguaje de programación Java automatizó la comprobación de límites que eliminan muchos de las áreas problema de C y C++. Con Java, no hay apuntadores, y el manejo de memoria es automático. El beneficio obvio de esto es que los problemas con los apuntadores y "fugas" de memoria son cosas del pasado. Ya no se puede apuntar a una región de memoria equivocada y entonces "romper"(*crash the program*) el programa. Ya no puede haber errores cuando se está liberando memoria y causar una fuga de memoria baja que eventualmente causará que el sistema se quede sin memoria (*run out of memory*).

Java tiene muchas otras características que hacen que sea divertido programar en él y es un refugio para los programadores quienes están cansados de las dificultades asociadas con el desarrollo en C++. En Java los arreglos son encapsulados en una estructura de clase (*class*), el cual es otra característica que facilita la comprobación de límites internos de Java. Java realiza la recolección de basura automática, la cual es otra característica que facilita el manejo de memoria interno de Java.

La rutina de recolección de basura muestra la capacidad miltihilos (*multithreading*) interna de Java. Mediante la utilización de rutinas que corren en el fondo que liberan memoria y suministran una limpieza general de las funciones, Los programas Java corren más rápido y eficientemente.

3. Fundamentos del funcionamiento del lenguaje Java.

3.1 Java: Un lenguaje de programación compilado e interpretado.

Los programas Java se crean usando la estructura de un lenguaje de alto nivel. Cuando se termina de crear el programa Java, se compila el código fuente a un nivel intermedio llamado *bytecode*. Entonces se puede correr el *bytecode* compilado, el cual es interpretado por el ambiente de corrida de *Java (Java Runtime Environment*).

El *bytecode* es muy diferente del código máquina. El código máquina está representado por una serie de 0's y 1's. Los *bytecode* son conjunto de instrucciones que se ven como código ensamblador. Sin embargo la computadora sólo puede ejecutar código máquina, los *bytecodes* deben ser interpretados antes de que puedan ser ejecutados. El compromiso entre el código máquina y el *bytecode* es muy profundo. El código máquina es utilizable solo en una plataforma específica para la cual éste fue compilado. Por otro lado, el programa en *bytecode* puede correr en cualquier plataforma que sea capaz de usar el ambiente de corrida de Java (JRE). Esta capacidad es la que hace de Java una arquitectura neutral.

El intérprete de corrida de Java traduce el *bytecode* en un conjunto de instrucciones que la computadora puede entender. Debido a que el *bytecode* es una forma intermedia, sólo hay un ligero retardo en traducir éste a una forma que la computadora entienda.

Este es un lenguaje que permite compilar el código fuente a una forma intermedia e independiente de la máquina en que se ejecutará, cercanamente tan rápido como si este fuera completamente compilado.

El truco de crear el Java *bytecode* es que el código fuente es compilado para una máquina que no existe. Esta máquina es llamada Java Virtual Machine y esta sólo existe en el nanoespacio, o confinada en la memoria de la computadora. Tan sorprendente es crear una máquina en la memoria de la computadora, como que Java es la prueba viviente de que este concepto no solo es posible, sino poderoso.

Engañar al compilador de Java para que cree el *bytecode* para una máquina no existente es solo la mitad del ingenioso proceso que hace de Java una arquitectura neutral. El interprete de Java debe también hacer que la computadora y el archivo de *bytecode* crean que están corriendo sobre una *Java Virtual Machine*. Esto lo hace posible actuando como un intermediario entre la máquina virtual y la computadora real.

Las capas de interacción para los programas Java son:

<table>
<tr><td colspan="3">Aplicaciones del Usuario</td></tr>
<tr><td colspan="3">Objetos Java</td></tr>
<tr><td colspan="3">Máquina virtual de Java (Java Virtual Machine)</td></tr>
<tr><td>UNIX</td><td>Windows OS/2</td><td>Macintosh</td></tr>
<tr><td colspan="3">Sistemas Operativos.</td></tr>
</table>

4. Principales características de Java.

Java es:

- Arquitectura neutral
- Distribuido
- Dinámico
- Interpretado y compilado
- Multihilos
- Listo para red y con compatibilidad
- Orientado a objetos
- Transferible
- Robusto
- Seguro

Estas características están interrelacionadas y permiten el desarrollo de los siguientes puntos:

- Ambientes de redes distribuidos
- Alto desempeño (buen funcionamiento, buena velocidad)
- Fácil reutilización de código.
- Seguridad.

4.1 Diseñado para ambientes de redes distribuidas.

Un ingrediente clave para el éxito del lenguaje Java diseñado para un ambiente tan complejo como el Internet es la capacidad de correr en plataformas heterogeneas y distribuidas. Java es capaz de lograrlo debido a que:

- Es de arquitectura neutral, esto significa que los ejecutables de Java pueden correr sobre cualquier plataforma que tenga (en el que exista) el ambiente de corrida de Java (JRE).
- Altamente transferible, significa que Java tiene características que lo hacen fácil de utilizar en plataformas heterogéneas. Para sustentar la transferibilidad, Java se sujeta al estándar IEEE (*Intitute of Electrical*

and Electronics Engineers) para estructuras de datos, tales como el uso de enteros, números de punto flotante, y cadenas.

- Distribuido, significa que permite utilizar los objetos localizados en máquinas locales y remotas.
- Listo para red y con compatibilidad, significa que esta listo para ser usado en redes complejas y características de soporte directo para protocolos de redes comunes, tales como FTP y HTTP, que lo hacen compatible para usarse en redes.

4.2 Diseñado para alto desempeño.

Otro ingrediente clave para el éxito de Java diseñado para ambientes de redes complejos es su desempeño. Java tiene muchas características que lo hacen un lenguaje de alto desempeño, incluyendo a su compilador y a su sistema de corrida (*runtime*). El intérprete de Java es capaz de ejecutar el *bytecode* a velocidades que aproximan a las de un código compilado en un formato leíble por la máquina porque es multihilos.

A diferencia de muchos otros lenguajes de programación, Java tiene internamente capacidades multihilos. Multihilos es la capacidad de correr más de un proceso hilo a la vez. Los multihilos permiten al interprete Java correr procesos tales como el recolector de basura y el controlador de memoria en el "fondo". Los procesos que corren en el "fondo" hacen uso del tiempo "*idle*" del CPU de la computadora. Los programas que son multihilados son priorizados (se les establecen prioridades). Cuando la computadora esta esperando una entrada del usuario, el proceso en el "fondo" puede estar muy ocupado limpiando la memoria, o cuando la computadora se "detiene" por un instante después de separar un número, un proceso de "fondo" puede lograr entrar en unos cuantos ciclos de reloj para preparar una sección de la memoria a utilizar. El ser capaz de correr procesos en el "fondo" mientras un proceso en el "frente" espera por una entrada del usuario o se detiene momentáneamente, es fundamental para un desempeño óptimo.

4.3 Diseñado para fácil reutilización del código.

Aunque el diseño de Java orientado a objetos es la característica clave que hace posible la reutilización de código, los diseñadores de Java sabían que los ambientes de redes complejos tienden a cambiar rápidamente. Por lo tanto, para asegurar que los desarrolladores de Java pudieran fácilmente reusar código aún si el ambiente cambió, Java fue diseñado para ser dinámico y robusto. Java lo logra retardando el "encuadre"(*binding*) de objetos, así como a través del enlace dinámico de clases al tiempo de corrida, lo cual evita errores si el ambiente ha cambiado desde que el programa fue compilado. Otra forma en que Java evita los errores es que comprueba las estructuras de datos durante la compilación y al tiempo de corrida.

Robustez y confiabilidad van de la mano. Para que Java sea robusto debe ser confiable también. Como se sabe, los apuntadores y el manejo de memoria son la fuente de muchos problemas en programas de C/C++. Por lo tanto Java asegura robustez y

confiabilidad al no usar punteros; utilizando comprobación de límites dinámico, lo cual asegura que los límites de la memoria no pueden ser violados; y manejando la memoria automáticamente, lo cual previene contra "fugas" y violaciones de memoria.

4.4 Diseñado para ser seguro.

La seguridad en ambientes de redes distribuidos es esencial, especialmente en un mundo automatizado donde los virus de computadora, caballos de Troya y gusanos abundan. Muchas de las características Java aseguran que es extremadamente seguro. Por ejemplo, los programadores maliciosos no pueden acceder a la "*heap*", "*stack*" del sistema o secciones protegidas de memoria porque Java no usa apuntadores a memoria y sólo localiza memoria durante el tiempo de corrida. Esto evita que los programadores maliciosos alcancen secciones restringidas del sistema.

Java también asegura la seguridad del sistema durante el tiempo de corrida haciendo que el interprete Java cumpla con un doble deber como verificador de *bytecode*. Antes de que el interprete Java ejecute un programa, éste realiza una comprobación para asegurarse de que programa es un código de Java válido. Java prueba la validez del código usando un probador de teoremas que previenen contra lo siguiente:

- Violaciones de accesos
- Punteros perdidos
- Conversiones ilegales de datos
- Incorrectos valores y parámetros
- Alteración maliciosa o mal uso de clases
- *Overflow* y *underflow* del *stack*
- Actividad sospechosa o no garantizada

5. Conceptos de programación orientada a objetos y Java.

Los conceptos relacionados a la programación orientada a objetos son:

- Objetos
- Encapsulamiento y paso de mensajes.
- Clases
  - Declaración de clases
    - Clase
    - Clase abstracta
    - Interface
- Librerías (paquetes en Java)
  - Espacio de nombres
  - La sentencia `package`
  - Declaración de un paquete
  - La sentencia `import`

- Herencia
- Polimorfismo
- Modificadores de Acceso

5.1 Objetos.

La unidad fundamental en la programación orientada a objetos es el objeto. Todos los objetos tienen propiedades (o estados) y comportamientos.

Los elementos de datos y sus valores asociados pertenecen a las propiedades (o estados) de un objeto. Cada cosa que un objeto sabe de estos elementos y sus valores describen las propiedades (o estados) del objeto (p. Ej. para un auto: cuantas ruedas tiene, cuantos pasajeros, color, carga, y más datos que conozca). Los elementos de datos asociados con los objetos son llamados variables instancia.

El comportamiento de un objeto de pende de las acciones que el objeto pueda realizar sobre las variables instancia definidas dentro del objeto. En programación procedural tales construcciones son llamadas funciones. En terminología orientada a objetos, esta construcción es llamada método. Un método pertenece a la clase de la que este es miembro.

Así el estado de un objeto depende de las cosas que el objeto conoce, y el comportamiento del objeto de las acciones que el objeto puede realizar.

Se puede pensar en un objeto como un bloque de código que realiza un conjunto de acciones específicas generalmente relacionadas llamadas métodos.

Cada objeto es una instancia de una clase. Los objetos son creados por instanciación a una clase con el operador `new`. Sólo es posible hacer instancias de clases que no sean abstractas, ni tampoco de interfaces.

En el siguiente ejemplo se crean dos objetos pluma, la primera de color azul tiene como número de identificación un 1, la segunda de color rojo tiene como número de identificación un 2.
[Véase la declaración de la clase `pluma` en la sección: **Clases.**]

```
pluma plumaAzul = new pluma(1,"azul");
pluma plumaRoja = new pluma(2,"roja");
```

5.1.1 Encapsulamiento y paso de mensajes.

Los objetos encapsulan variables instancia y métodos relacionados en una sola unidad identificable. Por lo tanto, los objetos son fáciles de reusar, actualizar y mantener. Se puede fácil y rápidamente:

- Determinar las entradas necesarias al objeto y la salida del objeto.
- Encontrar dependencia de variables.
- Aislar los efectos de los cambios.
- Hacer actualizaciones cuando sea necesario.
- Crear subclases basadas en el objeto original.

Un objeto puede invocar a uno o más métodos para realizar una tarea. Se inicia un método pasando un mensaje al objeto. Un mensaje debe contener el nombre del objeto al que se esta enviando el mensaje, los nombre de los métodos a efectuar, y los valores necesitados por esos métodos. Los objetos que reciben el mensaje usan esta información para invocar los métodos apropiados con los valores especificados.

El beneficio del encapsulamento de las variables instancia y de los métodos es que se pueden enviar mensajes a cualquier objeto sin saber como trabaja el objeto. Todo lo que se necesita saber es que valores aceptará el método.

5.2 Clases.

La clase es el principal bloque de construcción de un programa Java que encapsula los objetos con atributos y comportamientos definidos y que puede heredar variables y métodos de otras clases.

Las clases encapsulan objetos. Una sola clase puede ser usada para instanciar multiples objetos. Esto significa que se pueden tener muchos objetos activos o instancias de una clase.

Recuerde que cada objeto de una clase mantiene sus propios estados (o propiedades) y comportamientos. Mediante el encapsulamento de objetos en una estructura clase, se pueden agrupar conjuntos de objetos por tipo.

El Java API (*Application Programing Interface*) (Interface de programación de aplicaciones de Java) especifica un conjunto de objetos que realizan funciones relacionadas y comparten características comunes.

La definición de una clase especifica los atributos y comportamientos de los objetos que pertenecen a esa clase. Los atributos corresponden a las variables instancia de la clase. Los comportamientos corresponden a los métodos de la clase. Los valores en las variables de instancia de un objeto de una clase particular pueden diferir de los de otro objeto de esa clase, pero ambos comparten los mismos métodos.

5.2.1 Declaraciones de clases.

En Java, todas las clases son derivadas de la clase del sistema llamada `Object`. Esto hace a `Object` la raíz en la jerarquía de clases y significa que todos los métodos y variables en la clase `Object` están disponibles a todas las otras clases. Es esta estructura de clases la que hace posible la reutilización del código en Java. Todas las clases son por dafault privadas a menos que se declaren a ser públicas.

La definición (declaración) de una clase no puede ser dividida a través de varios archivos fuente (Los métodos nativos son una excepción a esta regla).

La declaración general de una clase se realiza mediante:

```
[Modificador de Acceso] class nombre_clase [extends clase_padre]{
    // variables y métodos de la clase
  }
```

Veamos un ejemplo de la implementación de una clase muy sencilla llamada pluma:

```
class pluma{
 int nId;
 String strColor;

 public pluma(int n , String str){  //observese que no tiene valor de retorno // y no lleva void porque es el constructor de la clase, y los constructores no // tienen valor de retorno
   nId = n;
   strColor = str;
 }

 public String obtenColor(){
  return strColor;
 }

 public void hazImpresión(){
  System.out.println("pluma seleccionada" + strColor);
 }
}
```

Una clase que contiene cualquier método abstracto es llamada abstracta, y debe ser declarada como tal. Las clases abstractas no pueden ser construidas con `new` (no se puede crear una instancia). Sin embargo, pueden declararse variables basadas en clases abstractas y pueden ser referidas como instancias válidas de sus subclases.

```
[Modificador de Acceso] abstract class nombre_clase [extends clase_padre]{
    // variables de la clase
    // [Modificador de Acceso] abstract [valor_retorno] nombre_método([parámetros]);
    // y métodos no abstractos de la clase
  }
```

Java soporta herencia simple de clases. Cada clase excepto `Object` tiene sólo una superclase. Esto significa que cualquier clase que se cree puede extender o heredar las funciones de una sola clase. Sin embargo esto puede parecer una limitación si se ha programado en un lenguaje que permite herencia múltiple de clases, Java soporta herencia múltiple de los métodos de clases, lo cual se logra a través de la clase `interface`.

5.2.2 ¿Qué es una `interface`?

Es una clase donde todos los métodos son abstractos (no implementa sus métodos y deja su implementación a otras clases). Las interfaces pueden a sí mismo descender de otras interfaces pero no de otras clases.

Son útiles para generar relaciones entre clases que de otro modo no están relacionadas.

5.2.3 Declaración de interfaces.

Las interfaces son clases abstractas. A través de una `interface` se pueden definir los protocolos para los métodos y variables finales sin tener que preocuparse por la implementación específica. Esto permite crear lo que podríamos llamar un esbozo para las estructuras de programación que después se definirán. Mientras que las interfaces son por default privadas a menos que sean declaradas públicas, los métodos y las constantes de la `interface` son públicos(as).

La declaración de la `interface` tiene el formato general:

```
[Modificador de Acceso] [abstract] interface nombre_interface [extends interface_padre]{
    // métodos abstractos y
    // variables estáticas asociadas con la interface
  }
```

Todos los métodos abstractos en la `interface` se declaran como:

`[Modificador de Acceso] abstract` [valor_retorno]|void `nombre_método([parámetros]);`

Las declaraciones de clases que usan (implementan) interface(s) tienen el formato general:

`[Modificador de Acceso] class nombre_clase implements nombre_interface1[,nombre_interface2[,nombre_interface3[,...]]]` { // cuerpo de la clase }

Las declaraciones de las clases que implementan interface(s) deben implementar todos los métodos de la(s) interface(s), si sólo implementan algunos métodos deben declararse como clases abstractas. Esto es:

`[Modificador de Acceso] abstract class nombre_clase implements nombre_interface1[,nombre_interface2[,nombre_interface3[,...]]]` { // cuerpo de la clase }

Las interfaces no pueden ser construidas con `new` (no se puede crear una instancia). Sin embargo, pueden declararse variables basadas en interfaces y pueden ser referidas como instancias válidas de sus subclases.

5.3 Herencia.

La herencia es un poderoso aspecto de la programación orientada a objetos que permite fácilmente reusar el código y extender la funcionalidad de las clases existentes.

Un programador puede definir un nuevo objeto basado en la definición de un objeto existente. El nuevo objeto hereda todos los atributos y comportamientos del objeto original. Pueden añadirse atributos y comportamientos adicionales, pero no pueden borrase de la nueva definición del objeto. Se dice que los nuevos objetos son "derivados" del objeto original. La nueva clase hereda todas las variables instancia y los métodos de la clase original. La operación de cualquiera de los métodos originales puede ser cambiada simplemente redefiniendo ése método en la nueva clase. Métodos y variables instancia pueden ser añadidos a la nueva clase. Las variables instancia y los métodos de la clase original no pueden ser borrados de la nueva clase, debido a que la nueva clase debe mostrar la misma funcionalidad que estuvo presente en la clase original.

5.4 Polimorfismo.

Ya que el programador no puede borrar los atributos y comportamientos cuando está extendiendo un objeto, un nuevo objeto derivado puede ser utilizado como si este fuera el objeto original; ninguna de la funcionalidad necesaria es perdida (sin embargo esta pudo

haber sido redefinida). Un procedimiento que invoca a un comportamiento (método) del objeto original también puede invocar al mismo comportamiento del nuevo objeto definido sin ninguna modificación.

Como ejemplo supóngase un procedimiento que simula una máquina que envía a desayunar (máquina virtual), esta máquina toma un grupo de objetos empleado (u objetos derivados) y los envía a desayunar invocando a su método irADesayunar. Si el objeto vicepresidente es añadido a este grupo, la máquina envía a desayunar mandará cada objeto a la localización correcta para desayunar. Los objetos empleado a la cafetería y el objeto vice-presidente al restaurante. La máquina enviar a desayunar trabaja correctamente sin saber nada más a cerca de los objetos en el grupo mas que ellos son objetos empleado (o una clase derivada del objeto empleado) y que tienen un método llamado irADesayunar (al cual la máquina invoca).

El objeto vice-presidente puede ser utilizado en cualquier parte donde el objeto empleado sea válido. Sin embargo, lo inverso no es cierto. El objeto original no puede ser utilizado en cualquier parte donde el nuevo objeto es requerido.

6. Paquetes (librerías).

En C++ y otros lenguajes de programación, una colección de clases o funciones relacionadas es llamada una librería. Java da un giro en el concepto de librerías usando el término paquete para describir una colección de clases relacionadas. Al igual que las clases encapsulan objetos, los paquetes encapsulan clases en Java. Esta "capa" adicional de encapsulamento hace más fácil controlar el acceso a los métodos.

Los paquetes agrupan librerías de clases. Un paquete es la más grande unidad lógica de objetos en Java.

Los paquetes en Java agrupan una variedad de clases y/o interfaces juntas. En paquetes, las clases pueden ser únicas comparadas con las clases en otros paquetes. Los paquetes también proveen un método de controlar la seguridad en los accesos. Finalmente, los paquetes proveen una forma de "ocultar" las clases, evitando que otros paquetes o programas accedan a las clases que son para uso interno de una aplicación solamente.

6.1 Declaración de un paquete.

Los paquetes son declarados usando la palabra clave `package` seguido por un nombre de paquete. Esto debe ocurrir como la primera sentencia en un archivo fuente Java, excluyendo los comentarios y espacios en blanco. Aquí esta un ejemplo:

```
package mamíferos;
class Unmamífero{
 ... cuerpo de la clase Unmamífero
}
```

En este ejemplo, el nombre del paquete es `mamíferos`. La clase `Unmamífero` es considerada como parte del paquete. Incluir otras clases en el paquete `mamíferos` es fácil: Simplemente coloque una línea `package` idéntica al principio de sus archivos fuentes también. Debido a que cada clase generalmente es colocada en su propio archivo fuente, cada archivo fuente que contiene clases para un paquete particular debe incluir esta línea. Sólo puede haber una sentencia `package` en cualquier archivo fuente.

Obsérvese que el compilador Java sólo requiere que las clases que sean declaradas públicas sean puestas en un archivo fuente separado. Las clases no públicas pueden ser puestas en el mismo archivo fuente. Sin embargo es buena práctica de programación poner cada clase en su propio archivo fuente, una sentencia `package` al principio del archivo se aplicará a todas las clases declaradas en ese archivo.

Java soporta el concepto de jerarquía de paquetes. Esto es similar a la jerarquía de directorios encontrada en muchos sistemas operativos. Esto se logra especificando múltiples nombres en una sentencia `package`, separados por un punto. En el siguiente código, la clase `Unmamífero` pertenece al paquete `mamífero` que está dentro del paquete `animal` en la jerarquía de paquetes:

```
package animal.mamífero;
class Unmamífero{
  ... cuerpo de la clase Unmamífero
}
```

Esto permite relacionar las clases relacionadas dentro de un paquete y entonces agrupar los paquetes relacionados dentro de un paquete más grande. Para referirse al miembro de otro paquete, el nombre del paquete es añadido al frente al nombre de la clase. Este es un ejemplo de la llamada a un método `preguntaNombre` en la clase `Ungato` en el subpaquete `mamífero` en el paquete `animal`:

```
animal.mamífero.Ungato.preguntaNombre();
```

La analogía con la jerarquía de directorios es reforzada por el interprete Java

6.2 La sentencia `package`.

El API de Java incluye un grupo de paquetes bajo el paquete núcleo `java`. Todas las clases y objetos que se crean se asume que están bajo este paquete por default. Por lo tanto, a menos que se indique otra cosa, Java utiliza el *path* actual y generalmente supone que el código compilado esta en el directorio actual cuando se corre un programa Java. Para decir a Java donde buscar los paquetes se usa la sentencia `package`. Se usa al principio del archivo fuente:

```
package Nombre_de_paquete;
```

Cuando se están definiendo nuevos paquetes, se debe seguir la estructura de nombrado jerárquico. Esto significa que los subniveles en la estructura de nombrado de paquetes deberá ser separada de los nombres de paquetes en otros subniveles por un punto. A continuación se muestra un ejemplo de la sentencia `package` para un espacio de nombre de paquetes multinivel:

```
package Mi_paquete.Nombre_del_subpaquete;
```

Los nombres de paquete en cada subnivel del espacio de nombres debe reflejar la estructura del directorio en el sistema de archivos. Esto es debido a que Java transforma los nombres de paquetes en nombres de ruta (*path*) para localizar las clases y métodos asociados con un paquete. Esto es logrado reemplazando los puntos con el separador del sistema de archivos. Así el nombre de paquete `Mi_paquete.Nombre_del_subpaquete` será transformado en el siguiente nombre de ruta para un sistema operativo UNIX:

```
Mi_paquete/Nombre_del_subpaquete
```

En el sistema operativo Windows 95/NT, el nombre de paquete `Mi_paquete.Nombre_del_subpaquete` será transformado en el siguiente nombre de ruta:

```
Mi_paquete\Nombre_del_subpaquete
```

Java encuentra las clases asociadas con un paquete usando el nombre de ruta. Java buscará por el paquete `Mi_paquete.Nombre_del_subpaquete` y sus clases y métodos asociados en el directorio `\Mi_paquete\Nombre_del_subpaquete`.

Las especificaciones del lenguaje Java recomiendan que se utilice un nombre de paquete único y global que pertenezca a su dominio de Internet cuando los paquetes estarán puestos a la disposición de usuarios no locales. Además es recomendado que se utilice mayúsculas para el primer componente del dominio principal, tales como COM, EDU o GOV.

Recuérdese que el orden de precedencia en Java, es el orden inverso del nombre del dominio, componente por componente. Así un paquete disponible globalmente llamado `equix` asociado con `compañia.com` deberá ser llamado `COM.compañía.equix`. Este nombre de paquete es transformado en el siguiente nombre de ruta para el sistema operativo UNIX:

```
COM/compañía/equix
```

En el sistema operativo Windows 95/NT, el nombre de paquete `COM.compañía.equix` es transformado en el siguiente nombre de ruta

```
COM\comapñia\equix
```

6.3 La sentencia `import`.

Como en otros lenguajes, Java incluye un núcleo de funcionalidad que es accesible globalmente. En Java estas funciones del "núcleo" están en el paquete `java.lang`. Para acceder a los paquetes, clases, y objetos que no están declarados en este paquete, se utiliza la sentencia `import`. La sentencia `import` ayuda a definir y resolver el espacio de nombres actual.

Si no se utiliza la sentencia `import`, se debe especificar la referencia completa del paquete antes del nombre de cada clase en el código fuente. Teniendo que especificar la referencia completa del paquete puede resultar tedioso, espacialmente en programas grandes.

La sentencia `import` deberá aparecer antes de cualquier otra declaración en el código fuente y generalmente seguido por la sentencia `package` si es que es utilizada. La idea detrás de la sentencia import es ayudar a Java a encontrar los métodos apropiados y evitar conflictos en el espacio de nombres. Para poder acceder a la clase AWT, se deberá utilizar la sentencia `import` en el código fuente:

```
import java.awt;
```

Java permite importar sólo la clase que se necesite:

```
import java.awt.Button;
```

Una forma más eficiente de importar paquetes es importarlos por demanda:

```
import java.awt.*;
```

Un asterísco en el último elemento de una sentencia `import` permite a Java importar clases como sean necesarias. Esto permite añadir tipos públicos al espacio de nombres dinámicamente.

La mayoría de los programadores en Java importan clases por demanda. Esto ahorra mucha escritura, sin embargo se deberá ser tan específico como sea posible acerca de los paquetes que se necesitan. Específicamente, raramente se utilizará una sentencia `import` que haga que todas las clases en Java estén disponibles. Cuando dos clases en dos diferentes paquetes tienen el mismo nombre, existirá un conflicto a menos que se añada el nombre del paquete al principio del nombre de la clase.

7. Espacio de nombres.

Mientras más complejo sea el programa que se cree, más fácil es reutilizar un nombre de variable o parámetro. En C++, un conflicto en el espacio de nombres puede tomar horas o días repararlo, especialmente cuando se tienen librerías de clases grandes. Java cuenta con un sistema de nombre único que evita conflictos en el espacio de nombres por medio de la "anidación" de los nombres en varios niveles.

Cada componente de un nombre es anidado de acuerdo a los siguientes niveles de espacio de nombres:

- 0 - Paquetes del espacio de nombres
- 1 - Unidad de compilación del espacio de nombres
- 2 - Tipo del espacio de nombres
- 3 - Método del espacio de nombres
- 4 - Bloque local del espacio de nombres
- 5 - Bloque local anidado del espacio de nombres

El interprete de Java es el responsable de mantener y traducir el espacio de nombres. El interprete resuelve los nombres de objetos buscando sucesivamente en cada uno de los niveles del espacio de nombres conocidos. Cada nivel del espacio de nombres es considerado en orden de precedencia desde el más alto nivel, el paquete del espacio de nombres, al nivel más bajo, el bloque local anidado del espacio de nombres. Java generalmente utiliza el primer "casamiento" encontrado.

Los niveles en el espacio de nombres representa una jerarquía que refleja la estructura asociada con los paquetes y clases Java. Los nombres asociados con cada nivel son separados de los nombres en otros niveles por un punto. El concepto de espacio de nombre y niveles en el espacio de nombres tiene más sentido cuando se utilizan.

Los paquetes Java en la librería de paquetes original tienen el nombre de paquete `java` añadido al frente y son generalmente seguidos por el nombre de la clase en el paquete. A continuación se muestra la declaración para la clase `BorderLayout` en el paquete *Abstract Windowing Toolkit* (AWT):

```
java.awt.BorderLayout
```

El nombrado de las llamadas a funciones en los programas Java utilizan la convención del espacio de nombres también. La siguiente es una declaración para un método llamado `println`:

```
java.lang.System.out.println
```

Java resuelve la localización del método `println` buscando la única ocurrencia de la clase `System` en el espacio de nombres definido. Cuando Java encuentra una ocurrencia

de la clase `System`, busca por una ocurrencia de la subclase `out`, y finalmente busca el método `println`.

8. La estructura del lenguaje Java.

8.1 Bases.

Usa el conjunto de caracteres UNICODE, el cual es un super conjunto del ASCII. UNICODE usa 16 bits para representar caracteres (ASCII usa 8 bits).

8.2 Applets contra Aplicaciones.

El término Applet Java generalmente se refiere a una pequeña aplicación que esta diseñada para ser utilizada en la *Wolrd Wide Web*. Los applets requieren un programa visor externo, a fin de poder usar el applet se requiere de un *browser Web* o un *applet viewer*. Recuerde que lo pequeño de un applet es completamente relativo y que la clave para que sea un applet es que esta diseñado para usarse con un visor externo.

El término aplicación Java generalmente se refiere a una aplicación que esta diseñada para usarse sola. Un termino usado frecuentemente en la jerga para las aplicaciones es *app*. Las aplicaciones no requieren de un visor externo. Esto significa que se puede ejecutar una aplicación Java directamente usando el intérprete Java. Recuérdese que algunos programas Java pueden correr como programas applet que requieren de un visor externo y como aplicaciones que corren solas.

8.3 Estructuras para el nombrado de los programas.

El archivo de programa que contiene las instrucciones en el lenguaje de programación Java es llamado fuente o código fuente. Se nombran los archivos de código fuente para Java con la extensión `.java`. Estos archivos deberán ser archivos de texto plano. Cuando se compilan los archivos fuente usando el compilador Java, `javac`, se compila el código fuente a un *bytecode* que puede ser ejecutado.

En un lenguaje procedural tal como Pascal, el código ejecutable completo para un programa esta generalmente almacenado en un archivo. En un lenguaje orientado a objetos tal como Java, el código ejecutable está generalmente almacenado de acuerdo a la estructura de la clase o librería. Después de que se compila el código fuente Java, se tiene un archivo para cada clase que se declaró en el archivo fuente. Cada uno de estos archivos será llamado con la extensión `.class`. Estos archivos individuales son llamados unidades de compilación.

Las unidades de compilación contienen sentencias `package` (sentencias que le indican al compilador a cual paquete corresponde esta unidad de compilación) y sentencias

`import` (sentencias que llaman a otros paquetes y unidades de compilación) y declaraciones para clases e interfaces. Estos cuatro componentes forman la estructura básica de los programas Java.

8.4 Comentarios.

```
/* Este comentario es como en C */
// Este comentario es como en C++
/** Este es un comentario para generación automática de documentación en Java */
```

8.5 Separadores.

Java utiliza los siguientes separadores `( )` , `{ }` , `[ ]` , `;` (punto y coma), `,` (coma) y `.` (punto).

El compilador utiliza estos separadores para dividir el código en segmentos. También pueden utilizarse para forzar precedencia de evaluaciones aritméticas en una expresión. Los separadores son útiles también como localizadores visuales y lógicos para los programadores.

8.6 Palabras clave reservadas en Java ver. 1.0:

Palabras clave para la declaración de datos
boolean
byte
char
double
float
int
long
short
Palabras clave para ciclos
break
continue
do
for
while
Palabras clave condicionales
case
else
if
switch

Palabras clave para excepciones
catch
finally
throw
try
Palabras clave para estructuras
abstract
class
default
extends
implements
instance of
interface
Palabras clave de modificadores y acceso
final
native
new
private
protected
public
static
synchronized
threadsafe
transient
void
Palabras clave misceláneas
false
import
null
package
return
super
this
true

Las siguientes palabras clave son reservadas y no se utilizan en Java ver. 1.0; serán utilizadas en liberaciones futuras:

byvalue
cast
const
future
generic

goto
inner
operator
outer
rest
var

8.7 Literales.

Las literales en Java están basadas sobre representaciones de caracteres y números. Los tipos de literales son enteras, de punto flotante, lógicas (*boolean*), carácter y cadena (*string*).

Cada variable consiste de una literal y un tipo de dato. La diferencia entre las dos (literales y tipos de datos) es que las literales son introducidas explícitamente en el código. Los tipos de datos son información acerca de las literales, tal como cuanto espacio será reservado en memoria para esa variable, así como también los posibles rangos de valores para la variable.

8.7.1 Literales enteras.

Los enteros en Java pueden ser expresados en decimal, hexadecimal u octal y son distinguidos por sus caracteres de comienzo.

Las literales decimales comienzan con un dígito diferente de cero (1 - 9).
Los números hexadecimales comienzan con 0x ó 0X seguido de (0 - 9, a - f, A - F).
Las literales octales comienzan con un 0 (cero) seguido de (0 - 7)

Pueden ser precedidos por "-" para indicar valores negativos, técnicamente no es parte de la literal pero es aplicado como operador unitario. Un sufijo "L" (mayúscula o minúscula) puede añadirse para indicar que el valor es un `long` de otra forma es tomado como un `int`.

Un error de compilación se generará, si la literal entera excede los rangos máximo o mínimo según si el tipo es `int` o `long`.

8.7.2 Literales de punto flotante.

Una literal de punto flotante representa un número que tiene un punto decimal en él. Los estándares de Java especifican que los números de punto flotante deben seguir las especificaciones de los estándar industriales indicados en IEEE-754

El punto flotante se compone de: el número entero, un punto decimal, una parte fraccional, un exponente y un sufijo de tipo.

El exponente es indicado por la letra e ó E

Una literal de punto flotante es considerada `double` (doble precisión) a menos que el sufijo f ó F sea añadido, en este caso la literal es de tipo `float` (simple precisión).

Un error de compilación se generará, si la literal de punto flotante excede los rangos máximo o mínimo según si el tipo es de simple precisión o doble precisión.

Ejemplos:

1.123 (literal de tipo `double`)
4.578e+47 (literal de tipo `double`)
1.123f (literal de tipo `float`)
5.678e6f (literal de tipo `float`)

8.7.3 Literales `boolean`.

Hay dos literales `boolean`: `true` y `false`. Ya que el tipo boolean es un tipo definido internamente en Java estas dos literales no corresponden a un valor aritmético en particular (p.ej. cero y uno). Aunque la sentencia `y?1:0` convertirá la variable `boolean` y a `0` para falso y a `1` para true.

8.7.4 Literales carácter.

Una literal carácter es un carácter (o secuencia de escape) entre comillas sencillas.

Descripción o secuencia de escape	Secuencia	Salida
Cualquier carácter	'c'	C
Espacio atrás *Backspace* (BS)	'\b'	Espacio atrás
Tabulador horizontal *Horizontal tab* (HT)	'\t'	Tabulador
Alimenta línea *Linefeed* (LF)	'\n'	Alimenta linea
Alimenta hoja *Formfeed* (FF)	'\f'	Alimenta hoja
Retorno de carro *Carriage return* (CR)	'\r'	Retorno de carro
Comilla doble	'\" '	"
Comilla sencilla	'\' '	'
Diagonal invertida Backslash	'\\'	\
Patrón de bits Octal	'\ddd'	Carácter del valor octal dddH
Patrón de bits Hexadecimal	'\xdd'	Carácter del valor hexadecimal dd
Carácter UNICODE	'\udddd'	El carácter UNICODE de

		ddddHH

H donde ddd son dígitos octales (se pueden poner uno, dos o tres, y no es necesario el cero al principio).
HH donde dddd son dígitos hexadecimales.

8.7.5 Literales cadena.

Las literales cadena (*string*) son cualquier número de caracteres encerrados entre comillas dobles. Se pueden concatenar cadenas con el operador binario +. Una literal cadena no puede dividirse en más de una línea; esto se puede lograr separándola la literal cadena durante la declaración y luego concatenándola durante la corrida.

8.8 Utilización de los tipos de datos.

Una variable es algo que cambia, o varía. En Java, una variable almacena datos. Los tipos de datos definen el tipo de dato que puede ser almacenado en una variable y los límites de los datos.

Un ejemplo del uso de tipos de datos es:

```
char mi_letra;
```

Este ejemplo demuestra las dos partes esenciales de una variable: el tipo (`char`) y el identificador (`mi_letra`).

Hay dos tipos principales de datos en Java: tipos por referencia y tipos primitivos.
Los tipos de datos pueden ser almacenados en variables, pasados como argumentos, regresados como valores, y operados.

Nota: La declaración (o definición) de una variable no asigna un valor a la variable; esta simplemente identifica los posibles valores de esa variable.

8.8.1 Tipos de datos primitivos.

Aunque Java es orientado a objetos, éste define varios tipos de datos primitivos no objetos

Grupo	Subgrupo	Tipo	Tamaño en Bits	Máximo/Mínimo
Aritméticos	Entero	`byte`	8	$2^7 - 1 = 127$ $-2^7 = -128$
		`short`	16	$2^{15} - 1 = 32,767$ $-2^{15} = -32,768$

		`int`ϑ	32	2^{31} - 1 = 2,147,483,647 -2^{31} = -2,147,483,648
		`long`ϑ	64	2^{63} - 1 = 9,223,372,036,854,775,807 2^{63} = -9, 223,372,036,854,775,808
	Punto flotante	`float`	32	± 3.40282347e+38 ± 1.40239846e-45
		`double`	64	± 1.79769313486231570e+308 ± 4.94065645841246544e-324
No aritméticos	Carácter	`char`H	16	N/D*
	Lógico	`boolean`	1	N/D*

* N/D No disponible

H El valor por default de la declaración de una variable de tipo `char` es *null*.

ϑ El único caso en el cual `ArithmeticException` es llamado es cuando se intenta dividir entre cero, así se prueba para cero antes de intentar efectuar la operación de división (en este caso la excepción es arrojada automáticamente según esta definido en la clase `java.lang.Math`).

Cuando el valor de un entero excede sus rangos definidos no se produce una excepción aritmética `ArithmeticException`. En cambio, sucede que el valor se "envuelve" por el otro extremo del rango numérico para ese tipo.

Java inicia todos los tipos de datos primitivos a valores por default si el valor inicial no es especificado explícitamente por el programador. Las variables enteras y de punto flotante son iniciadas con cero. El tipo de dato `boolean` es establecido como `false`.

A diferencia de C, Java especifica el tamaño de cada primitiva y el rango resultante de valores. Los tamaños definidos por el lenguaje son un elemento para facilitar su transferencia (*portable*).

Java no tiene la elección de sus tipos signados y sin signar. Los primitivos aritméticos son signados, y el tipo primitivo `char` es sin signar.

8.8.2 Tipos de datos por referencia.

Existen situaciones en las cuales las variables deben ser agrupadas juntas lógicamente para manipulación, posiblemente porque van a ser accedidas en secuencia o los identificadores deben apuntar a objetos localizados dinámicamente. Estos son llamados tipos de datos por referencia.

Una variable de un tipo de dato primitivo contiene un valor de ese tipo de dato primitivo; un tipo de dato por referencia contiene la dirección de un valor en vez de el valor

por sí mismo. La ventaja es que tipos de datos por referencia pueden contener direcciones que apuntan a una colección de otros tipos de datos. Esos mismos tipos de datos pueden ser tipos de datos primitivos o tipos de datos por referencia.

Hay tres tipos de variables por referencia: *array*, *class*, e *interface*.

8.8.2.1 Arreglos.

Los arreglos de variables son grupos de variables unidimensionales o multidimensionales.

Los elementos pueden ser de tipo primitivo, tales como `float`, `char` ó `int`. Los elementos también pueden ser `class` o `interface`. Los arreglos pueden consistir de otros arreglos.

Se puede declarar un arreglo sin localizarlo. En otras palabras, la variable por si misma es creada, pero ningún espacio es reservado en memoria para los objetos del arreglo hasta que el arreglo es inicializado o los valores son asignados a los elementos del arreglo. Los arreglos son generalmente iniciados con el comando `new`, el cual crea una nueva instancia de un tipo de dato por referencia. Esto es conceptualmente similar a la orden `malloc` en C o el comando `new` en C++. En Java, declarar un tipo de dato por referencia no crea automáticamente el espacio para ese tipo de dato; esto solamente crea espacio para la dirección que apunta a ese tipo.

Los arreglos son declarados incluyendo corchetes cuadrados después del tipo de dato y antes o después del nombre del arreglo.

A continuación se muestra un fragmento de código para la declaración de un arreglo, se creación y la asignación de valores a los elementos del arreglo:

```
class Arreglo{
 public static void main (String args[ ] ){
  int TAMAÑOLISTA = 5;
  String [ ] Lista_de_Compras; //declara el arreglo
  int i = TAMAÑOLISTA;

   // crea el arreglo

   Lista_de_Compras = new String[TAMAÑOLISTA];

   //inicializa arreglo

   Lista_de_Compras[0] = "papas";
       Lista_de_Compras[1] = "jitomate";
```

```
      Lista_de_Compras[2] = "leche";
      Lista_de_Compras[3] = "arroz";
      Lista_de_Compras[4] = "frijol";

      for(i=0; i < TAMAÑOLISTA; i++){
        System.out.println(Lista_de_Compras[i]);
      }
 }
}
```

Se puede realizar la inicialización en la misma línea que la declaración:

```
char ArregloCaracteres[ ] = new char[5];
```

La orden `new` inicializa el arreglo de caracteres a *null*. El espacio de memoria es reservado al arreglo de tal forma que diferentes valores puedan ser asignados a él.

En Java, los subíndices siempre comienzan en cero y van hasta la longitud del arreglo menos uno. Todos los elementos del arreglo deben ser del mismo tipo.

Un ejemplo de un arreglo de tipo `long` en dos dimensiones se define (declara) como:

```
long [ ] [ ] dosDimNumeros;
```

Nota: En teoría, todos los arreglos Java son de una dimensión. De hecho, un arreglo de arreglos puede definirse en Java, funcionando como un arreglo de dos dimensiones. (Esta característica de Java, porque las especificaciones técnicas de Java indican que todos los arreglos son de una dimensión).

8.9 Operadores.

Los operadores son los símbolos utilizados para las operaciones aritméticas y lógicas.

Los operadores aritméticos de Java.

Operador	Operación	Ejemplo
+	Suma	g + h
-	Resta	g - h
*	Multiplicación	g * h
/	División	g / h
%	Módulo	g % h

Los operadores de asignación de Java.

Operador	Operación	Ejemplo	Significado
=	Asigna un valor	a = 6	a = 6
+=	Añade a la variable a la izquierda	g += h	g = g + h
-=	Resta de la variable a la izquierda	g -= h	g = g - h
*=	Multiplica por la variable a la izquierda	g *= h	g = g * h
/=	Divide la variable a la izquierda	g /= h	g = g / h
%=	Módulo de la variable a la izquierda	g %= h	g = g % h

Los operadores de incremento y decremento de Java

Operador	Operación	Ejemplo	Significado
++	Aumenta en uno	g++ ó ++g	g = g + 1
--	Disminuye en uno	g-- ó --g	g = g - 1

Los operadores de comparación de Java (los cuales regresan `true` ó `false`)

Operador	Operación	Ejemplo	Significado
==	Igual	g == h	¿g es igual a h?
!=	No igual (diferente)	g != h	¿g es diferente de h?
<	Menor que	g < h	¿g es menor que h?
>	Mayor que	g > h	¿g es mayor que h?
<=	Menor o igual que	g <= h	¿g es menor o igual que h?
>=	Mayor o igual que	g >= h	¿g es mayor o igual que h?

Los operadores entre pares de bits (*bitwise*)

Operador	Operación
&	Operación AND bit con bit
|	Operación OR bit con bit
^	Operación XOR bit con bit
<<	Corrimiento a la izquierda
>>	Corrimiento a la derecha
>>>	Corrimiento a la derecha llenado con ceros
~	Complemento bit a bit
<<=	Asignación de corrimiento a la izquierda
>>=	Asignación de corrimiento a la derecha
>>>=	Asignación de corrimiento a la derecha y llenado con ceros
X&=y	Asignación AND
X|=y	Asignación OR

X^=y	Asignación NOT (negada)

Recurriendo a las tablas anteriores se tienen los siguientes operadores unarios.

Operador	Operación
-	Negación unaria (solo trabaja con la variable "temporalmente")
~	Complemento bit por bit (solo trabaja con la variable "temporalmente")
++	Aumenta en uno (si modifica el valor de la variable)
--	Disminuye en uno (si modifica el valor de la variable)
!	No (solo trabaja con la variable "temporalmente")

8.9.1 Orden de evaluación en Java.

Java evalúa de izquierda a derecha. Otra forma de recordar que Java evalúa de izquierda a derecha es entre operadores del mismo nivel de precedencia.. En Java, + y - son evaluados en el mismo nivel de precedencia. Por lo tanto, si ambas operaciones + y - suceden en la misma expresión, Java resolverá la expresión de izquierda a derecha.

A continuación se muestran la lista de operadores de la más alta a la más baja precedencia. Cualquier operador que este en la misma línea es evaluado en el mismo orden.

Operadores			
[ ]	( )		
--	!	~	instanceof
new (tipo) expresión			
*	/	%	
+	-		
<<	>>	>>>	
<	>	<=	>=
==	!=		
&			
^			
&&			
||			
?:			
=	op=		

8.9.2 Conversión entre tipos primitivos.

Los tipos primitivos pueden cambiar de tipo en 3 formas: por promoción aritmética, por conversión de asignación, y por forzamiento explícito.

8.9.2.1 Promoción aritmética.

Con el fin de llevar a cabo las operaciones binarias primitivas, Java puede convertir uno o los dos argumentos a un tipo mayor. Las reglas son las siguientes:

- Si un operando es de doble precisión, el otro es convertido a doble precisión.
- De otro modo, si un operando es un `float`, el otro es convertido a un `float`.
- De otro modo, si un operando es un `long`, el otro es convertido a un `long`.
- De otro modo, ambos operandos son convertidos a `int`.

Los boolean no pueden ser convertidos a y de los tipos de datos aritméticos; un tipo de dato char puede ser convertido a y de cualquier tipo numérico.

8.9.2.2 Conversión de asignación.

En una sentencia de asignación, si el tipo de variable difiere de la expresión del lado derecho, una conversión puede llevase a cabo, o un error durante la compilación puede ocurrir. Una conversión se lleva a cabo si la magnitud del resultado puede ser conservado en la variable receptora. Java se refiere a esto como una conversión de ensanchamiento (*widening*). Como podemos esperar lo contrario es una conversión de adelgazamiento (*narrowing*), donde la magnitud del resultado no podría ser conservada. Un error de compilación se producirá si una conversión de adelgazamiento es implicada por la asignación (se puede suprimir este error con un forzamiento explícito). Algunas conversiones de ensanchamiento pueden producir un error de redondeo, por ejemplo, cuando se esta convirtiendo de un `long` a un `float`.

Conversiones entre tipos primitivos.

de: / a:	`byte`	`char`	`short`	`int`	`long`	`float`	`double`
`byte`		n	w	w	w	w	w
`char`	n		n	w	w	w	w
`short`	n	n		w	w	w	w
`int`	n	n	n		w	p	w
`long`	n	n	n	n		p	p
`float`	n	n	n	n	n		w
`double`	n	n	n	n	n	n	

n - Conversión de adelgazamiento (narrowing)
w - Conversión de ensanchamiento (widening)
p - Conversión de ensanchamiento con pérdida de precisión potencial.

8.9.2.3 Forzamientos explícitos.

A diferencia de las conversiones de asignación, un forzamiento puede realizar una conversión de adelgazamiento. Como en C los forzamientos son especificados encerrando un tipo de datos entre paréntesis, por ejemplo,

```
long obtenunlong = 1234567890;
int reducelo = (int) obtenunlong;
```

En este ejemplo, el forzamiento a un `int` es necesario, ya que el compilador no permitirá que un `long` sea asignado directamente a un `int`.

Cualquier tipo primitivo puede ser forzado a cualquier otro, excepto para el `boolean`, el cual no puede ser forzado a ningún otro. Para convertir un boolean a valores aritméticos podemos utilizar el operador "?". ((b)?1:0) donde b es `boolean`.

8.10 Sentencias y control de flujo.

Una sentencia es cualquier línea de código que termina en un punto y coma.
Sentencias: p. Ej. a=1;
i++;
llamada a método;
una declaración;

Un bloque es un grupo de sentencias que forman una sola sentencia compuesta. ¿Cómo indicarle a Java donde comienza y donde termina un bloque?. Los caracteres { y } agrupan tales secciones juntas. El carácter {, el carácter }, y todo el código entre estos caracteres es llamado un bloque. Las llaves deben estar siempre apareadas, para indicar donde comienza y donde termina un bloque.

8.10.1 Expresiones condicionales.

Las expresiones condicionales generalmente ejecutarán una de varias secciones de código basado en la prueba de una condición. Este código puede ser tan simple como una sola sentencia, o de secciones más complejas de código que estarán hechas de muchas sentencias. Las expresiones condicionales son usadas para tomar decisiones en un programa. Son usadas para evaluar si una condición es cierta o falsa y saltará a diferentes secciones del código basado en la respuesta.

8.10.1.1 La sentencia `if`, y la cláusula opcional `else`.

```
if (expresión) sentencia;
```

o bien

```
if (expresión) {
```

```
  sentencia(s);
}
[ else {
     sentencia(s);
  } ]
```

8.10.1.2 La sentencia `switch`.

```
switch (expresión){
 case valor1:
     sentencia(s);
    break;
 case valor2:
     sentencia(s);
     break;
        .
        .
        .
 default:
     sentencia(s);
     break;
}
```

8.10.2 Expresiones de ciclos.

Las expresiones de ciclos generalmente continúan un ciclo a través de una sección de código hasta que se da una cierta condición. Algunas expresiones de ciclos prueban la condición antes de ejecutar el código. Otras expresiones de ciclos prueban la condición después de ejecutar el código.

8.10.2.1 while.

```
while (expresión) sentencia;
```

ó

```
while (expresión){
   sentencia(s);
}
```

8.10.2.2 do-while.

```
do sentencia; while (expresión);
```

ó

```
do {
  sentencia(s);
} while (expresión);
```

8.10.2.3 for.

```
for (inicialización; expresión; modificación) sentencia;
```

ó

```
for (inicialización; expresión; modificación){
  sentencia(s);
}
```

break.

`break` puede ser utilizado para salirse a la mitad de un ciclo `for`, `do`, `while`. Cuando una sentencia `break` es encontrada, la ejecución del ciclo se detiene inmediatamente y continua en la primera sentencia inmediatamente después del ciclo donde se encuentra el `break`.

continue.

La construcción `continue` puede ser utilizado para saltarse partes de un ciclo for, do, while. Cuando una sentencia `continue` es encontrada , la ejecución del ciclo continua inmediatamente al principio del ciclo, saltando todo el otro código entre ésta y el final del ciclo.

Ciclos etiquetados.

Si `break` y `continue` solo envían al final o al principio del ciclo, ¿que hacer si se tienen ciclos anidados y se necesita salirse de más de uno de los ciclos? Java proporciona una versión extendida de `break` y `continue` para este propósito. Añadiendo una etiqueta a un lazo y referenciándolo en una sentencia `break` o `continue`, se puede lograr que la ejecución continúe al final o al principio del ciclo elegido.

9. Visión de las clases para Aplicaciones y de las clases para *Applet*.

Los *applets* son más restringidos en Java en lo que pueden hacer. Debido a que los navegadores *Web*
Son una forma común para acceder a los *applets*, lo que un *applet* puede hacer al sistema local, es un tema de seguridad. Por lo tanto, se evita que los *applets* lean o escriban archivos al disco local, accesos a memoria directamente y conexiones de aperturas a otros *applets* que no residan en el mismo servidor que el *applet* que realiza la conexión de apertura.

Esto provee seguridad básica, ya que evita que un *applet* es una página *Web* desconocida obtenga automáticamente una copia del archivo de *password* y la envíe al autor del *applet*. Sin embargo, Java permite que algunas restricciones sean reducidas.usando el *appletviewer*, se puede permitir a un *applet* el acceso para leer o escribir a un directorio particular de un conjunto de directorios. Esto es potencialmente peligroso y deberá ser utilizado con precaución.

El mecanismo que provee la seguridad para los *applets* esta en la aplicación del cliente que corre el *applet* (por ejemplo, el navegador o el *applet viewer*). Este mecanismo es usualmente el controlador de seguridad de *applets*.

Las aplicaciones no tienen estas restricciones. Los usuarios que corren aplicaciones Java se les permite acceder a cualquier archivo, localización de memoria, y a los recursos de la red a los que normalmente podrían acceder.

9.1 Clases para aplicaciones.

Las aplicaciones Java son verdaderas aplicaciones, como las desarrolladas en cualquier otro lenguaje, tal como C o C++. La desventaja de las aplicaciones en Java contra las desarrolladas en otros lenguajes que son compilados es la velocidad. Es difícil para un lenguaje interpretado competir contra un código compilado nativamente.

Algunos kits de desarrollo de software para Java incluyen compiladores com el de Symantec Café.

Las aplicaciones Java son similares en estructura a los applets. La única diferencia importante son los métodos de declaración y la invocación del programa.

`main.`

Cada aplicación Java debe definir un método llamado `main`, el cual es similar al `main` en C y C++. Cuando una aplicación Java es invocada, el intérprete de Java busca el método llamado `main` y comienza la ejecución ahí.

Los métodos `main` siempre deben ser declarados públicos. El método `main` debe ser declarado en sí mismo en un formato fijo:

```
public static void main (String args[ ]) {
   ... cuerpo de main ...
}
```

La declaración `public` posibilita a que `main` sea accesible desde programas externos. `static` indica que este método no puede ser modificado por subclases. `void` significa que este método no tiene valor de retorno de ningún tipo. `main` es el nombre requerido para el método. `(String args[])` indica que `main` tendrá argumentos desde la línea de comandos que consiste de un arreglo del tipo `String`. Este arreglo contiene los argumentos especificados en la línea de comandos cuando la aplicación comenzó.

A continuación se muestra una aplicación estándar Java del clásico Hola mundo!:

```
public class HolaMundo{
 public static void main (String args[]) {
    System.out.println("Hola mundo!");
  }
}
```

Para compilar el ejemplo teclee: `javac HolaMundo.java`

Cuando la compilación termine, invoque al interprete de Java: `java HolaMundo`

9.2 Clases para *Applet*

Los *applet* son el aspecto más famoso del lenguaje de programación Java al momento. Java esta asociado con la *World Wide Web*; los *applets* de Java son poderosos porque pueden transformar una página *Web* bastante estática y convertirla en un ambiente multimedia altamente interactivo y "animado".

Los *applets* son una extensión de una clase existente de Java, `java.applet.Applet`. Un *applet* es un ejemplo de una subclase.

A continuación se muestra un ejemplo de un *applet*:

```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;

public class HolaMundoApplet extends Applet {
 public void paint (Graphics g){
  g.drawString("Hola Mundo!",10,10);
 }
}
```

Después de compilar el código fuente para el *applet*, se necesitará crear un documento HTML que será utilizado por un navegador o un *applet viewer* para mostrar el *applet*. Aquí esta un ejemplo de un documento HTML para el programa `HolaMundoApplet`:

```
<HTML>
<HEAD>
<TITLE>HolaMundoApplet</TITLE>
</HEAD>
<BODY>
<APPLET CODE="HolaMundoApplet" WIDTH=300 HEIGHT=300></APPLET>
</BODY>
</HTML>
```

Los *Applets* generalmente deben ser más a "prueba de balas" que las aplicaciones debido a los ambientes en los cuales corren. No solo deben correr, deben ser capaces de manejar eventos como clicks de mouse, repintados, suspenciones temporales, y otros. Muchas cosas que aparecen en la superficie parecen estar siendo manejadas por el navegador pero en realidad están siendo manejadas por el código *applet*.

Un ejemplo de esto es la siguiente secuencia de eventos: un *applet* imprime una imagen sobre la pantalla. El usuario trae otra ventana que parcialmente obstruye la imagen original, entonces quita la nueva ventana. ¿Quién es el responsable de repintar la imagen del *applet*? No es el navegador sino el *applet*. El navegador informará al *applet* que el repintado es necesario, pero es el *applet* el que realiza el repintado o lleva a cabo otra acción.

Los *applets* comienzan la ejecución diferente que las aplicaciones. Las aplicaciones comienzan la ejecución del programa llamando al método `main`. Los *applet* en cambio usan los métodos `init` y `start`. El navegador invocará al método `init` seguido por el método `start` cada vez que el *applet* Java es comenzado. Esto no tiene que estar declarado explícitamente en el *applet*.

Revisando el ejemplo `HolaMundoApplet`. Observamos que no hay una llamada que invoque a `paint`, pero la pantalla es pintada de cualquier forma. Muchas cosas suceden detrás de escena en los *applets* Java. Este es un ejemplo de un *applet* que comienza la ejecución sin una declaración explícita de los métodos `init` y `start`, en cambio hacen uso de los métodos default `init` y `start`. El método default `start` llama automáticamente al método `repaint`, el cual entre otras actividades, invoca al método `paint`. `paint` debe ser declarado explícitamente para que cualquier cosa sea escrita a la pantalla.

Resumiendo la serie de eventos que suceden cuando un navegador invoca un *applet*. Es importante entender esta secuencia si cualesquiera de los métodos de comienzo va a ser

sobreescrito. El *applet* no efectuará lo que se espera si cualquier cosa es dejada fuera. Cuando un navegador llama a un *applet*, primero el *applet* llama a `init`. Ya sea que este explícitamente declarado o sea su default. El navegador entonces llama a `start`, el cual ya sea explícitamente declarado o sea el default. Dentro de `start` hay una llamada a `repaint`, el cual dentro de su lista de actividades hace una llamada al método `paint`. El método implícito `paint` no escribe nada a la pantalla, y por lo tanto el método debe ser sobreescrito.

`init.`

El método `init` sólo es llamado la primera vez que un *applet* es cargado dentro del navegador. Las inicializaciones de una sola vez se llevan a cabo en la llamada al método `init`. Este es un buen lugar para obtener argumentos de la línea de comandos (órdenes) de la página HTML desde la cual el *applet* fue invocado.

El método `init` tiene un formato fijo. Este siempre debe llamarse `init`, tiene un valor de retorno de tipo `void`, es declarado público, y no tiene argumentos.

No se requiere un método `init`. Si se hace uno, éste sobreescribe un método `init` existente en la clase `java.applet.Applet`; esta es la razón por la que el nombre, tipo de retorno, y la otra información están fijas.

`start.`

El método `start` es llamado después del método `init` la primera vez que el *applet* es cargado dentro del navegador o si un *applet* ha sido suspendido y debe ser recomenzado. Si el compilador no puede encontrar un método `start` explícitamente declarado en el *applet*, utilizará el método `start` por default en `java.applet.Applet`.

No se requiere de un método `start`. Como `init`, Si se hace uno, éste sobreescribe un método `start` existente en la librería `java.applet.Applet`. El método `start` tiene un formato fijo. Este siempre debe llamarse `start`, tiene un valor de retorno de tipo `void`, es declarado público, y no tiene argumentos.

`stop.`

El método stop es llamado siempre que un *applet* debe ser detenido o suspendido. Sin este método el *applet* continua corriendo, consumiendo recursos aún si el usuario ha dejado la página sobre la cual el *applet* esta localizado. Hay ocasiones cuando es deseable que la ejecución continúe, pero en general no lo es.

Al igual que con `init` y `start`, no se requiere de un método explícito `stop`. Si se hace uno, éste sobreescribe un método `stop` existente en la librería `java.applet.Applet`. El método `stop` tiene un formato fijo. Este siempre debe llamarse `stop`, tiene un valor de retorno de tipo `void`, es declarado público, y no tiene argumentos.

`paint.`

El método `paint` es utilizado para pintar o repintar la pantalla. Éste es llamado automáticamente por `repaint` o puede ser llamado explícitamente por el *applet*. El *applet* llama a `paint` cuando el navegador requiere un repintado, tal como cuando un *applet* ocluido es traído al frente de la pantalla otra vez.

`paint` tiene un formato fijo. Éste siempre debe ser llamado `paint`, tiene un tipo de retorno `void`, y debe ser declarado público. A diferencia de `init` y `start`, éste tiene un argumento de tipo `Graphics`. Éste es un tipo predefinido en Java que tiene muchos de los métodos para escribir gráficos a la pantalla.

El compilador no requiere el método `paint`. Si se hace uno, éste sobreescribe un método `paint` existente en la clase `java.applet.Applet`. Sin embargo, si no se sobreescribe éste, no se escribirá nada en la pantalla.

10. Modificadores para las clases de acuerdo al acceso y al tipo.

Los modificadores afectan ciertos aspectos de las clases. Se especifican en la declaración de una clase antes del nombre de la clase.

Los modificadores de las clases son utilizados para especificar dos aspectos de las clases: el acceso y el tipo. Los modificadores de acceso son utilizados para regular el uso interno y externo de las clases. Los modificadores de tipo declaran la implementación de una clase. Una clase puede ser utilizada como una plantilla (*template*) para sus subclases o como una clase en y de sí misma.

Las clases pueden ser declaradas con seguridad tal que sólo pueden ser accedidas fuera de su paquete cuando se utilice la palabra reservada `public` en la declaración de la clase. Si no se especifica ningún modificador de acceso al momento de la declaración, la clase sólo puede ser accedida desde su propio paquete. Se genera un error de compilación si se utiliza otro modificador de acceso en la declaración, tales como `private` y `protected`, los cuales están reservados para los métodos.

Sin embargo se puede hacer uso de modificadores de tipo en la declaración de la clase. Estos modificadores para las clases son `abstract` y `final`. El default es no especificar ningún modificador de tipo en la declaración de la clase.

11. Variables en las clases.

Las variables en las clases son utilizadas para contener los datos que serán utilizados después. Es una buena práctica de programación colocar las variables inmediatamente después de la declaración de la clase.

Cada clase puede tener variables asociadas a ella. Éstas variables caen en dos categorías: las que son particulares a cada instancia, llamadas variables instancia, y las que son globales a todas las instancias de una clase particular llamadas variables estáticas. El uso de la variable determina su tipo.

11.1 Variables instancia.

Las variables instancia existen sólo para una instancia particular de una clase (objeto). Esto significa que diferentes instancias de la clase dada tiene cada una la variable del mismo nombre, ya que Java almacena valores diferentes de esa variable en diferentes lugares en memoria. Cada una de estas variables instancia son manipuladas individualmente. Si uno de estos objetos es destruido, también su variable instancia. Se pueden acceder las variables instancia de otras instancias de una clase, pero la variable en si misma existe solamente en el objeto (instancia de la clase) en particular.

Las variables instancia son declaradas después de la declaración de la clase pero antes de la declaración de los métodos. Cada instancia de la clase tiene una copia de su variable y puede modificarla cuando lo necesite sin afectar ninguna otra copia de la variable en las otras instancias.

11.2 Instancias predefinidas.

Java trae tres valores de objetos predefinidos: `null`, `this`, y `super`.

11.2.1 null.

¿Qué sucede cuando una clase creada es una superclase? Una variable puede ser creada como un "lugar" (espacio en memoria) donde las subclases puedan llenar con valores. En este caso, la variable puede ser declarada null, teniendo el significado que ningún valor es asignado a la variable.

```
class unCoche{
 static String tipo = null;
 public void main (String args[ ]){
  unCoche coche = new unGato( );
```

```
   if (coche.tipo == null) {
    promptUnTipo("Introduce el tipo> ");
    }
 }
}
```

11.2.2 this.

En la instancia de un método no estático, hay una variable especial, `this`, que se refiere a la instancia actual. La variable `this` puede ser usada como un argumento a otro método. Esta también puede ser usada para llamar a un constructor de otro constructor (cuando hay sobrecarga de constructores). En este caso, la sintaxis es: `this(argumentos);`.

Para referirse al objeto actual use la palabra clave `this`, lo cual permite a la instancia actual de una variable ser referida explícitamente. Esto es valioso cuando la instancia actual de una variable va a ser pasada a otra clase que también usará la variable:

```
void promptUnTipo(String prompt){
 stringBuffer tipo;
 char ch ='\0';
  name = new StringBuffer( );
  System.out.println(prompt);
  System.out.flush( );
 while(ch != '\n'){
  try { ch = (char)System.in.read( );}
  catch(IOException e) {}
  tipo.append(ch);
  }
  this.tipo = tipo.toString( );
}
```

11.2.3 super.

Similar a la variable `this`, esta la variable `super`, La cual se refiere a las variables y métodos de la superclase inmediata. La variable `super` es útil para acceder a los métodos y variables en el padre que son sobreeescritos u ocultos en la clase. `super ([argumento(s)])`; puede ser llamado desde un constructor para llamar al constructor del padre.

`super` es una referencia a la superclase. Con frecuencia es utilizada para acceder a un miembro de la superclase de la clase actual. En el siguiente código, una subclase

llamada `unCocheFord` usa la palabra clave `super` para hacer referencia al método `promptUnTipo` en su superclase, `unCoche`:

```
class unCocheFord extends unCoche{
 void obtenInfCoche{
   super.promptUnTipo("Introduce el tipo> ");
 }
}
```

11.3 Variables estáticas.

Se puede modificar la declaración de las variables (y métodos) con el modificador `static`. Las variables estáticas existen solamente en una localidad en memoria y son accedidas globalmente por todas las instancias de una clase. Una variable no puede ser cambiada por una subclase si ha sido declarada `static` y `final`. Además, las variables estáticas comparten la misma información en todas las instancias de la clase.

Esta es una herramienta muy valiosa en situaciones en las cuales una variable debe ser compartida por varias instancias de una clase y/o subclases. Todas las instancias tendrán siempre el mismo valor para la variable. Todas las instancias que acceden a esa variable apuntan al mismo lugar en memoria. La variable permanecerá en ese lugar hasta que la última instancia que accede a la variable es removida de la memoria.

Una variable estática es declarada de la misma forma que una variable instancia pero tiene la palabra clave `static` al frente de la declaración.

12. Métodos de las clases.

Los métodos son funcionalmente similares a las funciones en C y C++. Suministran una forma de agrupar un bloque de código junto y entonces referirse a él por un nombre. Se puede utilizar el bloque de código otra vez simplemente refiriéndose al nombre del método.

Los métodos no requieren de una palabra clave explícita para su declaración tal como la requieren las clases. Ellos tienen nombres, argumentos y tipos de retorno.

Los argumentos son parámetros que son pasados al método cuando es llamado, un método puede construirse para realizar diferentes tareas. El código interno del método sabe como manipular los parámetros de entrada.

Los métodos también tienen un tipo de retorno. Este es un valor que puede ser regresado al código que llamó al método. Los valores de retorno pueden ser de cualquier tipo de datos Java válidos, incluyendo cadenas y valores numéricos.

Los métodos pueden ser llamados no solo de la clase actual, sino también de las subclases, superclases, y también de clases no relacionadas.

12.1 Tipos de datos de retorno.

Los métodos pueden regresar cualquier tipo de datos Java válido. Éstos podrían ser un `boolean`, un arreglo, un objeto.

`void` es un tipo de retorno especial en Java que indica que no hay un tipo de retorno de ningún tipo. Esto es usado por métodos que no necesitan regresar nada al programa que los llamó, o que modifican solo los argumentos del método o las variables globales.

Un ejemplo de un método que no necesita regresar nada es uno que sólo imprime su salida a la pantalla. No hay necesidad de regresar nada porque no hay ningún procesamiento posterior. No es necesario usar la palabra clave `return` en ninguna parte del método si el método es declarado `void`.

12.2 Modificadores de métodos.

Los modificadores del método controlan el acceso al método. Los modificadores también son usados para declarar el tipo de método.

12.3 Declarando la seguridad del método y su accesibilidad.

Las oraciones de declaración de métodos proveen de información al compilador acerca de los accesos permitidos. En términos de Java, la accesibilidad es seguridad. Los cinco niveles de acceso son:

Nivel	Acceso permitido
public	Todas las otras clases
private	Ninguna otra clase
protected	Subclases o mismo paquete
private protected	Subclases solamente
<default>	Mismo paquete

La clase que declara al método o variable siempre puede accederlo.

12.3.1 private.

El modificador private especifica que ninguna clase, incluyendo las subclases, pueden llamar a este método. Éste puede ser usado para ocultar completamente un método

de todas las otras clases. Si ninguna otra clase puede acceder al método, éste puede ser modificado como sea necesario sin causar problemas.

12.3.2 protected.

El modificador protected especifica que solo la clase en la cual es método esta definido o las subclases de esa clase pueden llamar al método. Esto permite el acceso para objetos que son parte de la misma aplicación, pero no de otras aplicaciones.

12.3.3 private protected.

El modificador `private protected` es una combinación especial de los modificadores de acceso `private` y `protected`. Este modificador especifica que sólo la clase en la cual el método esta definido o las subclases de la clase pueden llamar al método. No permite el acceso del paquete al igual que lo hace `protected` pero también permite el acceso a las subclases como `private` no lo permite.

12.3.4 default.

Los métodos default pueden ser llamados por la clase en la cual están declarados, las subclases y las clases en el mismo paquete. En este caso, ningún tipo de modificador es escrito explícitamente en la oranción de la declaración del método.

12.3.5 static.

El modificador `static` esta asociado sólo con métodos y variables, no con clases. El modificador `static` es utilizado para especificar un método que sólo puede ser declarado una vez. No se permite a ninguna subclase implementar un método del mismo nombre. Esto es usado para métodos tales como `main`, que son puntos de entrada dentro de un programa. El sistema operativo podría no saber cual método llamar primero si hubiera dos métodos `main`. Los métodos estáticos son usados para especificar métodos que no deberán ser sobreescritos en las subclases.

Un método estático sólo puede acceder variables estáticas. Tampoco puede llamar a métodos no estáticos. Los métodos estáticos son usualmente definidos para manipular variables estáticas, o para suministrar una sola funcionalidad que no depende de ninguna instancia de la clase. Los métodos estáticos son invocados directamente sin necesidad de crear un objeto de la clase. Para llamar a un método estático se utiliza el nombre de la clase, más un punto, como prefijo al nombre del método, por ejemplo:

```
double y = Math.sin(x);
```

o bien,

```
double y = java.lang.Math.sin(x);
```

12.3.6 final.

El modificador `final` indica que un objeto es fijo y no puede ser cambiado. Cuando se utiliza este modificador con un objeto al nivel de clases, esto significa que la clase no puede tener subclases. Cuando se aplica este modificador a un método, el método nunca puede ser sobreescrito. Cuando se aplica este modificador a una variable, el valor de la variable permanece constante. Se obtendrá un error durante la compilación si se intenta sobreescribir un método final o una clase final. También se obtendrá un error de compilación si se intenta cambiar el valor de una variable `final`.

Los métodos pueden ser declarados como finales, en este caso, la clase puede tener subclases, pero los métodos finales no pueden ser sobreescritos.

12.3.7 abstract.

El modificador `abstract` es utilizado para crear métodos plantilla (*templates*), los cuales son muy similares a las funciones prototipo en C y C++. Los métodos abstractos definen el tipo de retorno, nombre, y argumentos pero no incluyen los "cuerpos" de los métodos. Se requiere que las subclases implementen los métodos abstractos y suministren un cuerpo.

Las subclases deben implementar un "cuerpo" para los métodos abstractos, o causará un error de compilación.

12.4 Modificadores de Acceso.

En programación orientada a objetos, el acceso a los métodos y las variables es controlado a través de los modificadores de acceso. El lenguaje de programación Java define cuatro niveles de control de acceso:

- Métodos y variables privadas. (*private*)
- Métodos y variables protegidas. (*protected*)
- Métodos y variables amigables. (*friendly*)
- Métodos y variables públicos. (*public*)

12.4.1 Métodos y variables privadas (`private`).

Los métodos y variables que son controlados por un objeto asociado (los métodos y variables son parte del objeto) y que no son accesibles a objetos de diferentes clases son generalmente considerados como privados. La ventaja de esto es que solo los objetos de una clase particular pueden acceder a los métodos y variables sin limitación. Los métodos y variables privadas son igualmente accesibles sólo por objetos dentro de la misma clase.

12.4.2 Métodos y variables protegidas (`protected`).

Los métodos y variables que son controlados por un objeto asociado (los métodos y variables son parte del objeto) y son accesibles a objetos de la clase actual (la clase actual se refiere a la clase donde se encuentran los métodos y variables que van a ser accedidos) o una subclase de la clase actual son generalmente considerados como protegidos. La ventaja es que solo objetos de clases específicas pueden acceder las variables sin limitación.. Los métodos y variables protegidas son igualmente accesibles sólo por métodos en la misma clase o subclase.

12.4.3 Métodos y variables amigables (`friendly`).

Los métodos y variables que son accesibles en la mayoría de las circunstancias son considerados como amigables. Por *default*, los métodos y variables que se declaran en Java se suponen como amigables y son accesibles por cualquier clase y objeto en el mismo paquete (mismo paquete se refiere al paquete donde se encuentra la clase que contiene a los métodos y variables que van a ser accedidos). La ventaja de esto es que los objetos en un paquete particular (generalmente un conjunto de clases relacionadas) pueden accederse uno con otro sin limitación.

12.4.4 Métodos y variables públicas (`public`).

Los métodos y variables que son accesibles a todos los objetos, aún los que están fuera de la clase actual (la clase actual se refiere a la clase donde se encuentran los métodos y variables que van a ser accedidos) o del mismo paquete (mismo paquete se refiere al paquete donde se encuentra la clase que contiene a los métodos y variables que van a ser accedidos), son considerados como públicos. Los métodos y variables públicos(as) son accesibles por cualquier objeto o clase. Por lo tanto, los métodos o variables públicos(as) pueden ser accedidos sin limitación.

12.5 Sobrecarga de métodos.

A diferencia de las variables, los métodos pueden compartir nombres. En este caso, la lista de parámetros debe diferir como en C++, El compartir nombres entre métodos es llamado sobrecarga, los tipos de retorno pueden diferir entre los métodos que tienen el mismo nombre. La sobre carga de constructores también es permitida.

Durante la compilación, si es necesario, Java decide cual de los métodos sobrecargados deberá ser llamado.

En Java se puede sobrecargar cualquier método en la clase actual o cualquier superclase a menos que el método sea declarado estático.

12.6 Métodos nativos.

Java permite métodos que estén escritos en otros lenguajes. Los métodos nativos son indicados por la palabra clave `native`, y sólo contiene punto y coma después de la declaración, p.ej.:

```
native public static double nombre_metodo([parámetro(s)]);
```

13. Objetos.

En un lenguaje procedural, los datos usualmente son pensados como una cosa diferente al código. En programación orientada a objetos, el código es pensado como una cosa con los datos.

13.1 Creación y destrucción de objetos.

En lenguajes procedurales el espacio para los datos es pre-localizado en memoria. El compilador conoce el tamaño, tipo, y número de variables y localiza el espacio respectivo. El espacio localizado para datos esta fijo en memoria y continua localizado hasta que la aplicación termina.

El espacio de datos no es pre-localizado en Java, este es creado conforme se necesita. El espacio en memoria es utilizado mientras se estén accediendo los datos, entonces la memoria es automáticamente liberada. Esto es similar a utilizar las llamadas al sistema en C como `malloc` y `free`.

Entre Java y C++ hay diferencias. En C++, las funciones constructora y destructora corren sin importar si un objeto es creado o destruido. Java tiene un equivalente a la función constructora, pero no hay un equivalente a la función destructora de C++. Todos los objetos en Java son removidos usando un colector de basura automático. No hay forma de invocar a los destructores manualmente. Hay un método llamado `finalize` que puede ser usado como un destructor C++ para hacer la limpieza final de un objeto antes de que la colección de basura ocurra, pero los finalizadores tienen serias limitaciones.

13.2 Creación de una instancia.

Java localiza espacio de memoria similar al `malloc` en C usando la palabra clave `new`. `new` crea un objeto virtualmente de cualquier tipo, las excepciones son los tipos de datos primitivos. P. Ej.:

```
String cad;
Cad = new String(10);
```

13.3 Destrucción de una instancia.

El método `destroy` es llamado siempre que un *applet* se ha completado o esta siendo dado de baja (*shut down*). Cualquier limpieza final toma lugar aquí. El método `destroy` siempre debe ser llamado `destroy`, tener un tipo de retorno de tipo `void`, ser declarado público, y no tener argumentos.

`destroy` puede realizar muchas cosas, tales como: terminar limpiamente una conexión de red, escribir la información histórica (condensada) a los archivos (*log*) y otras acciones finales.

No se requiere de un método `destroy`. Este sobreescribe un método `destroy` existente en la clase `java.applet.Applet`; por lo tanto, el nombre, tipo de retorno, acceso de seguridad y argumentos están fijos.

13.4 El método constructor.

Recuérdese que Java no localiza espacio de memoria para los objetos al momento de iniciar la aplicación sino más bien cuando la instancia es creada por la palabra clave `new`. Varias cosas ocurren cuando `new` es invocado. Primero Java localiza suficiente memoria para mantener el objeto. Segundo, Java inicializa cualquier variable instancia a sus valores por default.

Tercero, Java hace una llamada a cualquier constructor que exista para esa clase. Los constructores son métodos especiales que son usados para inicializar un objeto. Un constructor puede hacer cualquier cosa que un método normal pueda hacer, pero usualmente éste es utilizado simplemente para inicializar variables dentro del objeto a algún valor de inicio.

Los constructores no tienen una palabra clave explícita. En cambio, el nombre del método constructor es el mismo nombre de la clase en la cual el constructor esta declarado.

Los métodos constructores siempre tienen un tipo de retorno `void`, por lo que no es necesario indicarlo explícitamente al momento de declarar el método.

Los constructores son como los métodos normales declarados explícitamente, en el sentido que pueden ser sobrecargados mediante la declaración de ellos más de una sola vez con diferentes argumentos. Múltiples constructores son creados declarando varios métodos con el mismo nombre que la clase. Igualmente, como en los métodos sobrecargados, Java determina cual constructor utilizar basado en los argumentos que se pasan sobre `new`.

Si una clase no tiene un constructor definido, entonces Java genera uno. Éste no toma argumentos y llama al constructor de la superclase inmediata sin argumentos. También inicializa todas las variables instancia a su valor por default. Si la superclase no tiene un constructor sin argumentos, entonces se produce un error de compilación.

13.5 El método `finalize`.

Cuando un objeto ya no es referenciado por cualquier otro objeto. Java recupera la memoria utilizando el colector de basura. Java llama a un método destructor antes de que la colección de basura se lleve a cabo. A diferencia de los métodos constructores, los métodos destructores tienen un nombre específico `finalize`.

El método `finalize` siempre tiene un tipo de retorno de tipo `void`, no tiene parámetros y sobreescribe un destructor por default que esta en la clase `java.lang.Objetc`.

Un programa puede llamar al método `finalize` directamente al igual que se puede hacer con cualquier otro método. De cualquier forma, llamando a `finalize` no se inicia ningun tipo de colección de basura Este es tratado como cualquier otro método si es llamado directamente. Cuando Java hace la colección de basura, `finalize` también es llamado aún si éste ya había sido llamado directamente por el programa.

El método `finalize` es como cualquiera de los otros métodos en el sentido que éste puede ser sobrecargado. Recuérdese, sin embargo, que Java llama a `finalize` automáticamente sin ningún argumento. Si Java encuentra un método `finalize` con argumentos durante la recolección de basura, buscará el método `finalize` sin argumentos. Si no lo encuentra, Java utilizará el método `finalize` por default en su lugar.

Una diferencia entre `finalize` y un destructor de C++ es que el sistema sólo llama a `finalize` cuando éste esta listo para recuperar la memoria asociada con el objeto. Esto no sucede inmediatamente después de que un objeto ya no es referenciado. La limpieza del sistema esta basada en la base de cuando se necesita. Puede existir un retardo significativo entre cuando la aplicación termina y cuando `finalize` es llamado. Esto es cierto cuando el sistema esta ocupado, pero no hay necesidad inmediata de más memoria. Por estas razones, puede ser mejor llamar al método `finalize` directamente en la

terminación del programa en vez de esperar que el colector de basura lo invoque automáticamente. Esto depende de la aplicación.

14. Herencia.

La herencia es una metodología según la cual el código desarrollado para un uso puede ser extendido para usarse para cualquier otra cosa sin necesidad de hacer una copia del código.

En Java, la herencia se lleva a cabo creando nuevas clases que son extensiones (`extends`) de otras clases. La nueva clase es conocida como una subclase. La clase original es conocida como superclase. La subclase tiene todos los atributos y comportamientos de la superclase, además de los atributos y comportamientos que defina ella misma. Una clase puede tener sólo una superclase. Esto es conocido como herencia simple. Una superclase puede tener múltiples subclases.

Si la clausula `extends` es omitida de la declaración de la clase, entonces la clase es una subclase inmediata de la clase `Object` de Java. La clase `Object` sirve como una superclase común a cualquier otra clase Java.

Para definir clases, parta de los aspectos mas generales (que engloban al mayor número de objetos) hacia los particulares (son específicos a cada objeto), viendo que tienen en común todos los objetos y cree una clase general (superclase), luego haga más específica su clasificación buscando los aspectos comunes de esta subclasificación (subclase(s)). Siga este proceso hasta agotar los aspectos de los objetos analizados.

15. Interfaces.

Las interfaces permiten la herencia múltiple de métodos. Una interfaz (`interface`) sólo define el nombre del método, tipo de retorno, y argumentos. No incluye código ejecutable. Se puede pensar de una interfaz como una plantilla de estructura y no de uso.

Las interfaces son usadas para definir la estructura de un conjunto de métodos que serán implementados por clases que van a ser diseñadas y codificadas. En otras palabras, los argumentos de llamada y los valores de retorno deben coincidir con los de la definición en la interfaz. El compilador comprueba esta coincidencia. Sin embargo, el código interno a un método definido por una interfaz puede lograr el resultado pretendido en una manera completamente diferente que el método en otra clase.

El concepto de usar interfaces es una variación de la herencia usada extensamente en Java. El principal beneficio de las interfaces es que muchas clases diferentes pueden implementar la misma interfaz. Esto garantiza que todas esas clases implementarán unos

pocos métodos comunes. Esto también asegura que todas las clases implementarán estos métodos comunes usando el mismo tipo de retorno, nombre y argumentos.

Los modificadores para las variables están limitados a un conjunto específico: `public static final`. En otras palabras, las variables declaradas en las interfaces pueden sólo funcionar como constantes. `public static final` son el modificador por default.

Los modificadores para los métodos están limitados a un conjunto específico también: public abstract, indicando que los métodos declarados dentro de una interfaz sólo pueden ser abstractos.

Se pueden declarar métodos sobrecargados en una interfaz al igual que en una clase.

Las interfaces también pueden extender otras interfaces.

16. Expresiones de referencia.

Los objetos y los arreglos son considerados tipos de datos de referencia en Java, esto es debido a que las variables objeto y los arreglos mantienen una referencia (apuntador) a los datos.

16.1 Arreglos.

Los arreglos son objetos de la superclase `Object`. Cualquier método de la clase `Object` puede ser llamado por un arreglo. P ej.:

```
int[] cuadrados = {1, 4, 9, 16, 25};
for (int i = 0; i<cuadrados.length; i++)
  System.out.println(cuadrados[i]      +      "      "      +
Math.sqrt(cuadrados[i]));
```

16.2 Asignación.

```
int[] a1 = {1,2,3};
int[] a2 = a1;
a1[0] = -1;
```

En este caso `a1` y `a2` apuntan a la misma localidad en memoria, por lo tanto si se modifica el elemento cero del arreglo de `a1`, también se ve modificado el elemento cero del arreglo `a2`.

```
int[] a1 = {1,2,3};
int[] a2 = (int[]) a1.clone();
a1[0] = -1;
```

En este caso el arreglo `a1` hace uso del método `clone` que heredo de la clase `java.lang.Object`, se crea una copia exacta (un clon) de `a1`, en otra localidad de memoria, por lo que ahora al modificar el elemento cero del arreglo `a1`, no altera el elemento cero del arreglo `a2`.

16.3 Comparación.

```
String s1 = new String("para comparar");
String s2 = new String("para comparar");
boolean prueba = (s1 == s2);
```

En este caso `prueba` es `false` porque `s1` y `s2` refieren a diferentes localidades de memoria.

```
String s1 = new String("para comparar");
String s2 = new String("para comparar");
boolean prueba = s1.equals(s2);
```

A fin de poder comparar las cadenas la clase `Object` define un método `equals`. En este caso `prueba` es `true`.

16.4 El operador `instanceof`.

El operador sólo puede ser aplicado a tipo de datos de referencia: `instanceof`. Éste provee de una forma para comprobar los tipos de datos durante la corrida del programa. El resultado es cierto si el primer argumento es del tipo o es una subclase del segundo argumento. P. Ej.:

```
if (argumento1 instanceof String)
  argumento1 = "algo";
```

16.5 Conversiones entre tipos por referencia.

Existen diferencias entre conversiones por asignación y conversiones por forzamiento. Los tipos de datos por referencia tienen sus propias reglas para estas conversiones. A continuación se muestran que asignaciones son válidas en la sentencia `s = t` donde `s` es un objeto de la clase `S` y `t` es un objeto de la clase `T`.

Pruebas del compilador para asignaciones `Object`.

S es: \ T es:	clase, no final	clase final	interface
clase, no final	T debe ser una subclase de S	T debe ser una subclase de S	Error de compilación
clase final	Error de compilación	T debe ser la misma clase que S	Error de compilación
interface	T debe implementar a S	T debe implementar a S	T debe ser subinterface de S

Si T es un arreglo con elementos con tipo de datos B, entonces S debe ser `Object`, o S debe tener elementos que son del mismo tipo primitivo, o un tipo de datos de referencia al cual B pueda ser asignado.

Para ciertos forzamientos explícitos, el compilador puede ser capaz de probar que son incorrectos. Si este es el caso se produce un error de compilación. De otra forma, la validez es comprobada durante la corrida del programa.

A continuación se muestran las reglas para el forzamiento de clases.

S es: \ T es:	clase, no final	clase final	interface
clase, no final	T debe ser una subclase de S, o viceversa	T debe ser una subclase de S	Correcto durante la compilación
clase final	S debe ser una subclase de T	T debe ser la misma clase que S	S debe implementar a T
interface	Correcto durante la compilación	T debe implementar a S	Correcto durante la compilación

Respecto a los arreglos, las reglas para el forzamiento son las mismas que para las conversiones por asignación, excepto que un Object puede ser capaz de ser forzado explícitamente a un arreglo. La validez durante la corrida dependerá si el objeto mantiene una referencia a un arreglo del mismo tipo de datos del arreglo de mano izquierda.

17. Unidades de compilación.

La unidad de compilación es el contenido de un archivo que puede ser compilado por el compilador Java. Una unidad de compilación puede no tener, tener una, o varias clases definidas en ella. Si tiene más de una clase, entonces no más de una puede ser declarada como pública.

Por cada una de las clases se genera un archivo `.class`

18. Referencias bibliográficas.

Norton, P., Stanek, W., *Guide to Java programming*, Sams net, Estados Unidos © 1996.
Groner, D., et. al., *Java Language API superbible*, Waite Group, Estados Unidos © 1996.
Siddalingaiah, M., Lockwood, S. D., *Java, How-To*, Waite Group, Estados Unidos © 1996.

Appendix D

D Manual Básico del Lenguaje Java. Una Introducción.
Referencias bibliográficas.

Appendix E
Métodos de resolución de sistemas de Ecuaciones Lineales .

En este apéndice vamos a mostrar algunos métodos especiales para resolver el sistema

$$[A]\vec{x} = \vec{b}$$

donde $[A]$ es una matriz de $n \times n$, $\vec{x}$ es un vector de tamaño *n* y $\vec{b}$ también es un vector de tamaño *n*. Definiremos el producto punto de dos vectores como:

$$\langle \vec{x}, \vec{y} \rangle = \vec{x}^T \vec{y} = \sum_{i=1}^{n} x_i y_i$$

E.1 Algoritmo del método del descenso más rápido.

input $\vec{x}$, [*A*], $\vec{b}$, *M*
output 0, $\vec{x}$
for *k = 1, 2, 3, ..., M* **do**
$\vec{v} \leftarrow \vec{b} - [A]\vec{x}$
$t = \langle \vec{v}, \vec{v} \rangle / \langle \vec{v}, [A]\vec{v} \rangle$
$\vec{x} \leftarrow \vec{x} + t\vec{v}$
output *k*, $\vec{x}$
end.

E.2 Algoritmo del método del gradiente conjugado.

input $\vec{x}$, [*A*], $\vec{b}$, *M*, ε, δ
$\vec{r} \leftarrow \vec{b} - [A]\vec{x}$
$\vec{v} \leftarrow \vec{r}$
$c \leftarrow \langle \vec{r}, \vec{r} \rangle$
for *k* = *1*, *2*, *3*, ..., *M* **do**
    if $\langle \vec{v}, \vec{v} \rangle^{1/2} < \delta$ **then stop**
    $\vec{z} \leftarrow [A]\vec{v}$
    $t \leftarrow c / \langle \vec{v}, \vec{z} \rangle$
    $\vec{x} \leftarrow \vec{x} + t\vec{v}$
    $\vec{r} \leftarrow \vec{r} - t\vec{z}$
    $d \leftarrow \langle \vec{r}, \vec{r} \rangle$
    if $d^2 < \varepsilon$ **then stop**
    $\vec{v} \leftarrow \vec{r} + (d/c)\vec{v}$
    $c \leftarrow d$
    output *k*, $\vec{x}$, $\vec{r}$
end.

E.3 Algoritmo del método del gradiente conjugado precondicionado.

input $\vec{x}$, [A], $\vec{b}$, M, ε, δ
$\vec{r} \leftarrow \vec{b} - [A]\vec{x}$
resuelva $[Q]\vec{z} = \vec{r}$ para $\vec{z}$
$\vec{v} \leftarrow \vec{z}$
$c \leftarrow \langle \vec{z}, \vec{r} \rangle$
for $k = 1, 2, 3, ..., M$ **do**
    if $\langle \vec{v}, \vec{v} \rangle^{1/2} < \delta$ **then stop**
    $\vec{z} \leftarrow [A]\vec{v}$
    $t \leftarrow c / \langle \vec{v}, \vec{z} \rangle$
    $\vec{x} \leftarrow \vec{x} + t\vec{v}$
    $\vec{r} \leftarrow \vec{r} - t\vec{z}$
    resuelva $[Q]\vec{z} = \vec{r}$ para $\vec{z}$
    $d \leftarrow \langle \vec{z}, \vec{r} \rangle$
    if $d^2 < \varepsilon$ **then**
        $e \leftarrow \langle \vec{r}, \vec{r} \rangle$
        if $e^2 < \varepsilon$ **then stop**
    endif
    $\vec{v} \leftarrow \vec{z} + (d/c)\vec{v}$
    $c \leftarrow d$
    output k, $\vec{x}$, $\vec{r}$
end.

Referencias bibliográficas.

Kincaid, David, Cheney, Ward, *Análisis numérico*, Addison-Wesley, USA, 1994.

Appendix E

E Métodos de resolución de Sistemas de Ecuaciones Lineales.

E.1 Algoritmo del método del descenso más rápido.

E.2 Algoritmo del método del gradiente conjugado.

E.3 Algoritmo del método del gradiente conjugado precondicionado.

Referencias bibliográficas.

Resumen de las clases y métodos matemáticos implementados en Java.

The classes that have been implemented are the following:

`ElementoLista` this class is used to generate instances of the elements in a double link list, with the following *variables* declared:

```
Object _dato;            // the heart of the leaf (data)
ElementoLista _enlace_proximo;// reference to the next leaf
ElementoLista _enlace_anterior;// reference to the prior leaf
```

This class has the *methods*:

This method is the constructor of the class:

```
public ElementoLista (Object dato, ElementoLista
 Enlace_proximo, ElementoLista enlace_anterior)
```

where,

`dato` is an object of any data type in Java,

`enlace_proximo` is an instance of the class `ElementoLista`, this represent the reference to the next element in the list,

`enlace_anterior` is an instance of the class `ElementoLista`, this represent the reference to the prior element in the list.

This method return the data of the leaf:

```
public Object dato()
```

This method return reference to the next leaf:

```
public ElementoLista enlace_proximo()
```

This method return reference to the prior leaf:

```
public ElementoLista enlace_anterior()
```

This method set data of leaf:

```
public void establece_dato(Object o)
```

where,

`o` is an object of any data type in Java.

This method set reference to next leaf:

```
public void establece_enlace_proximo(ElementoLista enlace)
```

where,

`enlace` is the reference to the next leaf.

This method set reference to prior leaf:
public void establece_enlace_anterior(ElementoLista enlace)
where,
enlace is the reference to the prior leaf.

List0 this class generate a double linked list of **Object**, this has declared the following *variables*:

```
ElementoLista _cabeza;// reference to the start of the list,
or null if empty list
ElementoLista _cola;// reference to the end of the list, or
null if empty list
int _cuenta;// number of items in the list (for efficiency)
```

This class has the *methods*:

This method is one of the constructors of the class, default constructor an empty list:
public List0()

This method is other constructor of the class, convenience constructor – a list from an array:
public List0(**Object** unArreglo[])
where,
unArreglo[] is an array of the data type **Object**.

This method create a new linked list:
public synchronized Object clona()

This method return the number of entries in list:
public int tamaño()

This method return true if an element is contained in the list:
public boolean contiene(**Object** elem)
where,
elem is the particular element that we are looking for.

This method return the index of a particular element, or –1 if not found:
public int indiceDe(**Object** elem)
where,
elem is the particular element that we are looking for the index of it.

This method return the index of a particular element looking from a starting index, or –1 if not found:
public synchronized int indiceDe(**Object** elem, **int** comienzo)
where,

`elem` is the particular element that we are looking for the index of it,
`comienzo` is the starting index.

This method grow (add) an element to end of list:
```
public synchronized void añadeElemento(Object obj)
```
where,
`obj` is the **`Object`** to be added at the end of the list

This method place an element at the start of the list:
```
public synchronized void principioElemento(Object obj)
```
where,
`obj` is the **`Object`** to be placed at the start of the list.

This method eliminate an element at the start of the list:
```
public synchronized void EliminaprincipioElemento()
```

This method eliminate an element at the end of the list:
```
public synchronized void EliminafinElemento()
```

This method put an element at a particular location in the list, return false if the location is negative or too large, this should probably throw an exception instead:
```
public synchronized boolean insertaElementoEn(Object obj, int indice)
```
where,
`obj` is the **`Object`** to be inserted at a particular location in the list,
`indice` is the particular location.

This method eliminate an element at a particular location in the list, return false if the location is negative or too large, this should probably throw an exception instead:
```
public synchronized boolean EliminaElementoEn(int indice)
```
where,
`indice` is the particular location.

This method replace an element at a particular location in the list, return false if the location is negative or too large, this should probably throw an exception instead:
```
public synchronized boolean ReemplazaElementoEn(Object obj, int indice)
```
where,
`obj` is the **`Object`** to be replaced at a particular location in the list,
`indice` is the particular location.

This method get an element at a particular location in the list, return `obj` (the originally passed as argument) if the location is negative or too large, this should probably throw an exception instead:
```
public synchronized Object ObtenElementoDe(Object obj, int indice)
```

where,

`obj` is the **`Object`** to be get at a particular location in the list,
`indice` is the particular location.

This method reset list to empty:
`public synchronized void` `eliminaTodosLosElementos()`

This method generate a **`String`** representation of list, by chaining together **`String`** representation of elements:
`public synchronized String` `toString()`

`TipoList0` this class is used as a filter, due that `TipoList0` extends the `List0` class. `TipoList0` augment the `List0` class to constrain allowable types in the list. In addition to constructors and type-checking methods, all elements insertion or replace methods in `List0` are overridden.

This class has the following *variables* declared:

```
Class _clase;
```

This class has the *methods*:

This method is the constructor, that generate an empty list, with a particular type:
`public` `TipoList0(`**`Class`** `clase_permitida)`
where,
`clase_permitida` is the class allowed, you can use subclass (more specialized class) but not are allowed superclases or not related classes.

This method check for valid type:
`public synchronized void` `compruebaTipo(`**`Object`** `obj)`
where,
`obj` is an **`Object`** to check for a valid type.

This method makes a complain, generates an `IllegalArgumentException`:
`void` `tipoReclamo(`**`Object`** `obj)`
where,
`obj` is an **`Object`** used to show its class name.

This method intercept the method `añadeElemento` of `List0`, to check their type, if right type add item at end of list:
`public synchronized void` `añadeElemento(`**`Object`** `obj)`
where,
`obj` is the **`Object`** to be added.

This method intercept the method `insertaElementoEn` of `List0`, to check their type, if right type insert item at a particular place in the list:

```
public synchronized void insertaElementoEn(Object obj, int indice)
```

where,

`obj` is the **`Object`** to be inserted at a particular location,

`indice` is the particular location.

This method intercept the method `principioElemento` of `List0`, to check their type, if right type add item at start of list:

```
public synchronized void principioElemento(Object obj)
```

where,

`obj` is the **`Object`** to be added.

This method intercept the method `ReemplazaElementoEn` of `List0`, to check their type, if right type replace item at a particular place in the list:

```
public synchronized void ReemplazaElementoEn(Object obj, int indice)
```

where,

`obj` is the **`Object`** to be inserted at a particular location,

`indice` is the particular location.

The class `Escalar` is used to define all operations involved with scalar numbers.

The *methods* of the class `Escalar` are:

`public` `Escalar arcoCoseno().` This method calculates the trigonometric arc cosine. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. The domain is (-1 , 1), outside of this zero is returned and the range of this function is (0 , π).

`public` `Escalar arcoSeno().` This method calculates the trigonometric arc sine. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. The domain is (-1 , 1), outside of this zero is returned and the range of this function is (-π/2 , π/2).

`public` `Escalar arcoTangente().` This method calculates the trigonometric arc tangent. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. The domain is (-Inf , Inf) and the range of this function is (-π/2 , π/2).

public Escalar arcoTangente2(Escalar x). This method calculates the trigonometric arc tangent in a wider range. The scalar x (parameter) is the numerator and the actual instance (scalar that call to this method) is the denominator, i.e. arc tangent of (x / actual instance). This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. The domain is (-Inf , Inf) and the range of this function is ($-\pi$, π).

public Escalar convierteTipoEscalar(Class clase). To convert a scalar kind to another scalar kind, for example Double to Float, Float to Long, etc... We can convert a scalar to any kind of the following types: Double, Float, Integer, Long.

public Escalar coseno(). This method calculates the cosine of an angle. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. The domain is (-Inf , Inf) and the range of this function is (-1 , 1).

public Escalar divide(**double** db1). This method is used to divide a scalar number (dividend) with a double number (divisor), this is realized making a Double scalar number from a double primitive type number. The result is stored in a scalar variable that could be the dividend or another scalar variable. If the kind of the scalar number that we are dividing is different (Float, Integer, Long) to Double, the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar divide(**Double** db1). This method is used to divide a scalar number (dividend) with a Double number (divisor). The result is stored in a scalar variable that could be the dividend or another scalar variable. If the kind of the scalar number that we are dividing is different (Float, Integer, Long) to Double, the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar divide(Escalar escalar2). This method is used to divide two scalar numbers. The escalar2 is the divisor. The result is stored in a scalar variable that could be any of both dividend or divisor or another scalar variable. If the kind of the scalar numbers that we are dividing are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public EscalarComplejo divide(EscalarComplejo escalarComplejo). This method is used to divide a scalar number (dividend) with a

complex scalar number (divisor). The result is a complex scalar number. The kind (Double, Float, Long, Integer) of the scalar real part and the kind (Double, Float, Long, Integer) of the scalar imaginary part on the result are defined by this same parts of the scalar complex number on the argument (divisor). This may produce loss of precision.

public Escalar divide(**float** fl2). This method is used to divide a scalar number (dividend) with a float number (divisor), this is realized making a Float scalar number from a float primitive type number. The result is stored in a scalar variable that could be the dividend or another scalar variable. If the kind of the scalar number that we are dividing is different (Double, Integer, Long) to Float, the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar divide(**Float** fl1). This method is used to divide a scalar number (dividend) with a Float number (divisor). The result is stored in a scalar variable that could be the dividend or another scalar variable. If the kind of the scalar number that we are dividing is different (Double, Integer, Long) to Float, the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar divide(**int** in2). This method is used to divide a scalar number (dividend) with a int number (divisor), this is realized making a Integer scalar number from a int primitive type number. The result is stored in a scalar variable that could be the dividend or another scalar variable. If the kind of the scalar number that we are dividing is different (Double, Float, Long) to Integer, the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar divide(**Integer** in1). This method is used to divide a scalar number (dividend) with an Integer number (divisor). The result is stored in a scalar variable that could be the dividend or another scalar variable. If the kind of the scalar number that we are subtracting is different (Double, Float, Long) to Integer, the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar divide(**long** lg2). This method is used to divide a scalar number (dividend) with a long number (divisor), this is realized making a Long scalar number from a long primitive type number. The result is stored in a scalar variable that could be the dividend or another scalar variable. If the kind of the scalar number that we are dividing is different (Double, Float, Integer) to Long, the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar divide(**Long** lg1). This method is used to divide a scalar number (dividend) with a Long number (divisor). The result is stored in a scalar variable that could be the dividend or another scalar variable. If the kind of the scalar number that we are dividing is different (Double, Float, Integer) to Long, the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar elevaAlcuadrado(). This method get the square of the scalar number that is calling to this method. The domain is [-Inf , Inf] and the range is [-Inf , Inf].

public Escalar enteroMayor(). This method calculates the largest integer less than or equal to the scalar number that is calling to this method. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar enteroMenor(). This method calculates the smallest integer greater than or equal to the scalar number that is calling to this method. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar(**Class** clase). This constructor uses an kind of class to build an scalar of that type. The possible kinds of class are: Double, Integer, Float, Long. The initialization is always to zero.

public Escalar(**double** db1). This constructor get an primitive type of number to build an scalar number of kind Double. The initial value is db1.

public Escalar(**Double** db1). This constructor get an object of kind Double class to build an scalar number. The initial value is db1.

public Escalar(**float** fl1). This constructor get an primitive type of number to build an scalar number of kind Float. The initial value is fl1.

public Escalar(**Float** fl1). This constructor get an object of kind Float class to build an scalar number. The initial value is fl1.

public Escalar(**int** in1). This constructor get an primitive type of number to build an scalar number of kind Integer. The initial value is in1.

public Escalar(**Integer** in1). This constructor get an object of kind Integer class to build an scalar number. The initial value is in1.

public Escalar(**long** lg1). This constructor get an primitive type of number to build an scalar number of kind Long. The initial value is lg1.

public Escalar(**Long** lg1). This constructor get an object of kind Long class to build an scalar number. The initial value is lg1.

public Escalar exponencial(). This method computes the exponential of the scalar number that is calling to this method. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public boolean igual(Escalar escalar2). This method return true if the value of both scalar numbers are the same. If the kind of the scalar numbers are different (Double, Float, Integer, Long), they both are converted to equivalently compare them to the kind of the instance that call to this method. This can produce erroneous comparison. Caution must be observed due to this process of conversion some loss of precision is reached.

public Escalar logaritmoBase(Escalar base). This method computes the base log of the scalar number that is calling to this method. Where base is the parameter of this method. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar logaritmoNatural(). This method computes the natural log of the scalar number that is calling to this method. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar mayor(Escalar otro). This method return the larger of two scalar numbers. The number returned is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar menor(Escalar otro). This method return the smaller of two scalar numbers. The number returned is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar multiplica(**double** db1). This method is used to multiply a double number to a scalar number, this is realized making a Double scalar number from a double primitive type number. The result is stored in a scalar variable that could be the multiplicand or multiplier or another scalar variable. If the kind of the scalar number that we are multiplying is different (Float, Integer, Long) to Double, the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the

result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar multiplica(**Double** db1). This method is used to multiply a Double number to a scalar number. The result is stored in a scalar variable that could be the multiplicand or multiplier or another scalar variable. If the kind of the scalar number that we are multiplying is different (Float, Integer, Long) to Double, the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar multiplica(Escalar escalar2). This method is used to multiply two scalar numbers. The result is stored in a scalar variable that could be any of both multiplicand or multiplier or another scalar variable. If the kind of the scalars numbers that we are multiplying are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public EscalarComplejo multiplica(EscalarComplejo escalarComplejo). This method is used to multiply a scalar number with a complex scalar number. The result is a complex scalar number. The kind (Double, Float, Long, Integer) of the scalar real part and the kind (Double, Float, Long, Integer) of the scalar imaginary part on the result are defined by this same parts of the scalar complex number on the argument. This may produce loss of precision.

public Escalar multiplica(**float** fl2). This method is used to multiply a float number to a scalar number, this is realized making a Float scalar number from a float primitive type number. The result is stored in a scalar variable that could be the multiplicand or multiplier or another scalar variable. If the kind of the scalar number that we are multiplying is different (Double, Integer, Long) to Float, the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar multiplica(**Float** fl1). This method is used to multiply a Float number to a scalar number. The result is stored in a scalar variable that could be the multiplicand or multiplier or another scalar variable. If the kind of the scalar number that we are multiplying is different (Double, Integer, Long) to Float, the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar multiplica(**int** in2). This method is used to multiply a int number to a scalar number, this is realized making a Integer scalar number from a int primitive type number. The result is stored in a scalar variable that could be the

multiplicand or multiplier or another scalar variable. If the kind of the scalar number that we are multiplying is different (Double, Float, Long) to Integer, the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar multiplica(**Integer** in1). This method is used to multiply an Integer number to a scalar number. The result is stored in a scalar variable that could be the multiplicand or multiplier or another scalar variable. If the kind of the scalar number that we are multiplying is different (Double, Float, Long) to Integer, the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar multiplica(**long** lg2). This method is used to multiply a long number to a scalar number, this is realized making a Long scalar number from a long primitive type number. The result is stored in a scalar variable that could be the multiplicand or multiplier or another scalar variable. If the kind of the scalar number that we are multiplying is different (Double, Float, Integer) to Long, the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar multiplica(**Long** lg1). This method is used to multiply a Long number to a scalar number. The result is stored in a scalar variable that could be the multiplicand or multiplier or another scalar variable. If the kind of the scalar number that we are multiplying is different (Double, Float, Integer) to Long, the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public void numeroAleatorio(). This method generates a pseudo-random number. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. Note: The result is stored in the variable that call to this method.

public Escalar potencia(Escalar potencia). This method calculates a scalar number (the scalar number that call this method) raised to the power (the scalar number passed as parameter). This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar raizCuadrada(). This method obtain the square root of the scalar number that is calling to this method. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of

the scalar number that is calling to this method. This may produce loss of precision. The domain is (0,Inf] and the range is (0,Inf].

public Escalar redondea(). This method returns a rounded value of the scalar number that is calling to this method. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar redondea2(). This method returns a rounded value of the scalar number that is calling to this method. If the kind of the scalar number that is calling to this method is double it is converted to the kind long. If the kind of the scalar that is calling to this method is float it is converted to the kind int. Finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. Since floats and doubles cover numbers larger in magnitude than ints and longs, results are not correct for very large or very negative numbers.

public Escalar residuoRedondeado(Escalar denominador). This method calculates the remainder resulting from dividing two scalar numbers, using the IEEE 754 standard. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar residuoTruncado(Escalar denominador). This method calculates the remainder resulting from dividing two scalar numbers, using the operator modulus. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar resta(**double** db1). This method subtract a double number (subtrahend) from a scalar number (minuend), this is realized making a Double scalar number from a double primitive type number. The result is stored in a scalar variable that could be the minuend or another scalar variable. If the kind of the scalar number that we are subtracting is different (Float, Integer, Long) to Double, the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar resta(**Double** db1). This method subtract a Double number (subtrahend) from a scalar number (minuend). The result is stored in a scalar variable that could be the minuend or another scalar variable. If the kind of the scalar number that we are subtracting is different (Float, Integer, Long) to Double, the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar resta(Escalar escalar2). This method subtract two scalar numbers. The escalar2 is the subtrahend. The result is stored in a scalar variable that could be any of both minuend or subtrahend or another scalar variable. If the kind of the scalar numbers that we are subtracting are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public EscalarComplejo resta(EscalarComplejo escalarComplejo). This method subtract a scalar number (minuend) with a complex scalar number (subtrahend). The result is a complex scalar number. In this case the subtract is between the scalar number and the scalar real part of the scalar complex number. The kind (Double, Float, Long, Integer) of the scalar real part and the kind (Double, Float, Long, Integer) of the scalar imaginary part on the result are defined by this same parts of the scalar complex number on the argument (subtrahend). This may produce loss of precision.

public Escalar resta(**float** fl2). This method subtract a float number (subtrahend) from a scalar number (minuend), this is realized making a Float scalar number from a float primitive type number. The result is stored in a scalar variable that could be the minuend or another scalar variable. If the kind of the scalar number that we are subtracting is different (Double, Integer, Long) to Float, the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar resta(**Float** fl1). This method subtract a Float number (subtrahend) from a scalar number (minuend). The result is stored in a scalar variable that could be the minuend or another scalar variable. If the kind of the scalar number that we are subtracting is different (Double, Integer, Long) to Float, the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar resta(**int** in2). This method subtract a int number (subtrahend) from a scalar number (minuend), this is realized making a Integer scalar number from a int primitive type number. The result is stored in a scalar variable that could be the minuend or another scalar variable. If the kind of the scalar number that we are subtracting is different (Double, Float, Long) to Integer, the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar resta(**Integer** in1). This method subtract an Integer number (subtrahend) from a scalar number (minuend). The result is stored in a scalar variable that could be the minuend or another scalar variable. If the kind of the scalar

number that we are subtracting is different (Double, Float, Long) to Integer, the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar resta(**long** lg2). This method subtract a long number (subtrahend) from a scalar number (minuend), this is realized making a Long scalar number from a long primitive type number. The result is stored in a scalar variable that could be the minuend or another scalar variable. If the kind of the scalar number that we are adding is different (Double, Float, Integer) to Long, the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar resta(**Long** lg1). This method subtract a Long number (subtrahend) to a scalar number (minuend). The result is stored in a scalar variable that could be the minuend or another scalar variable. If the kind of the scalar number that we are subtracting is different (Double, Float, Integer) to Long, the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar seno(). This method calculates the sine of an angle. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. The domain is (-Inf , Inf) and the range of this function is (-1 , 1).

public Escalar suma(**double** db1). This method add a double number to a scalar number, this is realized making a Double scalar number from a double primitive type number. The result is stored in a scalar variable that could be the addends or another scalar variable. If the kind of the scalar number that we are adding is different (Float, Integer, Long) to Double, the numbers are converted to the kind of higher precision of both addends; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar suma(**Double** db1). This method add a Double number to a scalar number. The result is stored in a scalar variable that could be the addends or another scalar variable. If the kind of the scalar number that we are adding is different (Float, Integer, Long) to Double, the numbers are converted to the kind of higher precision of both addends; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar suma(Escalar escalar2). This method add two scalar numbers. The result is stored in a scalar variable that could be any of both addends or another scalar variable. If the kind of the scalars numbers that we are adding are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both

addends; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public EscalarComplejo suma(EscalarComplejo escalarComplejo). This method add a scalar number with a complex scalar number. The result is a complex scalar number. In this case the add is between the scalar number and the scalar real part of the scalar complex number. The kind (Double, Float, Long, Integer) of the scalar real part and the kind (Double, Float, Long, Integer) of the scalar imaginary part on the result are defined by this same parts of the scalar complex number on the argument. This may produce loss of precision.

public Escalar suma(**float** fl2). This method add a float number to a scalar number, this is realized making a Float scalar number from a float primitive type number. The result is stored in a scalar variable that could be the addend or another scalar variable. If the kind of the scalar number that we are adding is different (Double, Integer, Long) to Float, the numbers are converted to the kind of higher precision of both addends; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar suma(**Float** fl1). This method add a Float number to a scalar number. The result is stored in a scalar variable that could be the addend or another scalar variable. If the kind of the scalar number that we are adding is different (Double, Integer, Long) to Float, the numbers are converted to the kind of higher precision of both addends; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar suma(**int** in2). This method add a int number to a scalar number, this is realized making a Integer scalar number from a int primitive type number. The result is stored in a scalar variable that could be the addend or another scalar variable. If the kind of the scalar number that we are adding is different (Double, Float, Long) to Integer, the numbers are converted to the kind of higher precision of both addends; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar suma(**Integer** in1). This method add an Integer number to a scalar number. The result is stored in a scalar variable that could be the addend or another scalar variable. If the kind of the scalar number that we are adding is different (Double, Float, Long) to Integer, the numbers are converted to the kind of higher precision of both addends; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar suma(**long** lg2). This method add a long number to a scalar number, this is realized making a Long scalar number from a long primitive type number. The result is stored in a scalar variable that could be the addend or another scalar variable. If the kind of the scalar number that we are adding is different (Double, Float, Integer) to Long, the numbers are converted to the kind of higher precision of both

addends; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar suma(**Long** lg1). This method add a Long number to a scalar number. The result is stored in a scalar variable that could be the addend or another scalar variable. If the kind of the scalar number that we are adding is different (Double, Float, Integer) to Long, the numbers are converted to the kind of higher precision of both addends; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Escalar tangente(). This method calculates the cosine of an angle. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. The domain is (-Inf , Inf) with some values of discontinuity and the range of this function is (-Inf , Inf).

public Double tipoBaseDouble(). This method return the value of the instance variable: _SiEscalarDouble, of the class Escalar.

public Float tipoBaseFloat(). This method return the value of the instance variable: _SiEscalarFloat, of the class Escalar.

public Integer tipoBaseInteger(). This method return the value of the instance variable: _SiEscalarInteger, of the class Escalar.

public Long tipoBaseLong(). This method return the value of the instance variable: _SiEscalarLong, of the class Escalar.

public String toString(). We use this method in order to print the value of an scalar number to the screen. This method generates a representation of the scalar number.

public Escalar valorAbsoluto(). This method calculates the absolute value of a scalar number.

The class EscalarComplejo is used to define all operations involved with scalar complex numbers.

The *methods* of the class EscalarComplejo are:

public EscalarComplejo conjugado(). This method get the conjugate of the complex scalar number that is calling to this method.

public EscalarComplejo coseno(). This method calculates the cosine of the complex scalar number that is calling to this method. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the complex scalar number that is calling to this method. This may produce loss of precision. In general this operation return a complex scalar number.

public EscalarComplejo divide(Escalar escalar). This method is used to divide a complex scalar number (that is calling to this method) dividend with a scalar number (the argument: escalar) divisor. The parameter: escalar is the divisor. The result is stored in a complex scalar variable that could be the dividend or another complex scalar variable. If the kind of the scalar numbers that we are dividing are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the complex scalar number that is calling to this method. This may produce loss of precision.

public EscalarComplejo divide(EscalarComplejo escalarComplejo). This method is used to divide an complex scalar number (that is calling to this method) dividend with an complex scalar number (the argument: escalarComplejo) divisor. The result is a complex scalar number. The kind (Double, Float, Long, Integer) of the scalar real part and the kind (Double, Float, Long, Integer) of the scalar imaginary part on the result are defined by this same parts of the complex scalar number that is calling to this method (dividend). This may produce loss of precision.

public EscalarComplejo elevaAlcuadrado(). This method get the square of the complex scalar number that is calling to this method.

public EscalarComplejo(**Class** clase). This constructor uses an kind of class to build an complex scalar of that type (both real and imaginary part). The possible kinds of class are: Double, Integer, Float, Long. The initialization is always to zero.

public EscalarComplejo(**double** real, **double** imaginario). This constructor uses a double primitive type number as the real part and a double primitive type number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**Double** real, **double** imaginario). This constructor uses a Double number as the real part and a double primitive type number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**double** real, **Double** imaginario). This constructor uses a double primitive type number as the real part and a Double number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**Double** real, **Double** imaginario). This constructor uses a Double number as the real part and a Double number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**double** real, Escalar imaginario). This constructor uses a double primitive type number as the real part and a scalar number as the imaginary part. The real part is converted to a scalar number of kind Double.

public EscalarComplejo(**Double** real, Escalar imaginario). This constructor uses a Double number as the real part and a scalar number as the imaginary part. The real part is converted to a scalar number of kind Double.

public EscalarComplejo(**double** real, **float** imaginario). This constructor uses a double primitive type number as the real part and a float primitive type number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**Double** real, **float** imaginario). This constructor uses a Double number as the real part and a float primitive type number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**double** real, **Float** imaginario). This constructor uses a double primitive type number as the real part and a Float number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**Double** real, **Float** imaginario). This constructor uses a Double number as the real part and a Float number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**double** real, **int** imaginario). This constructor uses a double primitive type number as the real part and a int primitive type number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**Double** real, **int** imaginario). This constructor uses a Double number as the real part and a int primitive type number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**double** real, **Integer** imaginario). This constructor uses a double primitive type number as the real part and a Integer number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**Double** real, **Integer** imaginario). This constructor uses a Double number as the real part and a Integer number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**double** real, **long** imaginario). This constructor uses a double primitive type number as the real part and a long primitive type number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**Double** real, **long** imaginario). This constructor uses a Double number as the real part and a long primitive type number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**double** real, **Long** imaginario). This constructor uses a double primitive type number as the real part and a Long number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**Double** real, **Long** imaginario). This constructor uses a Double number as the real part and a Long number as the imaginary part. The real part is converted to a scalar number of kind Double and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(Escalar real, **double** imaginario). This constructor uses a scalar number as the real part and a double primitive type number as the imaginary part. The imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(Escalar real, **Double** imaginario). This constructor uses a scalar number as the real part and a Double number as the imaginary part. The imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(Escalar real, Escalar imaginario). This constructor uses two scalar numbers, one of them define the real part, the other one define the imaginary part.

public EscalarComplejo(Escalar real, **float** imaginario). This constructor uses a scalar number as the real part and a float primitive type number as the imaginary part. The imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo**(**Escalar real, **Float** imaginario). This constructor uses a scalar number as the real part and a Float number as the imaginary part. The imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(Escalar real, **int** imaginario). This constructor uses a scalar number as the real part and a int primitive type number as the imaginary part. The imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(Escalar real, **Integer** imaginario). This constructor uses a scalar number as the real part and a Integer number as the imaginary part. The imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(Escalar real, **long** imaginario). This constructor uses a scalar number as the real part and a long primitive type number as the imaginary part. The imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(Escalar real, **Long** imaginario). This constructor uses a scalar number as the real part and a Long number as the imaginary part. The imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**float** real, **double** imaginario). This constructor uses a float primitive type number as the real part and a double primitive type number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**Float** real, **double** imaginario). This constructor uses a Float number as the real part and a double primitive type number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**float** real, **Double** imaginario). This constructor uses a float primitive type number as the real part and a Double number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**Float** real, **Double** imaginario). This constructor uses a Float number as the real part and a Double number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**float** real, Escalar imaginario). This constructor uses a float primitive type number as the real part and a scalar number as the imaginary part. The real part is converted to a scalar number of kind Float.

public EscalarComplejo(**Float** real, Escalar imaginario). This constructor uses a Float number as the real part and a scalar number as the imaginary part. The real part is converted to a scalar number of kind Float.

public EscalarComplejo(**float** real, **float** imaginario). This constructor uses a float primitive type number as the real part and a float primitive type

number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**Float** real, **float** imaginario). This constructor uses a Float number as the real part and a float primitive type number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**float** real, **Float** imaginario). This constructor uses a float primitive type number as the real part and a Float number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**Float** real, **Float** imaginario). This constructor uses a Float number as the real part and a Float number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**float** real, **int** imaginario). This constructor uses a float primitive type number as the real part and a int primitive type number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**Float** real, **int** imaginario). This constructor uses a Float number as the real part and a int primitive type number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**float** real, **Integer** imaginario). This constructor uses a float primitive type number as the real part and a Integer number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**Float** real, **Integer** imaginario). This constructor uses a Float number as the real part and a Integer number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo**(float** real, **long** imaginario). This constructor uses a float primitive type number as the real part and a long primitive type number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**Float** real, **long** imaginario). This constructor uses a Float number as the real part and a long primitive type number as the

imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**float** real, **Long** imaginario). This constructor uses a float primitive type number as the real part and a Long number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**Float** real, **Long** imaginario). This constructor uses a Float number as the real part and a Long number as the imaginary part. The real part is converted to a scalar number of kind Float and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**int** real, **double** imaginario). This constructor uses a int primitive type number as the real part and a double primitive type number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**Integer** real, **double** imaginario). This constructor uses a Integer number as the real part and a double primitive type number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**int** real, **Double** imaginario). This constructor uses a int primitive type number as the real part and a Double number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**Integer** real, **Double** imaginario). This constructor uses a Integer number as the real part and a Double number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**int** real, Escalar imaginario). This constructor uses a int primitive type number as the real part and a scalar number as the imaginary part. The real part is converted to a scalar number of kind Integer.

public EscalarComplejo(**Integer** real, Escalar imaginario). This constructor uses a Integer number as the real part and a scalar number as the imaginary part. The real part is converted to a scalar number of kind Integer.

public EscalarComplejo(**int** real, **float** imaginario). This constructor uses a int primitive type number as the real part and a float primitive type number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**Integer** real, **float** imaginario). This constructor uses a Integer number as the real part and a float primitive type number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**int** real, **Float** imaginario). This constructor uses a int primitive type number as the real part and a Float number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**Integer** real, **Float** imaginario). This constructor uses a Integer number as the real part and a Float number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**int** real, **int** imaginario). This constructor uses a int primitive type number as the real part and a int primitive type number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**Integer** real, **int** imaginario). This constructor uses a Integer number as the real part and a int primitive type number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**int** real, **Integer** imaginario). This constructor uses a int primitive type number as the real part and a Integer number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**Integer** real, **Integer** imaginario). This constructor uses a Integer number as the real part and a Integer number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**int** real, **long** imaginario). This constructor uses a int primitive type number as the real part and a long primitive type number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**Integer** real, **long** imaginario). This constructor uses a Integer number as the real part and a long primitive type number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**int** real, **Long** imaginario). This constructor uses a int primitive type number as the real part and a Long number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**Integer** real, **Long** imaginario). This constructor uses a Integer number as the real part and a Long number as the imaginary part. The real part is converted to a scalar number of kind Integer and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**long** real, **double** imaginario). This constructor uses a long primitive type number as the real part and a double primitive type number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**Long** real, **double** imaginario). This constructor uses a Long number as the real part and a double primitive type number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**long** real, **Double** imaginario). This constructor uses a long primitive type number as the real part and a Double number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**Long** real, **Double** imaginario). This constructor uses a Long number as the real part and a Double number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Double.

public EscalarComplejo(**long** real, Escalar imaginario). This constructor uses a long primitive type number as the real part and a scalar number as the imaginary part. The real part is converted to a scalar number of kind Long.

public EscalarComplejo(**Long** real, Escalar imaginario). This constructor uses a Long number as the real part and a scalar number as the imaginary part. The real part is converted to a scalar number of kind Long.

public EscalarComplejo(**long** real, **float** imaginario). This constructor uses a long primitive type number as the real part and a float primitive type number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**Long** real, **float** imaginario). This constructor uses a Long number as the real part and a float primitive type number as the

imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**long** real, **Float** imaginario). This constructor uses a long primitive type number as the real part and a Float number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**Long** real, **Float** imaginario). This constructor uses a Long number as the real part and a Float number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Float.

public EscalarComplejo(**long** real, **int** imaginario). This constructor uses a long primitive type number as the real part and a int primitive type number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**Long** real, **int** imaginario). This constructor uses a Long number as the real part and a int primitive type number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**long** real, **Integer** imaginario). This constructor uses a long primitive type number as the real part and a Integer number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**Long** real, **Integer** imaginario). This constructor uses a Long number as the real part and a Integer number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Integer.

public EscalarComplejo(**long** real, **long** imaginario). This constructor uses a long primitive type number as the real part and a long primitive type number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**Long** real, **long** imaginario). This constructor uses a Long number as the real part and a long primitive type number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**long** real, **Long** imaginario). This constructor uses a long primitive type number as the real part and a Long number as the

imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Long.

public EscalarComplejo(**Long** real, **Long** imaginario). This constructor uses a Long number as the real part and a Long number as the imaginary part. The real part is converted to a scalar number of kind Long and the imaginary part is converted to a scalar number of kind Long.

public Escalar fase(). This method return the phase (a scalar number) of the complex number that is calling to this method. The range of the function is from -π to π. If the kind (Double, Float, etc...) of the real part is different from the kind of the imaginary part, then the numbers are converted to the kind of higher precision of both real part and imaginary part; finally the result is converted to the kind of the real part of the complex scalar number that is calling to this method. This may produce loss of precision.

public boolean igual(EscalarComplejo otro). This method return true if the value of both complex scalar numbers (actual instance and the parameter) are the same, otherwise return false. If its real and imaginary parts on both complex scalar numbers are the same then the method return true. If the kind of the parts (real and imaginary) in the complex scalar numbers are different (Double, Float, Integer, Long), they both are converted to equivalently compare them to the kind of the instance that call to this method. This can produce erroneous comparison. Caution must be observed due to this process of conversion some loss of precision is reached.

public Escalar magnitud(). This method return the magnitude (a scalar number) of the complex number that is calling to this method. If the kind (Double, Float, etc...) of the real part is different from the kind of the imaginary part, then the numbers are converted to the kind of higher precision of both real part and imaginary part; finally the result is converted to the kind of the real part of the complex scalar number that is calling to this method. This may produce loss of precision.

public EscalarComplejo multiplica(Escalar escalar). This method is used to multiply a complex scalar number (multiplicand) and a scalar number (multiplier). The result is stored in a scalar variable that could be the multiplicand (complex scalar number that call to this method) or another complex scalar variable. If the kind of the scalars numbers that we are multiplying are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the complex scalar number that is calling to this method. This may produce loss of precision.

public EscalarComplejo multiplica(EscalarComplejo escalarComplejo). This method is used to multiply an scalar number with an complex scalar number. The result is a complex scalar number. The kind (Double, Float, Long, Integer) of the scalar real part and the kind (Double, Float, Long, Integer) of the scalar imaginary part on the result are defined by this same parts of the complex scalar number on the argument. This may produce loss of precision.

public EscalarComplejo negativo(). This method get the negative value of the complex scalar number that is calling to this method.

public Escalar parteImaginaria(). This method return the imaginary part (a scalar number) of the complex number that is calling to this method.

public Escalar parteReal(). This method return the real part (a scalar number) of the complex number that is calling to this method.

public EscalarComplejo resta(Escalar escalar). This method subtract a scalar number (the parameter: escalar) from a complex scalar number minuend (that is calling this method). The escalar is the subtrahend. The result is stored in a complex scalar variable that could be the minuend or another complex scalar variable. If the kind of the scalar numbers that we are subtracting are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally the result is converted to the kind of the complex scalar number that is calling to this method. This may produce loss of precision.

public EscalarComplejo resta(EscalarComplejo escalarComplejo). This method subtract a complex scalar number (that is calling to this method: minuend) with an complex scalar number (the parameter: subtrahend). The result is a complex scalar number. The kind (Double, Float, Long, Integer) of the scalar real part and the kind (Double, Float, Long, Integer) of the scalar imaginary part on the result are defined by this same parts of the complex scalar number that is calling to this method (minuend). This may produce loss of precision.

public EscalarComplejo seno(). This method calculates the sine of the complex scalar number that is calling to this method. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the complex scalar number that is calling to this method. This may produce loss of precision. In general this operation return a complex scalar number.

public EscalarComplejo suma(Escalar escalar). This method add a complex scalar number (that is calling this method) and a scalar number (the parameter: escalar). The result is stored in a complex scalar variable that could be the complex scalar addend or another complex scalar variable. If the kind of the scalars numbers that we are adding are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both addends; finally the result is converted to the kind of the complex scalar number that is calling to this method. This may produce loss of precision.

public EscalarComplejo suma(EscalarComplejo escalarComplejo). This method add a complex scalar number (that is calling to this method) with a complex scalar number (the parameter: escalarComplejo). The result is a complex scalar number. The kind (Double, Float, Long, Integer) of the scalar real part and

the kind (Double, Float, Long, Integer) of the scalar imaginary part on the result are defined by this same parts of the complex scalar number that is calling to this method. This may produce loss of precision.

public EscalarComplejo tangente(). This method calculates the tangent of the complex scalar number that is calling to this method. This operation always convert to double precision in order to implement this operation; finally the result is converted to the kind of the complex scalar number that is calling to this method. This may produce loss of precision. In general this operation return a complex scalar number.

public String toString(). This method is used to print the value of an complex scalar number to the screen. This method generates a representation of the complex scalar number.

The class Vector is used to define all operations involved with vectors. The vectors are build as double linked list of scalar or complex scalar numbers. This class inherit all the attributes and behaviors of the class TipoList0. The class TipoList0 inherits all the attributes and behaviors of the class List0.

The *methods* of the class Vector are:

public Vector absoluto(**String** cadena). This method calculates the absolute value of each element in a vector of type "miguel.math.Escalar", the parameter (cadena) can be any String it doesn't matter (don't care). If the vector is of type "miguel.math.EscalarComplejo" the String indicates to which part of each complex scalar number in the vector will be calculated the absolute value: "parteReal", "parteImaginaria", "ambas". If the strings are not written exactly as was indicated a clon of the actual vector (actual instance that call to this method) is returned.

public Vector aComplejo(Vector vector2). This method is used to conform a vector of type EscalarComplejo from two vectors of type Escalar, the actual instance that call to this method is the real part and the parameter (vector2) is the imaginary part of the complex scalar vector. If the type of both vectors are not Escalar or both vector have different size, a null or empty vector of type EscalarComplejo is returned.

public void añadeTamaño(**int** Tamaño). This method is used to increase in Tamaño the quantity of scalar numbers on the vector. The initial value of all scalar numbers is zero. This method uses another method añadeElemento which in the class TipoList0 verifies that the new element appended must be the same type that the type declared for the vector. This type can be: miguel.math.Escalar. The kind (Double, Float, Integer Long) of the scalar numbers in the vector can be different

public void añadeTamaño(**int** Tamaño, Escalar valorInicial). This method is used to increase in Tamaño, quantity of scalar numbers on the vector. The initial value of all the scalar numbers is valorInicial. This

method uses another method añadeElemento which in the class TipoList0 verifies that the new element appended must be the same type that the type declared for the vector. This type can be: miguel.math.Escalar

public void añadeTamaño(**int** Tamaño, EscalarComplejo valorInicial). This method is used to increase in Tamaño the quantity of complex scalar numbers on the vector. The initial value of all the complex scalar numbers is valorInicial. This method uses another method añadeElemento which in the class TipoList0 verifies that the new element appended must be the same type that the type declared for the vector. This type can be: miguel.math.EscalarComplejo

public synchronized Vector clon(). This method duplicates a vector in other memory location.

public Vector clonar(**String** str). This method takes one of this two String arguments "parteReal" or "parteImaginaria", all the elements in the vector are the same type: miguel.math.EscalarComplejo. If the argument is the String "parteReal" this method duplicates the real part of the complex scalar numbers on the vector storing them in another vector of scalar numbers. If the argument is the String "parteImaginaria" this method duplicates the imaginary part of the complex scalar numbers on the vector storing them in another vector of scalar numbers. If the vector that call this method has the type: miguel.math.Escalar this method works like the method clon( ). If the parameter is wrong written this method works like the method clon().

public Vector concatena(Vector vector2). This method link together two vector. The parameter (vector2) is append to the vector (actual instance) that call to this method. Both vector must be the same type (miguel.math.Escalar or miguel.math.EscalarComplejo), if they are not the same type the actual instance that call to this method is returned the same.

public Vector copia(Vector vector2). This method is used to copy the parameter (vector2) on the actual vector (actual instance that call to this method), from the beginning position (the first element). If vector2 is greater than the actual vector, then the elements out of the bound of actual instance are not appended to the resultant vector. Both vector2 and actual vector must be the same type (miguel.math.Escalar or miguel.math.EscalarComplejo), if do not the resultant vector is returned equal to actual vector.

public Vector copiaANDen(Vector vector2, **int** indice). This method is used to copy the parameter (vector2) on the actual vector (actual instance that call to this method), from the position given by the parameter indice. If indice is a negative position greater in magnitude than the size of vector2 or greater than the size of the actual instance that call to this method, then the resultant vector is a null vector (empty vector). If indice is a position into the actual instance, vector2 replace from this position the elements of the actual instance with its own elements. If vector2 is greater than the actual vector, then the elements out of the bound of actual instance are not appended to the

resultant vector. Both vector2 and actual vector must be the same type (miguel.math.Escalar or miguel.math.EscalarComplejo), if do not the resultant vector is returned equal to actual vector.

public Vector copiaEn(Vector vector2, **int** indice). This method is used to copy the parameter (vector2) on the actual vector (actual instance that call to this method), from the position given by the parameter indice. If indice is a negative position greater in magnitude than the size of vector2 or greater than the size of the actual instance that call to this method, then the resultant vector is equal to actual vector. If indice is a position into the actual instance, vector2 replace from this position the elements of the actual instance with its own elements. If vector2 is greater than the actual vector, then the elements out of the bound of actual instance are not appended to the resultant vector. Both vector2 and actual vector must be the same type (miguel.math.Escalar or miguel.math.EscalarComplejo), if do not the resultant vector is returned equal to actual vector.

public Vector copiaORen(Vector vector2, **int** indice). This method is used to copy the parameter (vector2) on the actual vector (actual instance that call to this method), from the position given by the parameter indice. If indice is a negative position greater in magnitude than the size of vector2 or greater than the size of the actual instance that call to this method, then vector2 is append at the beginning or at the end of vector2 respectively on the resultant vector. If indice is a position into the actual instance, vector2 replace from this position the elements of the actual instance with its own elements. If vector2 is greater than the actual vector, then the elements out of the bound of actual instance are appended to the resultant vector. Both vector2 and actual vector must be the same type (miguel.math.Escalar or miguel.math.EscalarComplejo), if do not the resultant vector is returned equal to actual vector.

public Vector divide(Escalar escalar). This method is used to divide a vector (each number in the vector is a dividend) with a scalar number (divisor: escalar). The result is stored in a vector. The vector could be of type miguel.math.Escalar or miguel.math.EscalarComplejo. If the type vector is miguel.math.Escalar and the kind of the scalar numbers that we are dividing are different (Double, Float, Integer, Long) the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. If the type vector is miguel.math.EscalarComplejo and the kind of real part and imaginary part of the complex scalar numbers that we are dividing are different (Double, Float, Integer, Long) each part of the complex scalar numbers (real and imaginary part) are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Vector divide(EscalarComplejo escalarComplejo). This method is used to divide a vector (each number in the vector is a dividend) with a complex scalar number (divisor: escalar). The result is stored in a vector. The vector could be of type miguel.math.Escalar or miguel.math.EscalarComplejo. If the type vector is

miguel.math.Escalar and the kind of the scalar numbers that we are dividing are different (Double, Float, Integer, Long) the numbers are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the complex scalar number (escalarComplejo). This may produce loss of precision. If the type vector is miguel.math.EscalarComplejo and the kind of real part and imaginary part of the complex scalar numbers that we are dividing are different (Double, Float, Integer, Long) each part of the complex scalar numbers (real and imaginary part) are converted to the kind of higher precision of both dividend or divisor; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Vector fase(). This method evaluate the phase of each element in the vector. The vector must be of type EscalarComplejo, if does not return the vector of the actual instance that call to this method. NOTE: this operation does not is defined in vector mathematics, it is implemented for computational efficiency, which means save time and memory space.

public boolean insertaEn(Vector vector2, **int** indice). This method is used to insert the parameter (vector2) on the actual vector (actual instance that call to this method), in the position given by the parameter indice. If indice is a negative position or greater than the size of the actual instance that call to this method, then the method return false and nothing is inserted to the actual vector. If indice is a position into the actual instance, insert the elements of vector2 in that position. Both vector2 and actual vector must be the same type (miguel.math.Escalar or miguel.math.EscalarComplejo), if do not the method return false and nothing is inserted to the actual vector.

public Vector magnitud(). This method evaluate the magnitude of each element in the vector. The vector must be of type EscalarComplejo, if does not return the vector of the actual instance that call to this method. NOTE: this operation does not is defined in vector mathematics, it is implemented for computational efficiency, which means save time and memory space.

public boolean mismoTamañoVectores(Vector vector2). This method compares the size of two vectors. It does not matter if they are of different type (miguel.math.Escalar or miguel.math.EscalarComplejo). Return true if they both are the same size otherwise return false.

public boolean mismoTipoVectores(Vector vector2). This method compares two vectors. Return true if both vectors are the same type (miguel.math.Escalar or miguel.math.EscalarComplejo).

public Vector multiplica(Escalar escalar). This method is used to multiply a vector (each number in the vector is a multiplicand) with a scalar number (multiplier: escalar). The result is stored in a vector. The vector could be of type miguel.math.Escalar or miguel.math.EscalarComplejo. If the type vector is miguel.math.Escalar and the kind of the scalar numbers that we are multiplying are

different (Double, Float, Integer, Long) the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision. If the type vector is miguel.math.EscalarComplejo and the kind of real part and imaginary part of the complex scalar numbers that we are multiplying are different (Double, Float, Integer, Long) each part of the complex scalar numbers (real and imaginary part) are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Vector multiplica(EscalarComplejo escalarComplejo). This method is used to multiply a vector (each number in the vector is a multiplicand) with a complex scalar number (multiplier: escalarComplejo). The result is stored in a vector. The vector could be of type miguel.math.Escalar or miguel.math.EscalarComplejo. If the type vector is miguel.math.Escalar and the kind of the scalar numbers that we are multiplying are different (Double, Float, Integer, Long) the numbers are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the complex scalar number (escalarComplejo). This may produce loss of precision. If the type vector is miguel.math.EscalarComplejo and the kind of real part and imaginary part of the complex scalar numbers that we are multiplying are different (Double, Float, Integer, Long) each part of the complex scalar numbers (real and imaginary part) are converted to the kind of higher precision of both multiplicand or multiplier; finally the result is converted to the kind of the scalar number that is calling to this method. This may produce loss of precision.

public Object multiplica(Vector vector2). This method is used to multiply the actual vector (actual instance that call to this method) and the vector parameter vector2. Both vectors must be the same size, if do not return zero. The type (miguel.math.Escalar or miguel.math.EscalarComplejo) can be different of both vectors. This operation multiply each element (multiplicand) of actual vector with the corresponding element (multiplier) in the vector vector2. The result is a scalar or complex scalar number. If the kind of the elements that we are multiplying are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both multiplicand or multiplier; each multiply result is converted to the kind of each element of the actual vector that is calling to this method finally all this multiply result are added to give an scalar or complex scalar number. This may produce loss of precision.

public Vector recorta(**String** cadena, Escalar maximo,Escalar minimo). This method clip between a maximum and a minimum for each element in a vector of type "miguel.math.Escalar", the parameter (cadena) can be any String it doesn't matter (don't care). The parameters maximo and minimo specify the extremes maximum and minimum of the clipping respectively. The kinds (Double, Float, Long and Integer) of the scalar values maximo and minimo are converted to the kinds of each element in the actual vector (actual instance that call to this method) to compare them. If the value in the vector is higher that maximo then the value is replaced by maximo in the vector. If the value in the vector is lower that minimo then minimo replace the value of that element in the vector. If the vector is of type "miguel.math.EscalarComplejo" the String

indicates to which part of each complex scalar number in the vector will be clipped as was indicated before, the value of the String must be exact as is indicated: "parteReal", "parteImaginaria", "ambas". If the strings are not written exactly as was indicated a clon of the actual vector (actual instance that call to this method) is returned. This method work as an operator of clipping on the vector in order to confine all the values in the vector between the values maximo and minimo.

public Vector resta(Vector vector2). This method subtract the actual vector (actual instance that call to this method) with the vector parameter vector2. Both vectors must be the same size, if do not return the actual vector. The type (miguel.math.Escalar or miguel.math.EscalarComplejo) can be different of both vectors. This operation subtract each element (minuend) of actual vector with the corresponding element (subtrahend) in the vector vector2. The result is a vector with the same size of their addends. If the kind of the elements that we are subtracting are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both minuend or subtrahend; finally each subtract is converted to the kind of each element of the actual vector that is calling to this method. This may produce loss of precision.

public void revierte(). This method put in reverse order the elements of the vector. It is the first element in the vector (actual instance that call to this method) becomes to be the last one and the last element becomes to be the first.

public Vector subVector(**int** indiceinicio, **int** indicefin). This method is used to obtain a sub-vector from an actual vector (actual instance that call to this method). Starting in the number of element given by the parameter indiceinicio and ending in the number of element given by the parameter indicefin. If the values of indiceinicio and indicefin are out of the bounds of the actual vector then the extremes of the sub-vector are defined by the start (position zero) and the end (size of the actual vector minus one) of the actual vector respectively. If indiceinicio > inidicefin then a null sub-vector (empty sub-vector) is returned.

public Vector suma(Escalar escalar). This method add the actual vector (actual instance that call to this method) and the parameter escalar. The actual vector can be of type (miguel.math.Escalar or miguel.math.EscalarComplejo). If the vector is the type miguel.math.EscalarComplejo then the scalar number is added to the real part of each complex scalar number in the vector. If the kind of the elements that we are adding are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both addends; finally each add result is converted to the kind of each element of the actual vector that is calling to this method. NOTE: this operation does not is defined in vector mathematics, it is implemented for computational efficiency, which means save time and memory space.

public Vector suma(EscalarComplejo escalarcomplejo). This method add the actual vector (actual instance that call to this method) and the parameter escalarcomplejo. The actual vector can be of type (miguel.math.Escalar or miguel.math.EscalarComplejo). If the vector is the type miguel.math.Escalar then each

scalar number in the vector is added to the real part of the complex scalar number. If the kind of the elements that we are adding are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both addends; finally each add result is converted to the kind of real part and imaginary part of the each element of the actual vector that is calling to this method if its type is miguel.math.EscalarComplejo or each add result is converted to the kind of real part and imaginary part of the complex scalar number (the parameter) if the type of the vector that is calling to this method is miguel.math.Escalar. NOTE: this operation does not is defined in vector mathematics, it is implemented for computational efficiency, which means save time and memory space.

public Vector suma(Vector vector2). This method add the actual vector (actual instance that call to this method) and the vector parameter vector2. Both vectors must be the same size, if do not return the actual vector. The type (miguel.math.Escalar or miguel.math.EscalarComplejo) can be different of both vectors. This operation add each element of actual vector with the corresponding element in the vector vector2. The result is a vector with the same size of their addends vectors. If the kind of the elements that we are adding are different (Double, Float, etc...) the numbers are converted to the kind of higher precision of both addends; finally each add result is converted to the kind of each element of the actual vector that is calling to this method. This may produce loss of precision.

public Vector(**int** tam, Escalar escalar). This constructor build a vector with tam (tam: defines the number of scalars numbers) scalar numbers. All the scalar numbers are initialized to the scalar number: escalar.

public Vector(**int** tam, EscalarComplejo escalarComplejo). This constructor build a vector with tam (tam: defines the number of complex scalar numbers) complex scalar numbers. All the complex scalar numbers are initialized to the complex scalar number: escalarComplejo.

public Vector(**String** str). This constructor is used to build an empty Vector with a type of the vector that can be: "miguel.math.Escalar", "miguel.math.EscalarComplejo", "java.lang.Number". Does not recommended use the type "java.lang.Number" because all methods implemented in class Vector does not use it.

Implementación (código) de las clases y métodos para Cálculos numéricos en Java.

Clases matemáticas implementadas en Java, deben encontrarse en el paquete: miguel.math

(Archivo: ElementoLista.java)

```
package miguel.math;
// ElementoLista es un Elemento de la lista doble enlazada
// esta contiene datos (tipo Objeto), una referencia al
// proximo elemento, y una referencia al anterior elemento
// aquí se definen los elementos de datos metodos para establecer
// y obtener cada elemento sobre cada uno así como un constructor

public class ElementoLista {
 // el corazón de los elementos (datos)
 Object _dato;

 // referencia al próximo elemento
 ElementoLista _enlace_proximo;

 // referencia al anterior elemento
 ElementoLista _enlace_anterior;

 // solo el constructor - requiere el dato y ambos enlaces especificados
 // puede usar null sobre cualquier elmento si se requiere o si todavía no se conoce
 public ElementoLista(Object dato, ElementoLista enlace_proximo, ElementoLista
enlace_anterior)
  {
   _dato = dato;
   _enlace_proximo = enlace_proximo;
   _enlace_anterior = enlace_anterior;
  }

// regresa el dato del elemento
public Object dato() {return _dato;}

// regresa referencia al próximo elemento
public ElementoLista enlace_proximo() {return _enlace_proximo;}

// regresa referencia al elemento anterior
public ElementoLista enlace_anterior() {return _enlace_anterior;}

// establece el dato del elemento
public void establece_dato(Object o) {_dato = o;}

// establece la referencia al próximo elemento
public void establece_enlace_proximo(ElementoLista enlace) {_enlace_proximo = enlace;}

// establece la referencia al elemento anterior
public void establece_enlace_anterior(ElementoLista enlace) {_enlace_anterior = enlace;}

}
```

(Archivo: List0.java)

```
package miguel.math;
// implementa una lista doble enlazada de cualquier tipo de Object

import miguel.math.ElementoLista;
import java.util.Vector;

public class List0 implements Cloneable{

 // referencia al comienzo de la lista, o null si la lista esta vacía
```

```
ElementoLista _cabeza;

// referencia al final de la lista, o null si la lista esta vacia
ElementoLista _cola;

// número de elmentos en la lista (por eficiencia)
int _cuenta;

// constructor por default de una lista vacía
public List0()
{
 eliminaTodosLosElementos();
}

// conveniente constructor - una lista de un arreglo
public List0(Object unArreglo[]){
 eliminaTodosLosElementos();
 int longitud = unArreglo.length;

 for(int i=0; i < longitud; i++)
  añadeElemento(unArreglo[i]);
}

// crea una nueva lista enlazada
public synchronized Object clona(){
 List0 resultado = new List0();
 ElementoLista elemento = _cabeza;
 for (int i=0; i<_cuenta; i++){
  resultado.añadeElemento(elemento.dato());
  elemento = elemento.enlace_proximo();
 }
 return resultado;
}

// regresa el numero de entradas en la lista
public int tamaño()
{
  return _cuenta;
}

// regresa true si un elemento es contenido en la lista
public boolean contiene(Object elem){
 return indiceDe(elem) >= 0;
}

// regresa el índice de un elemento particular, o -1 si no es encontrado
public int indiceDe(Object elem){
 return indiceDe(elem,0);
}

// regresa el índice de un elemento particular, buscando desde un índice de comienzo
// -1 si no es encontrado

public synchronized int indiceDe(Object elem, int comienzo)
{
 if (elem == null || comienzo >= _cuenta)
  return -1;

 // adelanta elementos hasta llegar a comienzo
 ElementoLista elemento = _cabeza;
 int i;
 for(i=0; i<comienzo; i++)
  elemento = elemento.enlace_proximo();

 for(; elemento != null; i++){
  if (elem.equals(elemento.dato()))
   return i;
  elemento = elemento.enlace_proximo();
 }
 return -1;
}
```

```
 // coloca un elemento al final de la lista
 public synchronized void añadeElemento(Object obj)
 {
  ElementoLista nuevoelemento = new ElementoLista(obj, null, _cola);
  if (_cola != null)
   _cola.establece_enlace_proximo(nuevoelemento); // del ex-ultimo elemento establece su
                                                  // enlace_proximo con el nuevoelemento
  _cola = nuevoelemento;     // haz la _cola igual al nuevoelemento añadido
  if (_cabeza == null)       // esto se debe efectuar si la lista esta empezando
   _cabeza = nuevoelemento;
  _cuenta++;
 }

 // coloca un elemento al comienzo de la lista
 public synchronized void principioElemento(Object obj)
 {
  ElementoLista nueva_entrada = new ElementoLista(obj, _cabeza, null);
  if (_cabeza != null)
   _cabeza.establece_enlace_anterior(nueva_entrada); // del ex-primer elemento establece su
                                                     // enlace_anterior con la nueva_entrada
   _cabeza = nueva_entrada;   // haz la _cabeza igual a la nueva_entrada insertada al inicio
   if (_cola == null)         // esto se debe efectuar si la lista esta empezando
    _cola = nueva_entrada;
   _cuenta++;
 }

 // elimina un elemento al comienzo de la lista
 public synchronized void EliminaprincipioElemento()
 {
  ElementoLista elem;
  if ((_cabeza != null) && (_cola != null)){
   if (_cabeza == _cola) eliminaTodosLosElementos();
   else {
    elem =_cabeza;
    _cabeza=_cabeza.enlace_proximo();
    _cabeza.establece_enlace_anterior(null);
    elem.establece_dato(null);
    elem.establece_enlace_proximo(null);
    elem.establece_enlace_anterior(null);
    _cuenta--;
   }
  }
 }

// elimina un elemento al final de la lista
 public synchronized void EliminafinElemento()
 {
  ElementoLista elem;
  if ((_cabeza != null) && (_cola != null)){
   if (_cabeza == _cola) eliminaTodosLosElementos();
   else {
    elem =_cola;
    _cola=_cola.enlace_anterior();
    _cola.establece_enlace_proximo(null);
    elem.establece_dato(null);
    elem.establece_enlace_proximo(null);
    elem.establece_enlace_anterior(null);
    _cuenta--;
   }
  }
 }

 // coloca un elemento en una localización en particular
 // regresa falso si la localización es negativa o demasiado grande
 // esto deberá probablemente arrojar una excepción en cambio
 public synchronized boolean insertaElementoEn(Object obj, int indice)
 {
  // maneja principioElemento como caso simple
  if (indice == 0){
   principioElemento(obj);
```

```
   return true;
  }

  // maneja añadeElemento al final de la lista como caso simple
  if (indice == _cuenta){
   añadeElemento(obj);
   return true;
  }

  // fuera de los límites de la lista, podría arrojar una excepción
  if (indice < 0 || indice > _cuenta){
   return false;
  }

  // encuentra localización y ajusta los punteros
  ElementoLista elemento = _cabeza.enlace_proximo();
  int i = 1;
  while (elemento != null){
   if (indice == i) {
    ElementoLista antes = elemento.enlace_anterior();
    ElementoLista nueva_entrada = new ElementoLista(obj, elemento, antes);
    elemento.establece_enlace_anterior(nueva_entrada);
    antes.establece_enlace_proximo(nueva_entrada);
    _cuenta++;
    return true;
   }
   elemento =elemento.enlace_proximo();
   i++;
  }
  // no debería llegar aquí, pero en su caso
  return false;
 }

 // elimina un elemento en una localización en particular
 // regresa falso si la localización es negativa o demasiado grande
 // esto deberá probablemente arrojar una excepción en cambio
 public synchronized boolean EliminaElementoEn(int indice)
 {
  // maneja EliminaprincipioElemento como caso simple
  if (indice == 0){
   EliminaprincipioElemento();
   return true;
  }

  // maneja EliminafinElemento al final de la lista como caso simple
  if (indice == _cuenta-1){
   EliminafinElemento();
   return true;
  }

  // fuera de los límites de la lista, podría arrojar una excepción
  if (indice < 0 || indice >= _cuenta){
   return false;
  }

  // encuentra localización y ajusta los punteros
  ElementoLista elemento = _cabeza.enlace_proximo();
  int i = 1;
  while (elemento != null){
   if (indice == i) {
    ElementoLista antes = elemento.enlace_anterior();
    ElementoLista despues = elemento.enlace_proximo();
    despues.establece_enlace_anterior(elemento.enlace_anterior());
    antes.establece_enlace_proximo(elemento.enlace_proximo());
    elemento.establece_dato(null);
    elemento.establece_enlace_proximo(null);
    elemento.establece_enlace_anterior(null);
    _cuenta--;
    return true;
   }
   elemento =elemento.enlace_proximo();
```

```
  i++;
 }
 // no debería llegar aquí, pero en su caso
 return false;
}

// reemplaza un elemento en una localización en particular
// regresa falso si la localización es negativa o demasiado grande
// esto deberá probablemente arrojar una excepción en cambio
public synchronized boolean ReemplazaElementoEn(Object obj, int indice)
{
 // fuera de los límites de la lista, podría arrojar una excepción
 if (indice < 0 || indice > _cuenta-1){
  return false;
 }

 if(indice == 0) {
  ElementoLista elemento = _cabeza;
  elemento.establece_dato(obj);
  return true;
 }

 // encuentra localización y ajusta los punteros
 ElementoLista elemento = _cabeza.enlace_proximo();
 int i = 1;
 while (elemento != null){
  if (indice == i) {
   elemento.establece_dato(obj);
   return true;
  }
  elemento =elemento.enlace_proximo();
  i++;
 }
 // no debería llegar aquí, pero en su caso
 return false;
}

// obten un elemento de una localización en particular
// regresa obj si la localización es negativa o demasiado grande
// esto deberá probablemente arrojar una excepción en cambio
public synchronized Object ObtenElementoDe(Object obj, int indice)
{
 // fuera de los límites de la lista, regresa obj (el elemento pasado como parámetro)
 if (indice < 0 || indice > _cuenta-1){
  return obj;
 }

 if(indice == 0) {
  ElementoLista elemento = _cabeza;
  obj = elemento.dato();
  return obj;
 }

 // encuentra localización y ajusta los punteros
 ElementoLista elemento = _cabeza.enlace_proximo();
 int i = 1;
 while (elemento != null){
  if (indice == i) {
   obj = elemento.dato();
   return obj;
  }
  elemento =elemento.enlace_proximo();
  i++;
 }
 // no debería llegar aquí, pero en su caso
 return obj;
}

// resetea la lista a vacio
// se supone ques esto disparará el colector de basura sobre los elementos descartados
public synchronized void eliminaTodosLosElementos()
```

```
 {
  _cabeza = null;
  _cola = null;
  _cuenta = 0;
 }

 // Representación en cadena de una lista, mediante el encadenamiento junto de la
representación
 // cadena de sus elementos
 public synchronized String toString()
 {
  String resultado = "";
  ElementoLista elemento = _cabeza;
  while (elemento != null){
   Object obj = elemento.dato();
   resultado = resultado + obj.toString() + "\n";
   elemento =elemento.enlace_proximo();
  }
  return resultado;
 }
}
```

(Archivo: TipoList0.java)

```
package miguel.math;
// Aumenta la clase List0 para restringir tipos permitidos en la lista
// Además de los contructores y los métodos de comprobación de tipos,
// todos los métodos de insersión de elementos en List0 are sobreescritos

import miguel.math.List0;

public class TipoList0 extends List0 {

 // manteniendo la clase aceptable
 Class _clase;

 // construye una lista vacia, con un tipo particular
 public TipoList0(Class clase_permitida)
 {
  super();
  _clase = clase_permitida;
 }

 // métodos para comprobar para un tipo válido y reclamo

 public synchronized void compruebaTipo(Object obj)
 {
  for(Class c = obj.getClass(); c != null; c = c.getSuperclass()){
   if(c.equals(_clase))
    return;
  }
  tipoReclamo(obj);
 }

 void tipoReclamo(Object obj)
 {
  throw new IllegalArgumentException(
            "No puedo añadir un objeto del tipo " +
             obj.getClass().getName()             +
            " a TipoList0 que es del tipo "       +
             _clase.getName());
 }

 // intercepta todos los métodos que añaden elementos a la lista, para comprobar su tipo

 // añade elemento al final de la lista
 public synchronized void añadeElemento(Object obj)
 {
  compruebaTipo(obj);
  super.añadeElemento(obj);
```

```
 }

 // inserta un elemento en un lugar particular en la lista
 public synchronized boolean insertaElementoEn(Object obj, int indice)
 {
  compruebaTipo(obj);
  return super.insertaElementoEn(obj, indice);
 }

 // coloca elemento al inicio de la lista
 public synchronized void principioElemento(Object obj)
 {
  compruebaTipo(obj);
  super.principioElemento(obj);
 }

 // reemplaza un elemento en una localización particular en la lista
 public synchronized boolean ReemplazaElementoEn(Object obj, int indice)
 {
  compruebaTipo(obj);
  return super.ReemplazaElementoEn(obj, indice);
 }

}
```

(Archivo: Escalar.java)

```
package miguel.math;
import miguel.math.EscalarComplejo;

public class Escalar{

 // tipo de clase original del que es el Escalar
 Class   _clase;
 Double  _SiEscalarDouble;
 Float   _SiEscalarFloat;
 Long    _SiEscalarLong;
 Integer _SiEscalarInteger;

 // constructores de la clase Escalar
 public Escalar(Double db1) {
  _clase = db1.getClass();
  _SiEscalarDouble  = db1;
  _SiEscalarFloat   = null;
  _SiEscalarLong    = null;
  _SiEscalarInteger = null;
 }
 public Escalar(Float fl1) {
  _clase = fl1.getClass();
  _SiEscalarDouble  = null;
  _SiEscalarFloat   = fl1;
  _SiEscalarLong    = null;
  _SiEscalarInteger = null;
 }
 public Escalar(Long lg1) {
  _clase = lg1.getClass();
  _SiEscalarDouble  = null;
  _SiEscalarFloat   = null;
  _SiEscalarLong    = lg1;
  _SiEscalarInteger = null;
 }
 public Escalar(Integer in1) {
  _clase = in1.getClass();
  _SiEscalarDouble  = null;
  _SiEscalarFloat   = null;
  _SiEscalarLong    = null;
  _SiEscalarInteger = in1;
 }
 public Escalar(double db2) {
  _SiEscalarDouble  = new Double (db2);
```

```
 _SiEscalarFloat   = null;
 _SiEscalarLong    = null;
 _SiEscalarInteger = null;
 _clase = _SiEscalarDouble.getClass();
}
public Escalar(float fl2) {
 _SiEscalarDouble  = null;
 _SiEscalarFloat = new Float (fl2);
 _SiEscalarLong    = null;
 _SiEscalarInteger = null;
 _clase = _SiEscalarFloat.getClass();
}
public Escalar(long lg2) {
 _SiEscalarDouble  = null;
 _SiEscalarFloat   = null;
 _SiEscalarLong = new Long(lg2);
 _SiEscalarInteger = null;
 _clase = _SiEscalarLong.getClass();
}
public Escalar(int in2) {
 _SiEscalarDouble  = null;
 _SiEscalarFloat   = null;
 _SiEscalarLong    = null;
 _SiEscalarInteger = new Integer(in2);
 _clase = _SiEscalarInteger.getClass();
}
public Escalar(Class clase){
 Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
 Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
 if (clase.equals(tipo1.getClass())) {
  this._clase = tipo1.getClass();
  _SiEscalarDouble = new Double(0.);
  _SiEscalarFloat   = null;
  _SiEscalarLong    = null;
  _SiEscalarInteger = null;
 } else if (clase.equals(tipo2.getClass())) {
  this._clase = tipo2.getClass();
  _SiEscalarDouble  = null;
  _SiEscalarFloat = new Float(0.);
  _SiEscalarLong    = null;
  _SiEscalarInteger = null;
 } else if (clase.equals(tipo3.getClass())) {
  this._clase = tipo3.getClass();
  _SiEscalarDouble  = null;
  _SiEscalarFloat   = null;
  _SiEscalarLong = new Long(0);
  _SiEscalarInteger = null;
 } else if (clase.equals(tipo4.getClass())) {
  this._clase = tipo4.getClass();
  _SiEscalarDouble  = null;
  _SiEscalarFloat   = null;
  _SiEscalarLong    = null;
  _SiEscalarInteger = new Integer(0);
 }
}

// convierte el escalar de un tipo a otro tipo
public Escalar convierteTipoEscalar(Class clase){
 Escalar escalar = new Escalar(this._clase);
 Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
 Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
 if (this._clase.equals(tipo1.getClass())){
  if (clase.equals(tipo1.getClass()))
   {escalar = new Escalar(this._SiEscalarDouble);}
  if (clase.equals(tipo2.getClass()))
   {escalar = new Escalar(new Float((float)(this._SiEscalarDouble.doubleValue())));}
  if (clase.equals(tipo3.getClass()))
   {escalar = new Escalar(new Long((long)(this._SiEscalarDouble.doubleValue())));}
  if (clase.equals(tipo4.getClass()))
   {escalar = new Escalar(new Integer((int)(this._SiEscalarDouble.doubleValue())));}
 } else if (this._clase.equals(tipo2.getClass())) {
```

```
   if (clase.equals(tipo1.getClass()))
    {escalar = new Escalar(new Double((double)(this._SiEscalarFloat.floatValue())));}
   if (clase.equals(tipo2.getClass()))
    {escalar = new Escalar(this._SiEscalarFloat);}
   if (clase.equals(tipo3.getClass()))
    {escalar = new Escalar(new Long((long)(this._SiEscalarFloat.floatValue())));}
   if (clase.equals(tipo4.getClass()))
    {escalar = new Escalar(new Integer((int)(this._SiEscalarFloat.floatValue())));}
  } else if (this._clase.equals(tipo3.getClass())) {
   if (clase.equals(tipo1.getClass()))
    {escalar = new Escalar(new Double((double)(this._SiEscalarLong.longValue())));}
   if (clase.equals(tipo2.getClass()))
    {escalar = new Escalar(new Float((float)(this._SiEscalarLong.longValue())));}
   if (clase.equals(tipo3.getClass()))
    {escalar = new Escalar(this._SiEscalarLong);}
   if (clase.equals(tipo4.getClass()))
    {escalar = new Escalar(new Integer((int)(this._SiEscalarLong.longValue())));}
  } else if (this._clase.equals(tipo4.getClass())) {
   if (clase.equals(tipo1.getClass()))
    {escalar = new Escalar(new Double((double)(this._SiEscalarInteger.intValue())));}
   if (clase.equals(tipo2.getClass()))
    {escalar = new Escalar(new Float((float)(this._SiEscalarInteger.intValue())));}
   if (clase.equals(tipo3.getClass()))
    {escalar = new Escalar(new Long((long)(this._SiEscalarInteger.intValue())));}
   if (clase.equals(tipo4.getClass()))
    {escalar = new Escalar(this._SiEscalarInteger);}
  }
  return escalar;
 }

 // representación cadena del Escalar
 public String toString()
 {
  String result = "";
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   {result = this._SiEscalarDouble.toString();}
  else if (this._clase.equals(tipo2.getClass()))
   {result = this._SiEscalarFloat.toString();}
  else if (this._clase.equals(tipo3.getClass()))
   {result = this._SiEscalarLong.toString();}
  else if (this._clase.equals(tipo4.getClass()))
   {result = this._SiEscalarInteger.toString();}
  return result;
 }

 // suma dos escalares (realiza la operación con el de mayor precisión y deja el
 // resultado en el tipo de la instancia que llama al método
 public Escalar suma(Escalar escalar2){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())){
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
escalar2._SiEscalarDouble.doubleValue()));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
escalar2._SiEscalarFloat.doubleValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
escalar2._SiEscalarLong.doubleValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
escalar2._SiEscalarInteger.doubleValue()));
  } else if (this._clase.equals(tipo2.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() +
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
```

```
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() +
escalar2._SiEscalarFloat.floatValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() +
escalar2._SiEscalarLong.floatValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() +
escalar2._SiEscalarInteger.floatValue()));
  } else if (this._clase.equals(tipo3.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() +
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() +
escalar2._SiEscalarFloat.floatValue())));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Long(this._SiEscalarLong.longValue() +
escalar2._SiEscalarLong.longValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Long(this._SiEscalarLong.longValue() +
escalar2._SiEscalarInteger.longValue()));
  } else if (this._clase.equals(tipo4.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() +
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() +
escalar2._SiEscalarFloat.floatValue())));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() +
escalar2._SiEscalarLong.longValue())));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() +
escalar2._SiEscalarInteger.intValue()));
  }
  return escalar;
 }
 public Escalar suma(Double db1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
db1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() +
db1.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() +
db1.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() +
db1.doubleValue())));
  return escalar;
 }
 public Escalar suma(Float fl1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
fl1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() + fl1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() +
fl1.floatValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() +
fl1.floatValue())));
  return escalar;
```

```
 }
 public Escalar suma(Long lg1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
lg1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() + lg1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() + lg1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() +
lg1.longValue())));
  return escalar;
 }
 public Escalar suma(Integer in1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
in1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() + in1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() + in1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() + in1.intValue()));
  return escalar;
 }
 public Escalar suma(double db2){
  Escalar escalar = new Escalar(this._clase);
  Double db1 = new Double(db2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
db1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() +
db1.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() +
db1.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() +
db1.doubleValue())));
  return escalar;
 }
 public Escalar suma(float fl2){
  Escalar escalar = new Escalar(this._clase);
  Float fl1 = new Float(fl2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
fl1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() + fl1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() +
fl1.floatValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() +
fl1.floatValue())));
  return escalar;
 }
 public Escalar suma(long lg2){
  Escalar escalar = new Escalar(this._clase);
```

```
  Long lg1 = new Long(lg2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
lg1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() + lg1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() + lg1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() +
lg1.longValue())));
  return escalar;
 }
 public Escalar suma(int in2){
  Escalar escalar = new Escalar(this._clase);
  Integer in1 = new Integer(in2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() +
in1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() + in1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() + in1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() + in1.intValue()));
  return escalar;
 }
 // este método regresa los tipos de Escalares de retorno, según estén definidos en el
parámetro
 // escalarComplejo (Escalar real y Escalar imaginario)
 public EscalarComplejo suma(EscalarComplejo escalarComplejo){
  return escalarComplejo.suma(this);
 }

 // resta dos escalares (realiza la operación con el de mayor presición y deja el
 // resultado en el tipo de la instancia que llama al método. Escalar instancia que
 // llama al método menos escalar2
 public Escalar resta(Escalar escalar2){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())){
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
escalar2._SiEscalarDouble.doubleValue()));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
escalar2._SiEscalarFloat.doubleValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
escalar2._SiEscalarLong.doubleValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
escalar2._SiEscalarInteger.doubleValue()));
  } else if (this._clase.equals(tipo2.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() -
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() -
escalar2._SiEscalarFloat.floatValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() -
escalar2._SiEscalarLong.floatValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() -
escalar2._SiEscalarInteger.floatValue()));
```

```
  } else if (this._clase.equals(tipo3.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() -
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() -
escalar2._SiEscalarFloat.floatValue())));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Long(this._SiEscalarLong.longValue() -
escalar2._SiEscalarLong.longValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Long(this._SiEscalarLong.longValue() -
escalar2._SiEscalarInteger.longValue()));
  } else if (this._clase.equals(tipo4.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() -
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() -
escalar2._SiEscalarFloat.floatValue())));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() -
escalar2._SiEscalarLong.longValue())));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() -
escalar2._SiEscalarInteger.intValue()));
  }
  return escalar;
 }
 public Escalar resta(Double db1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
db1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() -
db1.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() -
db1.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() -
db1.doubleValue())));
  return escalar;
 }
 public Escalar resta(Float fl1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
fl1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() - fl1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() -
fl1.floatValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() -
fl1.floatValue())));
  return escalar;
 }
 public Escalar resta(Long lg1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
lg1.doubleValue()));
```

```
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() - lg1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() - lg1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() -
lg1.longValue())));
  return escalar;
 }
 public Escalar resta(Integer in1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
in1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() - in1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() - in1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() - in1.intValue()));
  return escalar;
 }
 public Escalar resta(double db2){
  Escalar escalar = new Escalar(this._clase);
  Double db1 = new Double(db2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
db1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() -
db1.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() -
db1.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() -
db1.doubleValue())));
  return escalar;
 }
 public Escalar resta(float fl2){
  Escalar escalar = new Escalar(this._clase);
  Float fl1 = new Float(fl2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
fl1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() - fl1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() -
fl1.floatValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() -
fl1.floatValue())));
  return escalar;
 }
 public Escalar resta(long lg2){
  Escalar escalar = new Escalar(this._clase);
  Long lg1 = new Long(lg2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
lg1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() - lg1.floatValue()));
```

```
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() - lg1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() -
lg1.longValue())));
  return escalar;
 }
 public Escalar resta(int in2){
  Escalar escalar = new Escalar(this._clase);
  Integer in1 = new Integer(in2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() -
in1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() - in1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() - in1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() - in1.intValue()));
  return escalar;
 }
 // este método regresa los tipos de Escalares de retorno, según estén definidos en el
parámetro
 // escalarComplejo (Escalar real y Escalar imaginario)
 public EscalarComplejo resta(EscalarComplejo escalarComplejo){
  escalarComplejo = escalarComplejo.multiplica(new Escalar(-1));
  return escalarComplejo.suma(this);
 }

 // multiplica dos escalares (realiza la operación con el de mayor presición y deja el
 // resultado en el tipo de la instancia que llama al método
 public Escalar multiplica(Escalar escalar2){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())){
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
escalar2._SiEscalarDouble.doubleValue()));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
escalar2._SiEscalarFloat.doubleValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
escalar2._SiEscalarLong.doubleValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
escalar2._SiEscalarInteger.doubleValue()));
  } else if (this._clase.equals(tipo2.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() *
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() *
escalar2._SiEscalarFloat.floatValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() *
escalar2._SiEscalarLong.floatValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() *
escalar2._SiEscalarInteger.floatValue()));
  } else if (this._clase.equals(tipo3.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() *
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() *
escalar2._SiEscalarFloat.floatValue())));
   else if (escalar2._clase.equals(tipo3.getClass()))
```

```
    escalar = new Escalar(new Long(this._SiEscalarLong.longValue() *
escalar2._SiEscalarLong.longValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Long(this._SiEscalarLong.longValue() *
escalar2._SiEscalarInteger.longValue()));
  } else if (this._clase.equals(tipo4.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() *
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() *
escalar2._SiEscalarFloat.floatValue())));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() *
escalar2._SiEscalarLong.longValue())));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() *
escalar2._SiEscalarInteger.intValue()));
  }
  return escalar;
 }
 public Escalar multiplica(Double db1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
db1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() *
db1.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() *
db1.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() *
db1.doubleValue())));
  return escalar;
 }
 public Escalar multiplica(Float fl1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
fl1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() * fl1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() *
fl1.floatValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() *
fl1.floatValue())));
  return escalar;
 }
 public Escalar multiplica(Long lg1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
lg1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() * lg1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() * lg1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() *
lg1.longValue())));
  return escalar;
```

```
 }
 public Escalar multiplica(Integer in1){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
in1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() * in1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() * in1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() * in1.intValue()));
  return escalar;
 }
 public Escalar multiplica(double db2){
  Escalar escalar = new Escalar(this._clase);
  Double db1 = new Double(db2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
db1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() *
db1.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() *
db1.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() *
db1.doubleValue())));
  return escalar;
 }
 public Escalar multiplica(float fl2){
  Escalar escalar = new Escalar(this._clase);
  Float fl1 = new Float(fl2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
fl1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() * fl1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() *
fl1.floatValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() *
fl1.floatValue())));
  return escalar;
 }
 public Escalar multiplica(long lg2){
  Escalar escalar = new Escalar(this._clase);
  Long lg1 = new Long(lg2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
lg1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() * lg1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() * lg1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() *
lg1.longValue())));
  return escalar;
 }
 public Escalar multiplica(int in2){
```

```
  Escalar escalar = new Escalar(this._clase);
  Integer in1 = new Integer(in2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() *
in1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() * in1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() * in1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() * in1.intValue()));
  return escalar;
 }
 // este método regresa los tipos de Escalares de retorno, según estén definidos en el
parámetro
 // escalarComplejo (Escalar real y Escalar imaginario)
 public EscalarComplejo multiplica(EscalarComplejo escalarComplejo){
  return escalarComplejo.multiplica(this);
 }

 // divide dos escalares (realiza la operación con el de mayor presición y deja el
 // resultado en el tipo de la instancia que llama al método, si se intenta dividir entre
cero
 // regresa el numerador como resultado de la división
 public Escalar divide(Escalar escalar2){
  if (escalar2.igual(new Escalar(0.)) == true) return this;
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())){
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
escalar2._SiEscalarDouble.doubleValue()));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
escalar2._SiEscalarFloat.doubleValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
escalar2._SiEscalarLong.doubleValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
escalar2._SiEscalarInteger.doubleValue()));
  } else if (this._clase.equals(tipo2.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() /
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() /
escalar2._SiEscalarFloat.floatValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() /
escalar2._SiEscalarLong.floatValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() /
escalar2._SiEscalarInteger.floatValue()));
  } else if (this._clase.equals(tipo3.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() /
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() /
escalar2._SiEscalarFloat.floatValue())));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Long(this._SiEscalarLong.longValue() /
escalar2._SiEscalarLong.longValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Long(this._SiEscalarLong.longValue() /
escalar2._SiEscalarInteger.longValue()));
  } else if (this._clase.equals(tipo4.getClass())) {
```

```
   if (escalar2._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() /
escalar2._SiEscalarDouble.doubleValue())));
   else if (escalar2._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() /
escalar2._SiEscalarFloat.floatValue())));
   else if (escalar2._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() /
escalar2._SiEscalarLong.longValue())));
   else if (escalar2._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() /
escalar2._SiEscalarInteger.intValue()));
  }
  return escalar;
 }
 public Escalar divide(Double db1){
  if (db1.equals(new Double(0.)) == true) return this;
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
db1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() /
db1.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() /
db1.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() /
db1.doubleValue())));
  return escalar;
 }
 public Escalar divide(Float fl1){
  if (fl1.equals(new Float(0f)) == true) return this;
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
fl1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() / fl1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() /
fl1.floatValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() /
fl1.floatValue())));
  return escalar;
 }
 public Escalar divide(Long lg1){
  if (lg1.equals(new Long(0L)) == true) return this;
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
lg1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() / lg1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() / lg1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() /
lg1.longValue())));
  return escalar;
 }
 public Escalar divide(Integer in1){
  if (in1.equals(new Integer(0)) == true) return this;
```

```
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
in1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() / in1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() / in1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() / in1.intValue()));
  return escalar;
 }
 public Escalar divide(double db2){
  if (db2 == 0.) return this;
  Escalar escalar = new Escalar(this._clase);
  Double db1 = new Double(db2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
db1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() /
db1.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() /
db1.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() /
db1.doubleValue())));
  return escalar;
 }
 public Escalar divide(float fl2){
  if (fl2 == 0f) return this;
  Escalar escalar = new Escalar(this._clase);
  Float fl1 = new Float(fl2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
fl1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() / fl1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)(this._SiEscalarLong.floatValue() /
fl1.floatValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.floatValue() /
fl1.floatValue())));
  return escalar;
 }
 public Escalar divide(long lg2){
  if (lg2 == 0L) return this;
  Escalar escalar = new Escalar(this._clase);
  Long lg1 = new Long(lg2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
lg1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() / lg1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() / lg1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.longValue() /
lg1.longValue())));
  return escalar;
 }
```

```
 public Escalar divide(int in2){
  if (in2 == 0) return this;
  Escalar escalar = new Escalar(this._clase);
  Integer in1 = new Integer(in2);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() /
in1.doubleValue()));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(this._SiEscalarFloat.floatValue() / in1.floatValue()));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue() / in1.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue() / in1.intValue()));
  return escalar;
 }
 // este método regresa los tipos de Escalares de retorno, según estén definidos en el
parámetro
 // escalarComplejo (Escalar real y Escalar imaginario)
 public EscalarComplejo divide(EscalarComplejo escalarComplejo){
  EscalarComplejo tempComplejo =
   new EscalarComplejo(this.convierteTipoEscalar(escalarComplejo._real._clase), new
Escalar(escalarComplejo._imaginario._clase));
  return tempComplejo.divide(escalarComplejo);
 }

 // compara dos escalares y regresa true si tienen el mismo valor, si no regresa false
 public boolean igual(Escalar escalar2){
  boolean res = false;
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())){
   if (escalar2._clase.equals(tipo1.getClass()))
    res = this._SiEscalarDouble.equals(escalar2._SiEscalarDouble);
   else if (escalar2._clase.equals(tipo2.getClass()))
    res = this._SiEscalarDouble.equals(new Double(escalar2._SiEscalarFloat.doubleValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    res = this._SiEscalarDouble.equals(new Double(escalar2._SiEscalarLong.doubleValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    res = this._SiEscalarDouble.equals(new
Double(escalar2._SiEscalarInteger.doubleValue()));
  } else if (this._clase.equals(tipo2.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    res = this._SiEscalarFloat.equals(new Float(escalar2._SiEscalarDouble.floatValue()));
   else if (escalar2._clase.equals(tipo2.getClass()))
    res = this._SiEscalarFloat.equals(escalar2._SiEscalarFloat);
   else if (escalar2._clase.equals(tipo3.getClass()))
    res = this._SiEscalarFloat.equals(new Float(escalar2._SiEscalarLong.floatValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    res = this._SiEscalarFloat.equals(new Float(escalar2._SiEscalarInteger.floatValue()));
  } else if (this._clase.equals(tipo3.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    res = this._SiEscalarLong.equals(new Long(escalar2._SiEscalarDouble.longValue()));
   else if (escalar2._clase.equals(tipo2.getClass()))
    res = this._SiEscalarLong.equals(new Long(escalar2._SiEscalarFloat.longValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    res = this._SiEscalarLong.equals(escalar2._SiEscalarLong);
   else if (escalar2._clase.equals(tipo4.getClass()))
    res = this._SiEscalarLong.equals(new Long(escalar2._SiEscalarInteger.longValue()));
  } else if (this._clase.equals(tipo4.getClass())) {
   if (escalar2._clase.equals(tipo1.getClass()))
    res = this._SiEscalarInteger.equals(new Integer(escalar2._SiEscalarDouble.intValue()));
   else if (escalar2._clase.equals(tipo2.getClass()))
    res = this._SiEscalarInteger.equals(new Integer(escalar2._SiEscalarFloat.intValue()));
   else if (escalar2._clase.equals(tipo3.getClass()))
    res = this._SiEscalarInteger.equals(new Integer(escalar2._SiEscalarLong.intValue()));
   else if (escalar2._clase.equals(tipo4.getClass()))
    res = this._SiEscalarInteger.equals(escalar2._SiEscalarInteger);
  }
  return res;
```

```
 }

 // eleva al cuadrado el escalar
 public Escalar elevaAlcuadrado(){
  Escalar escalar = new Escalar(this._clase);
  escalar = this.multiplica(this);
  return escalar;
 }

 // calcula la raiz cuadrada del Escalar (instancia actual) que llama al método, deja el
tipo
 // de dato como el definido por la instancia actual. El dominio de la función es de
[0,inf],
 // y el rango va de [0,inf]
 public Escalar raizCuadrada(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.sqrt(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.sqrt(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.sqrt(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)Math.sqrt(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // obten el residuo de la división de dos Escalares
 public Escalar residuoRedondeado(Escalar denominador) {
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())) {
   if (denominador._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new
Double(Math.IEEEremainder(this._SiEscalarDouble.doubleValue(),denominador._SiEscalarDouble.d
oubleValue())));
   else if (denominador._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new
Double(Math.IEEEremainder(this._SiEscalarDouble.doubleValue(),denominador._SiEscalarFloat.do
ubleValue())));
   else if (denominador._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new
Double(Math.IEEEremainder(this._SiEscalarDouble.doubleValue(),denominador._SiEscalarLong.dou
bleValue())));
   else if (denominador._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new
Double(Math.IEEEremainder(this._SiEscalarDouble.doubleValue(),denominador._SiEscalarInteger.
doubleValue())));
  } else if (this._clase.equals(tipo2.getClass())) {
   if (denominador._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new
Float((float)Math.IEEEremainder(this._SiEscalarFloat.doubleValue(),denominador._SiEscalarDou
ble.doubleValue())));
   else if (denominador._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new
Float((float)Math.IEEEremainder(this._SiEscalarFloat.doubleValue(),denominador._SiEscalarFlo
at.doubleValue())));
   else if (denominador._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new
Float((float)Math.IEEEremainder(this._SiEscalarFloat.doubleValue(),denominador._SiEscalarLon
g.doubleValue())));
   else if (denominador._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new
Float((float)Math.IEEEremainder(this._SiEscalarFloat.doubleValue(),denominador._SiEscalarInt
eger.doubleValue())));
  } else if (this._clase.equals(tipo3.getClass())) {
   if (denominador._clase.equals(tipo1.getClass()))
```

```
    escalar = new Escalar(new
Long((long)Math.IEEEremainder(this._SiEscalarLong.doubleValue(),denominador._SiEscalarDouble
.doubleValue())));
   else if (denominador._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new
Long((long)Math.IEEEremainder(this._SiEscalarLong.doubleValue(),denominador._SiEscalarFloat.
doubleValue())));
   else if (denominador._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new
Long((long)Math.IEEEremainder(this._SiEscalarLong.doubleValue(),denominador._SiEscalarLong.d
oubleValue())));
   else if (denominador._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new
Long((long)Math.IEEEremainder(this._SiEscalarLong.doubleValue(),denominador._SiEscalarIntege
r.doubleValue())));
  } else if (this._clase.equals(tipo4.getClass())) {
   if (denominador._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new
Integer((int)Math.IEEEremainder(this._SiEscalarInteger.doubleValue(),denominador._SiEscalarD
ouble.doubleValue())));
   else if (denominador._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new
Integer((int)Math.IEEEremainder(this._SiEscalarInteger.doubleValue(),denominador._SiEscalarF
loat.doubleValue())));
   else if (denominador._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new
Integer((int)Math.IEEEremainder(this._SiEscalarInteger.doubleValue(),denominador._SiEscalarL
ong.doubleValue())));
   else if (denominador._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new
Integer((int)Math.IEEEremainder(this._SiEscalarInteger.doubleValue(),denominador._SiEscalarI
nteger.doubleValue())));
  }
  return escalar;
 }

 // obten el residuo truncado de la división de dos Escalares
 public Escalar residuoTruncado(Escalar denominador) {
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())) {
   if (denominador._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() %
denominador._SiEscalarDouble.doubleValue()));
   else if (denominador._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() %
denominador._SiEscalarFloat.doubleValue()));
   else if (denominador._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() %
denominador._SiEscalarLong.doubleValue()));
   else if (denominador._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Double(this._SiEscalarDouble.doubleValue() %
denominador._SiEscalarInteger.doubleValue()));
  } else if (this._clase.equals(tipo2.getClass())) {
   if (denominador._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() %
denominador._SiEscalarDouble.doubleValue())));
   else if (denominador._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() %
denominador._SiEscalarFloat.doubleValue())));
   else if (denominador._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() %
denominador._SiEscalarLong.doubleValue())));
   else if (denominador._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Float((float)(this._SiEscalarFloat.doubleValue() %
denominador._SiEscalarInteger.doubleValue())));
  } else if (this._clase.equals(tipo3.getClass())) {
   if (denominador._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() %
denominador._SiEscalarDouble.doubleValue())));
```

```
   else if (denominador._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() %
denominador._SiEscalarFloat.doubleValue())));
   else if (denominador._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() %
denominador._SiEscalarLong.doubleValue())));
   else if (denominador._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Long((long)(this._SiEscalarLong.doubleValue() %
denominador._SiEscalarInteger.doubleValue())));
  } else if (this._clase.equals(tipo4.getClass())) {
   if (denominador._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() %
denominador._SiEscalarDouble.doubleValue())));
   else if (denominador._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() %
denominador._SiEscalarFloat.doubleValue())));
   else if (denominador._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() %
denominador._SiEscalarLong.doubleValue())));
   else if (denominador._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new Integer((int)(this._SiEscalarInteger.doubleValue() %
denominador._SiEscalarInteger.doubleValue())));
  }
  return escalar;
 }

 // saca el Valor absoluto al escalar
 public Escalar valorAbsoluto(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.abs(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float(Math.abs(this._SiEscalarFloat.floatValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(Math.abs(this._SiEscalarLong.longValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(Math.abs(this._SiEscalarInteger.intValue())));
  return escalar;
 }

 // saca el arcoCoseno al Escalar, convierte el resultado al tipo de la instancia (Escalar)
 // que llama al método. Precaución: los tipos Escalares (Long e Integer) regresarán cero
 // siempre, excepto en 0. El dominio de la función es [-1,1], fuera del dominio regresa
 // cero, el rango de la función es de [0,PI] (es un angulo), aunque es multi-valuada
 // ya que produce: un angulo en el rango de la función que es el valor principal aunque
 // existen infinitos valores secundarios: angulo +  n (2 PI), donde n = 0, +/- 1, +/- 2,
...
 // hasta infinito
 public Escalar arcoCoseno(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.acos(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.acos(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.acos(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)Math.acos(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // saca el arcoSeno al Escalar, convierte el resultado al tipo de la instancia (Escalar)
 // que llama al método. Precaución: los tipos Escalares (Long e Integer) regresarán cero
 // siempre, excepto en 1. El dominio de la función es [-1,1], fuera del dominio regresa
 // cero, el rango de la función es de [-PI/2,PI/2] (es un angulo), aunque es multi-valuada
 // ya que produce: un angulo en el rango de la función que es el valor principal aunque
```

```
 // existen infinitos valores secundarios: angulo +  n (2 PI), donde n = 0, +/- 1, +/- 2,
...
 // hasta infinito
 public Escalar arcoSeno(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.asin(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.asin(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.asin(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)Math.asin(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // saca el arcoTangente al Escalar, convierte el resultado al tipo de la instancia
(Escalar)
 // que llama al método. El dominio de la función es [-inf,inf], el rango de la función es
 // de [-PI/2,PI/2] (es un angulo), aunque es multi-valuada ya que produce: un angulo en el
 // rango de la función que es la rama principal aunque existen infinitas ramas secundarias:
 // angulo +  n (2 PI), donde n = 0, +/- 1, +/- 2, ... hasta infinito
 public Escalar arcoTangente(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.atan(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.atan(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.atan(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)Math.atan(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // saca el arcoTangente al Escalar, convierte el resultado al tipo de la instancia
(Escalar)
 // que llama al método. El dominio de la función es [-inf,inf], el rango de la función es
 // de [-PI,PI] (es un angulo), aunque es multi-valuada ya que produce: un angulo en el
 // rango de la función que es la rama principal aunque existen infinitas ramas secundarias:
 // angulo +  n (2 PI), donde n = 0, +/- 1, +/- 2, ... hasta infinito. El parámetro x
(parámetro)
 // es el numerador y la instancia actual (escalar que llama a este método) es el
denominador,
 // i.e.; arco tangente de (x / instancia actual)
 public Escalar arcoTangente2(Escalar x) {
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())) {
   if (x._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new
Double(Math.atan2(x._SiEscalarDouble.doubleValue(),this._SiEscalarDouble.doubleValue())));
   else if (x._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new
Double(Math.atan2(x._SiEscalarFloat.doubleValue(),this._SiEscalarDouble.doubleValue())));
   else if (x._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new
Double(Math.atan2(x._SiEscalarLong.doubleValue(),this._SiEscalarDouble.doubleValue())));
   else if (x._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new
Double(Math.atan2(x._SiEscalarInteger.doubleValue(),this._SiEscalarDouble.doubleValue())));
  } else if (this._clase.equals(tipo2.getClass())) {
   if (x._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new
Float((float)Math.atan2(x._SiEscalarDouble.doubleValue(),this._SiEscalarFloat.doubleValue())
));
```

```
   else if (x._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new
Float((float)Math.atan2(x._SiEscalarFloat.doubleValue(),this._SiEscalarFloat.doubleValue()))
);
   else if (x._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new
Float((float)Math.atan2(x._SiEscalarLong.doubleValue(),this._SiEscalarFloat.doubleValue())))
;
   else if (x._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new
Float((float)Math.atan2(x._SiEscalarInteger.doubleValue(),this._SiEscalarFloat.doubleValue()
)));
  } else if (this._clase.equals(tipo3.getClass())) {
   if (x._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new
Long((long)Math.atan2(x._SiEscalarDouble.doubleValue(),this._SiEscalarLong.doubleValue())));
   else if (x._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new
Long((long)Math.atan2(x._SiEscalarFloat.doubleValue(),this._SiEscalarLong.doubleValue())));
   else if (x._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new
Long((long)Math.atan2(x._SiEscalarLong.doubleValue(),this._SiEscalarLong.doubleValue())));
   else if (x._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new
Long((long)Math.atan2(x._SiEscalarInteger.doubleValue(),this._SiEscalarLong.doubleValue())))
;
  } else if (this._clase.equals(tipo4.getClass())) {
   if (x._clase.equals(tipo1.getClass()))
    escalar = new Escalar(new
Integer((int)Math.atan2(x._SiEscalarDouble.doubleValue(),this._SiEscalarInteger.doubleValue(
))));
   else if (x._clase.equals(tipo2.getClass()))
    escalar = new Escalar(new
Integer((int)Math.atan2(x._SiEscalarFloat.doubleValue(),this._SiEscalarInteger.doubleValue()
)));
   else if (x._clase.equals(tipo3.getClass()))
    escalar = new Escalar(new
Integer((int)Math.atan2(x._SiEscalarLong.doubleValue(),this._SiEscalarInteger.doubleValue())
));
   else if (x._clase.equals(tipo4.getClass()))
    escalar = new Escalar(new
Integer((int)Math.atan2(x._SiEscalarInteger.doubleValue(),this._SiEscalarInteger.doubleValue
())));
  }
  return escalar;
 }

 // calcula el entero más pequeño mayor que o igual que el Escalar de la instancia actual,
deja
 // el tipo de dato del Escalar de la instancia que llama al método
 public Escalar enteroMenor(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.ceil(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.ceil(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.ceil(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)Math.ceil(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // calcula el coseno del Escalar (instancia actual) que llama al método (rad), deja el tipo
de
 // dato como el definido por la instancia actual. El dominio de la función es de [-
inf,inf], y
 // el rango va de [-1,1]
 public Escalar coseno(){
```

```
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.cos(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.cos(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.cos(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)Math.cos(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // calcula el exponencial elevado a la potencia dada por el Escalar que llama al método,
 // deja el tipo de dato del Escalar de la instancia que llama al método
 public Escalar exponencial(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.exp(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.exp(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.exp(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)Math.exp(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // calcula el entero más grande menor que o igual que el Escalar de la instancia actual,
deja
 // el tipo de dato del Escalar de la instancia que llama al método
 public Escalar enteroMayor(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.floor(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.floor(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.floor(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new
Integer((int)Math.floor(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // calcula el logaritmo natural del Escalar que llama al método, deja el tipo de dato del
 // Escalar de la instancia que llama al método
 public Escalar logaritmoNatural(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.log(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.log(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.log(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)Math.log(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // calcula el logaritmo base parámetro, del Escalar que llama al método, deja el tipo de
dato del
 // Escalar de la instancia que llama al método
 public Escalar logaritmoBase(Escalar base){
```

```
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())) {
   if (base._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Double(Math.log(this._SiEscalarDouble.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarDouble.doubleValue()))));
   } else if (base._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Double(Math.log(this._SiEscalarDouble.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarFloat.doubleValue()))));
   } else if (base._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Double(Math.log(this._SiEscalarDouble.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarLong.doubleValue()))));
   } else if (base._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Double(Math.log(this._SiEscalarDouble.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarInteger.doubleValue()))));
   }
  } else if (this._clase.equals(tipo2.getClass())) {
   if (base._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Float((float)Math.log(this._SiEscalarFloat.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarDouble.doubleValue()))));
   } else if (base._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Float((float)Math.log(this._SiEscalarFloat.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarFloat.doubleValue()))));
   } else if (base._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Float((float)Math.log(this._SiEscalarFloat.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarLong.doubleValue()))));
   } else if (base._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Float((float)Math.log(this._SiEscalarFloat.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarInteger.doubleValue()))));
   }
  } else if (this._clase.equals(tipo3.getClass())) {
   if (base._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Long((long)Math.log(this._SiEscalarLong.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarDouble.doubleValue()))));
   } else if (base._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Long((long)Math.log(this._SiEscalarLong.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarFloat.doubleValue()))));
   } else if (base._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Long((long)Math.log(this._SiEscalarLong.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarLong.doubleValue()))));
   } else if (base._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Long((long)Math.log(this._SiEscalarLong.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarInteger.doubleValue()))));
   }
  } else if (this._clase.equals(tipo4.getClass())) {
   if (base._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Integer((int)Math.log(this._SiEscalarInteger.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarDouble.doubleValue()))));
   } else if (base._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Integer((int)Math.log(this._SiEscalarInteger.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarFloat.doubleValue()))));
   } else if (base._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Integer((int)Math.log(this._SiEscalarInteger.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarLong.doubleValue()))));
   } else if (base._clase.equals(tipo4.getClass())) {
```

```
    escalar = new Escalar(new Integer((int)Math.log(this._SiEscalarInteger.doubleValue())));
    escalar = escalar.divide(new Escalar(new
Double(Math.log(base._SiEscalarInteger.doubleValue()))));
   }
  }
  return escalar;
 }

 // regresa el valor más grande de los valores de dos Escalares, uno es la instancia que
llama
 // al método y el otro es el Escalar pasado como parámetro (otro), deja el tipo de dato
 // en el tipo primitivo del Escalar de la instancia que llama al método
 public Escalar mayor(Escalar otro){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())) {
   if (otro._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Double(Math.max(this._SiEscalarDouble.doubleValue(),
                                              otro._SiEscalarDouble.doubleValue())));
   } else if (otro._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Double(Math.max(this._SiEscalarDouble.doubleValue(),
                                              otro._SiEscalarFloat.doubleValue())));
   } else if (otro._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Double(Math.max(this._SiEscalarDouble.doubleValue(),
                                              otro._SiEscalarLong.doubleValue())));
   } else if (otro._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Double(Math.max(this._SiEscalarDouble.doubleValue(),
                                              otro._SiEscalarInteger.doubleValue())));
   }
  } else if (this._clase.equals(tipo2.getClass())) {
   if (otro._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Float((float)Math.max(this._SiEscalarFloat.floatValue(),
                                                    otro._SiEscalarDouble.floatValue())));
   } else if (otro._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Float((float)Math.max(this._SiEscalarFloat.floatValue(),
                                                    otro._SiEscalarFloat.floatValue())));
   } else if (otro._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Float((float)Math.max(this._SiEscalarFloat.floatValue(),
                                                    otro._SiEscalarLong.floatValue())));
   } else if (otro._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Float((float)Math.max(this._SiEscalarFloat.floatValue(),
                                                    otro._SiEscalarInteger.floatValue())));
   }
  } else if (this._clase.equals(tipo3.getClass())) {
   if (otro._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Long((long)Math.max(this._SiEscalarLong.longValue(),
                                                  otro._SiEscalarDouble.longValue())));
   } else if (otro._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Long((long)Math.max(this._SiEscalarLong.longValue(),
                                                  otro._SiEscalarFloat.longValue())));
   } else if (otro._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Long((long)Math.max(this._SiEscalarLong.longValue(),
                                                  otro._SiEscalarLong.longValue())));
   } else if (otro._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Long((long)Math.max(this._SiEscalarLong.longValue(),
                                                  otro._SiEscalarInteger.longValue())));
   }
  } else if (this._clase.equals(tipo4.getClass())) {
   if (otro._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Integer((int)Math.max(this._SiEscalarInteger.intValue(),
                                                    otro._SiEscalarDouble.intValue())));
   } else if (otro._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Integer((int)Math.max(this._SiEscalarInteger.intValue(),
                                                    otro._SiEscalarFloat.intValue())));
   } else if (otro._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Integer((int)Math.max(this._SiEscalarInteger.intValue(),
                                                    otro._SiEscalarLong.intValue())));
   } else if (otro._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Integer((int)Math.max(this._SiEscalarInteger.intValue(),
                                                    otro._SiEscalarInteger.intValue())));
```

```
   }
  }
  return escalar;
 }

 // regresa el valor más pequeño de los valores de dos Escalares, uno es la instancia que
llama
 // al método y el otro es el Escalar pasado como parámetro (otro), deja el tipo de dato
 // en el tipo primitivo del Escalar de la instancia que llama al método
 public Escalar menor(Escalar otro){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())) {
   if (otro._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Double(Math.min(this._SiEscalarDouble.doubleValue(),
                                              otro._SiEscalarDouble.doubleValue())));
   } else if (otro._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Double(Math.min(this._SiEscalarDouble.doubleValue(),
                                              otro._SiEscalarFloat.doubleValue())));
   } else if (otro._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Double(Math.min(this._SiEscalarDouble.doubleValue(),
                                              otro._SiEscalarLong.doubleValue())));
   } else if (otro._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Double(Math.min(this._SiEscalarDouble.doubleValue(),
                                              otro._SiEscalarInteger.doubleValue())));
   }
  } else if (this._clase.equals(tipo2.getClass())) {
   if (otro._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Float((float)Math.min(this._SiEscalarFloat.floatValue(),
                                                    otro._SiEscalarDouble.floatValue())));
   } else if (otro._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Float((float)Math.min(this._SiEscalarFloat.floatValue(),
                                                    otro._SiEscalarFloat.floatValue())));
   } else if (otro._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Float((float)Math.min(this._SiEscalarFloat.floatValue(),
                                                    otro._SiEscalarLong.floatValue())));
   } else if (otro._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Float((float)Math.min(this._SiEscalarFloat.floatValue(),
                                                    otro._SiEscalarInteger.floatValue())));
   }
  } else if (this._clase.equals(tipo3.getClass())) {
   if (otro._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Long((long)Math.min(this._SiEscalarLong.longValue(),
                                                  otro._SiEscalarDouble.longValue())));
   } else if (otro._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Long((long)Math.min(this._SiEscalarLong.longValue(),
                                                  otro._SiEscalarFloat.longValue())));
   } else if (otro._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Long((long)Math.min(this._SiEscalarLong.longValue(),
                                                  otro._SiEscalarLong.longValue())));
   } else if (otro._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Long((long)Math.min(this._SiEscalarLong.longValue(),
                                                  otro._SiEscalarInteger.longValue())));
   }
  } else if (this._clase.equals(tipo4.getClass())) {
   if (otro._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Integer((int)Math.min(this._SiEscalarInteger.intValue(),
                                                    otro._SiEscalarDouble.intValue())));
   } else if (otro._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Integer((int)Math.min(this._SiEscalarInteger.intValue(),
                                                    otro._SiEscalarFloat.intValue())));
   } else if (otro._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Integer((int)Math.min(this._SiEscalarInteger.intValue(),
                                                    otro._SiEscalarLong.intValue())));
   } else if (otro._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Integer((int)Math.min(this._SiEscalarInteger.intValue(),
                                                    otro._SiEscalarInteger.intValue())));
   }
  }
  return escalar;
```

```
 }

 // calcula la base, dada por la instancia que llama al método, elevada a una potencia dada
 // por el escalar pasado como parámetro (potencia), deja el tipo de dato del Escalar de la
 // instancia que llama al método
 public Escalar potencia(Escalar potencia){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass())) {
   if (potencia._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Double(Math.pow(this._SiEscalarDouble.doubleValue(),
                                              potencia._SiEscalarDouble.doubleValue())));
   } else if (potencia._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Double(Math.pow(this._SiEscalarDouble.doubleValue(),
                                              potencia._SiEscalarFloat.doubleValue())));
   } else if (potencia._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Double(Math.pow(this._SiEscalarDouble.doubleValue(),
                                              potencia._SiEscalarLong.doubleValue())));
   } else if (potencia._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Double(Math.pow(this._SiEscalarDouble.doubleValue(),
                                              potencia._SiEscalarInteger.doubleValue())));
   }
  } else if (this._clase.equals(tipo2.getClass())) {
   if (potencia._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Float((float)Math.pow(this._SiEscalarFloat.doubleValue(),

potencia._SiEscalarDouble.doubleValue())));
   } else if (potencia._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Float((float)Math.pow(this._SiEscalarFloat.doubleValue(),

potencia._SiEscalarFloat.doubleValue())));
   } else if (potencia._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Float((float)Math.pow(this._SiEscalarFloat.doubleValue(),

potencia._SiEscalarLong.doubleValue())));
   } else if (potencia._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Float((float)Math.pow(this._SiEscalarFloat.doubleValue(),

potencia._SiEscalarInteger.doubleValue())));
   }
  } else if (this._clase.equals(tipo3.getClass())) {
   if (potencia._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Long((long)Math.pow(this._SiEscalarLong.doubleValue(),

potencia._SiEscalarDouble.doubleValue())));
   } else if (potencia._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Long((long)Math.pow(this._SiEscalarLong.doubleValue(),
                                                  potencia._SiEscalarFloat.doubleValue())));
   } else if (potencia._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Long((long)Math.pow(this._SiEscalarLong.doubleValue(),
                                                  potencia._SiEscalarLong.doubleValue())));
   } else if (potencia._clase.equals(tipo4.getClass())) {
    escalar = new Escalar(new Long((long)Math.pow(this._SiEscalarLong.doubleValue(),

potencia._SiEscalarInteger.doubleValue())));
   }
  } else if (this._clase.equals(tipo4.getClass())) {
   if (potencia._clase.equals(tipo1.getClass())) {
    escalar = new Escalar(new Integer((int)Math.pow(this._SiEscalarInteger.doubleValue(),

potencia._SiEscalarDouble.doubleValue())));
   } else if (potencia._clase.equals(tipo2.getClass())) {
    escalar = new Escalar(new Integer((int)Math.pow(this._SiEscalarInteger.doubleValue(),

potencia._SiEscalarFloat.doubleValue())));
   } else if (potencia._clase.equals(tipo3.getClass())) {
    escalar = new Escalar(new Integer((int)Math.pow(this._SiEscalarInteger.doubleValue(),

potencia._SiEscalarLong.doubleValue())));
   } else if (potencia._clase.equals(tipo4.getClass())) {
```

```
    escalar = new Escalar(new Integer((int)Math.pow(this._SiEscalarInteger.doubleValue(),
potencia._SiEscalarInteger.doubleValue())));
   }
  }
  return escalar;
 }

 // genera un número aleatorio entre 0 y 1, lo guarda en la instancia del Escalar que llama
 // al método, deja el tipo de dato del Escalar de la instancia que llama al método
 public void numeroAleatorio(){
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   this._SiEscalarDouble = new Double(Math.random());
  else if (this._clase.equals(tipo2.getClass()))
   this._SiEscalarFloat = new Float((float)Math.random());
  else if (this._clase.equals(tipo3.getClass()))
   this._SiEscalarLong = new Long((long)Math.random());
  else if (this._clase.equals(tipo4.getClass()))
   this._SiEscalarInteger = new Integer((int)Math.random());
 }

 // regresa un valor redondeado, el tipo primitivo del Escalar de retorno es convertido al
 // tipo de dato primitivo definido por la instancia actual
 public Escalar redondea(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.rint(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.rint(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(this._SiEscalarLong.longValue()));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(this._SiEscalarInteger.intValue()));
  return escalar;
 }

 // regresa un valor redondeado convirtiendo el tipo primitivo del Escalar al tipo long si
 // el dato que se redondea es double, y a int si el tipo de dato primitivo que se redondea
 // es un float
 public Escalar redondea2(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Long(Math.round(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Integer(Math.round(this._SiEscalarFloat.floatValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long(Math.round(this._SiEscalarLong.longValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer(Math.round(this._SiEscalarInteger.intValue())));
  return escalar;
 }

 // calcula el seno del Escalar (instancia actual) que llama al método (rad), deja el tipo
de
 // dato como el definido por la instancia actual. El dominio de la función es de [-
inf,inf],
 // y el rango va de [-1,1]
 public Escalar seno(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.sin(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.sin(this._SiEscalarFloat.doubleValue())));
```

```
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.sin(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)Math.sin(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // calcula la tangente del Escalar (instancia actual) que llama al método (rad), deja el
 // tipo de dato como el definido por la instancia actual. El dominio de la función es de
 // [-inf,inf], y el rango va de [-inf,inf]
 public Escalar tangente(){
  Escalar escalar = new Escalar(this._clase);
  Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
  Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
  if (this._clase.equals(tipo1.getClass()))
   escalar = new Escalar(new Double(Math.tan(this._SiEscalarDouble.doubleValue())));
  else if (this._clase.equals(tipo2.getClass()))
   escalar = new Escalar(new Float((float)Math.tan(this._SiEscalarFloat.doubleValue())));
  else if (this._clase.equals(tipo3.getClass()))
   escalar = new Escalar(new Long((long)Math.tan(this._SiEscalarLong.doubleValue())));
  else if (this._clase.equals(tipo4.getClass()))
   escalar = new Escalar(new Integer((int)Math.tan(this._SiEscalarInteger.doubleValue())));
  return escalar;
 }

 // Regresa el valor de la variable instancia: _SiEscalarDouble, de la clase Escalar.
 public Double tipoBaseDouble(){
  return _SiEscalarDouble;
 }

 // Regresa el valor de la variable instancia: _SiEscalarFloat, de la clase Escalar.
 public Float tipoBaseFloat(){
  return _SiEscalarFloat;
 }

 // Regresa el valor de la variable instancia: _SiEscalarLong, de la clase Escalar.
 public Long tipoBaseLong(){
  return _SiEscalarLong;
 }

 // Regresa el valor de la variable instancia: _SiEscalarInteger, de la clase Escalar.
 public Integer tipoBaseInteger(){
  return _SiEscalarInteger;
 }

}
```

(Archivo: EscalarComplejo.java)

```
package miguel.math;
import miguel.math.Escalar;

public class EscalarComplejo {

 // Variables que contienen al escalar complejo

 Escalar _real;
 Escalar _imaginario;

 public EscalarComplejo(Escalar real, Escalar imaginario) {
  this._real = real;
  this._imaginario = imaginario;
 }
 public EscalarComplejo(Escalar real, Double imaginario) {
  this._real       = real;
  this._imaginario = new Escalar(imaginario);
 }
 public EscalarComplejo(Escalar real, Float imaginario) {
  this._real       = real;
  this._imaginario = new Escalar(imaginario);
```

```
}
public EscalarComplejo(Escalar real, Long imaginario) {
 this._real       = real;
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Escalar real, Integer imaginario) {
 this._real       = real;
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Escalar real, double imaginario) {
 this._real       = real;
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Escalar real, float imaginario) {
 this._real       = real;
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Escalar real, long imaginario) {
 this._real       = real;
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Escalar real, int imaginario) {
 this._real       = real;
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Double real, Escalar imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = imaginario;
}
public EscalarComplejo(Float real, Escalar imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = imaginario;
}
public EscalarComplejo(Long real, Escalar imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = imaginario;
}
public EscalarComplejo(Integer real, Escalar imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = imaginario;
}
public EscalarComplejo(double real, Escalar imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = imaginario;
}
public EscalarComplejo(float real, Escalar imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = imaginario;
}
public EscalarComplejo(long real, Escalar imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = imaginario;
}
public EscalarComplejo(int real, Escalar imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = imaginario;
}
public EscalarComplejo(Double real, Double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Double real, Float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Double real, Long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Double real, Integer imaginario) {
 this._real       = new Escalar(real);
```

```
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Double real, double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Double real, float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Double real, long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Double real, int imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Float real, Double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Float real, Float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Float real, Long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Float real, Integer imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Float real, double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Float real, float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Float real, long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Float real, int imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Long real, Double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Long real, Float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Long real, Long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Long real, Integer imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Long real, double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Long real, float imaginario) {
```

```
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Long real, long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Long real, int imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Integer real, Double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Integer real, Float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Integer real, Long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Integer real, Integer imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Integer real, double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Integer real, float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Integer real, long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Integer real, int imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(double real, Double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(double real, Float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(double real, Long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(double real, Integer imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(double real, double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(double real, float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(double real, long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
```

```
public EscalarComplejo(double real, int imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(float real, Double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(float real, Float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(float real, Long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(float real, Integer imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(float real, double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(float real, float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(float real, long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(float real, int imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(long real, Double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(long real, Float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(long real, Long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(long real, Integer imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(long real, double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(long real, float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(long real, long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(long real, int imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(int real, Double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
```

```
}
public EscalarComplejo(int real, Float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(int real, Long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(int real, Integer imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(int real, double imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(int real, float imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(int real, long imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(int real, int imaginario) {
 this._real       = new Escalar(real);
 this._imaginario = new Escalar(imaginario);
}
public EscalarComplejo(Class clase) {
 this._real = new Escalar(clase);
 this._imaginario = new Escalar(clase);
}

public Escalar parteReal() {
 return _real;
}

public Escalar parteImaginaria() {
 return _imaginario;
}

public Escalar magnitud() {
 Escalar resultado1,resultado2;
 resultado1 = _real.elevaAlcuadrado();
 resultado2 = _imaginario.elevaAlcuadrado();
 resultado1 = resultado1.suma(resultado2);
 resultado2 = resultado1.raizCuadrada();
 return resultado2;
}

public Escalar fase() {
 Escalar tempfase = new Escalar(this._real.getClass());
 tempfase = this._real.arcoTangente2(this._imaginario);
 return tempfase;
}

// representación cadena del EscalarComplejo
public String toString()
{
 String result = "(";
 Double  tipo1 = new Double(0.); Float   tipo2 = new Float(0.);
 Long    tipo3 = new Long(0); Integer tipo4 = new Integer(0);
 if (this._real._clase.equals(tipo1.getClass()))
  {result = result + this._real._SiEscalarDouble.toString();}
 else if (this._real._clase.equals(tipo2.getClass()))
  {result = result + this._real._SiEscalarFloat.toString();}
 else if (this._real._clase.equals(tipo3.getClass()))
  {result = result + this._real._SiEscalarLong.toString();}
 else if (this._real._clase.equals(tipo4.getClass()))
  {result = result + this._real._SiEscalarInteger.toString();}
```

```
  result = result + ",";
  if (this._imaginario._clase.equals(tipo1.getClass()))
   {result = result + this._imaginario._SiEscalarDouble.toString();}
  else if (this._imaginario._clase.equals(tipo2.getClass()))
   {result = result + this._imaginario._SiEscalarFloat.toString();}
  else if (this._imaginario._clase.equals(tipo3.getClass()))
   {result = result + this._imaginario._SiEscalarLong.toString();}
  else if (this._imaginario._clase.equals(tipo4.getClass()))
   {result = result + this._imaginario._SiEscalarInteger.toString();}
  result = result + "i)";
  return result;
 }

 // suma un escalar complejo con un escalar o un escalar complejo con otro escalar complejo
 // regresa el tipo de retorno de la instancia (escalar complejo) que llama al método
 public EscalarComplejo suma(Escalar escalar) {
  Escalar tempReal       = new Escalar(this._real._clase);
  Escalar tempImaginario = new Escalar(this._imaginario._clase);
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);
  tempComplejo._real = this._real.suma(escalar);
  tempComplejo._imaginario = this._imaginario;
  return tempComplejo;
 }
 public EscalarComplejo suma(EscalarComplejo escalarComplejo) {
  Escalar tempReal       = new Escalar(this._real._clase);
  Escalar tempImaginario = new Escalar(this._imaginario._clase);
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);
  tempComplejo._real = this._real.suma(escalarComplejo._real);
  tempComplejo._imaginario = this._imaginario.suma(escalarComplejo._imaginario);
  return tempComplejo;
 }

 // resta un escalar complejo menos un escalar o un escalar complejo menos otro escalar
complejo
 // regresa el tipo de retorno de la instancia (escalar complejo) que llama al método
 public EscalarComplejo resta(Escalar escalar) {
  Escalar tempReal       = new Escalar(this._real._clase);
  Escalar tempImaginario = new Escalar(this._imaginario._clase);
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);
  tempComplejo._real = this._real.resta(escalar);
  tempComplejo._imaginario = this._imaginario;
  return tempComplejo;
 }
 public EscalarComplejo resta(EscalarComplejo escalarComplejo) {
  Escalar tempReal       = new Escalar(this._real._clase);
  Escalar tempImaginario = new Escalar(this._imaginario._clase);
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);
  tempComplejo._real = this._real.resta(escalarComplejo._real);
  tempComplejo._imaginario = this._imaginario.resta(escalarComplejo._imaginario);
  return tempComplejo;
 }

 // multiplica un escalar complejo con un escalar o un escalar complejo con otro escalar
complejo
 // regresa el tipo de retorno de la instancia (escalar complejo) que llama al método
 public EscalarComplejo multiplica(Escalar escalar) {
  Escalar tempReal       = new Escalar(this._real._clase);
  Escalar tempImaginario = new Escalar(this._imaginario._clase);
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);
  tempComplejo._real = this._real.multiplica(escalar);
  tempComplejo._imaginario = this._imaginario.multiplica(escalar);
  return tempComplejo;
 }
 public EscalarComplejo multiplica(EscalarComplejo escalarComplejo) {
  Escalar tempReal       = new Escalar(this._real._clase);
  Escalar tempImaginario = new Escalar(this._imaginario._clase);
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);
  tempComplejo._real = this._real.multiplica(escalarComplejo._real);
  tempComplejo._real =
tempComplejo._real.resta(this._imaginario.multiplica(escalarComplejo._imaginario));
  tempComplejo._imaginario = this._imaginario.multiplica(escalarComplejo._real);
```

```
  tempComplejo._imaginario =
tempComplejo._imaginario.suma(this._real.multiplica(escalarComplejo._imaginario));
  return tempComplejo;
 }

 // divide un escalar complejo con un escalar o un escalar complejo con otro escalar
complejo
 // regresa el tipo de retorno de la instancia (escalar complejo) que llama al método
 public EscalarComplejo divide(Escalar escalar) {
  Escalar tempReal       = new Escalar(this._real._clase);
  Escalar tempImaginario = new Escalar(this._imaginario._clase);
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);
  tempComplejo._real = this._real.divide(escalar);
  tempComplejo._imaginario = this._imaginario.divide(escalar);
  return tempComplejo;
 }
 public EscalarComplejo divide(EscalarComplejo escalarComplejo) {
  Escalar tempReal       = escalarComplejo._real;
  Escalar tempImaginario = escalarComplejo._imaginario.multiplica(-1); //es el conjugado
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);
  EscalarComplejo tempComplejoNumerador   = this.multiplica(tempComplejo);
  EscalarComplejo tempComplejoDenominador = escalarComplejo.multiplica(tempComplejo);
  Escalar magnitudDenominador = tempComplejoDenominador.magnitud();
  tempComplejo = tempComplejoNumerador.divide(magnitudDenominador);
  return tempComplejo;
 }

 // obten el conjugado del número complejo
 public EscalarComplejo conjugado() {
  Escalar tempReal       = this._real;
  Escalar tempImaginario = this._imaginario.multiplica(-1); //es el conjugado
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);
  return tempComplejo;
 }

 // saca el negativo del número complejo de la instancia que llama al método
 public EscalarComplejo negativo() {
  Escalar tempReal       = this._real.multiplica(-1);
  Escalar tempImaginario = this._imaginario.multiplica(-1);
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);
  return tempComplejo;
 }

 // saca el coseno de un número complejo
 public EscalarComplejo coseno() {
  Escalar tempReal       = this._real;
  Escalar tempImaginario = this._imaginario;
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);

  tempComplejo._real = this._imaginario.multiplica(-1).exponencial().
   multiplica(this._real.coseno()).multiplica(0.5).
   suma(this._imaginario.exponencial().
   multiplica(this._real.coseno()).multiplica(0.5));

  tempComplejo._imaginario = this._imaginario.multiplica(-1).exponencial().
   multiplica(this._real.seno()).multiplica(0.5).
   suma(this._imaginario.exponencial().
   multiplica(this._real.seno()).multiplica(-0.5));
  return tempComplejo;
 }

 // saca el seno de un número complejo
 public EscalarComplejo seno() {
  Escalar tempReal       = this._real;
  Escalar tempImaginario = this._imaginario;
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);

  tempComplejo._imaginario = this._imaginario.multiplica(-1).exponencial().
   multiplica(this._real.coseno()).multiplica(-0.5).
   suma(this._imaginario.exponencial().
   multiplica(this._real.coseno()).multiplica(0.5));
```

```
  tempComplejo._real = this._imaginario.multiplica(-1).exponencial().
   multiplica(this._real.seno()).multiplica(0.5).
   suma(this._imaginario.exponencial().
   multiplica(this._real.seno()).multiplica(0.5));
  return tempComplejo;
 }

 // saca la tangente de un número complejo
 public EscalarComplejo tangente() {
  Escalar tempReal       = this._real;
  Escalar tempImaginario = this._imaginario;
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);

  tempComplejo = this.seno().divide(this.coseno());
  return tempComplejo;
 }

 // eleva al cuadrado un número complejo
 public EscalarComplejo elevaAlcuadrado() {
  Escalar tempReal       = this._real;
  Escalar tempImaginario = this._imaginario;
  EscalarComplejo tempComplejo = new EscalarComplejo(tempReal, tempImaginario);

  tempComplejo = this.multiplica(this);
  return tempComplejo;
 }

 // método que compara dos números complejos, si son iguales sus partes reales y
 // si son iguales sus partes imaginarias regresa true, si no regresa false.
 public boolean igual(EscalarComplejo otro){
  boolean res1 = this.parteReal().igual(otro.parteReal());
  boolean res2 = this.parteImaginaria().igual(otro.parteImaginaria());
  if ((res1==true) && (res2==true)) {return true;}
  else {return false;}
 }


}
```

(Archivo: Vector.java)

```
package miguel.math;
// genera un Vector como lista doble enlazada, comprueba que los objetos
// sean de la clase pasada como paramatro en el constructor de la clase.

import miguel.math.TipoList0;


public class Vector extends TipoList0{

 //tipo de clase del que es el Vector
 Class _clase;

 // constructores de la clase Vector
 public Vector(String str)
  throws IllegalArgumentException, ClassNotFoundException {
  super(Class.forName(str));
  _clase = Class.forName(str);
 }
 public Vector(int tam, Escalar escalar)
  throws IllegalArgumentException, ClassNotFoundException {
  super(escalar.getClass());
  this._clase = escalar.getClass();
  this.añadeTamaño(tam, escalar);
 }
 public Vector(int tam, EscalarComplejo escalarComplejo)
  throws IllegalArgumentException, ClassNotFoundException {
  super(escalarComplejo.getClass());
  this._clase = escalarComplejo.getClass();
```

```
  this.añadeTamaño(tam, escalarComplejo);
 }

 // método que pasa como parámetros el número (Tamaño) de elementos a añadir al Vector,
 // el valor inicial de todos los elementos añadidos es cero.
 public void añadeTamaño(int Tamaño){
  for(int i=0; i<Tamaño; i++) {
   Escalar escalar = new Escalar(0);
   EscalarComplejo escalarComplejo = new EscalarComplejo(0,0);
   if (this._clase == escalar.getClass())
    this.añadeElemento(new Escalar(this._clase));
   else if (this._clase == escalarComplejo.getClass())
    this.añadeElemento(new EscalarComplejo(this._clase));
  }
 }

 // métodos que pasan como parámetros el número (Tamaño) de elementos a añadir al Vector y
el
 // valorInicial de todos los elementos añadidos.
 public void añadeTamaño(int Tamaño, Escalar valorInicial){
  Escalar escalar = new Escalar(0);
  for(int i=0; i<Tamaño; i++){
   this.añadeElemento(valorInicial);
  }
 }
 public void añadeTamaño(int Tamaño, EscalarComplejo valorInicial){
  EscalarComplejo escalarComplejo = new EscalarComplejo(0,0);
  for(int i=0; i<Tamaño; i++){
   this.añadeElemento(valorInicial);
  }
 }

 // método que clona un Vector (copia su contenido en otra localidad de memoria)
 public synchronized Vector clon()
  throws ClassNotFoundException {
  Vector resultado = new Vector(this._clase.getName());
  ElementoLista elemento = this._cabeza;
  for (int i=0; i<this.tamaño(); i++){
   resultado.añadeElemento(elemento.dato());
   elemento = elemento.enlace_proximo();
  }
  return resultado;
 }

 // método que clona la "parteReal", la "parteImaginaria" de un Vector de EscalarComplejo en
 // un Vector Escalar, si la cadena argumento no se escribe tal cual esta entre comillas,
 // o el Vector que se clona no es complejo, entonces regresa el vector original
 public Vector clonar(String str)
  throws ClassNotFoundException {
  if (str.equals("parteReal")) {
   Escalar tipoVector1 = new Escalar(0);
   EscalarComplejo tipoVector2 = new EscalarComplejo(0,0);
   if (this._clase.equals(tipoVector1.getClass())){
    Vector temp = this.clon();
    return temp;
   } else if (this._clase.equals(tipoVector2.getClass())){
    Vector temp = new Vector(tipoVector1.getClass().getName());
    ElementoLista elemento = this._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2 = (EscalarComplejo) elemento.dato();
     temp.añadeElemento(tipoVector2.parteReal());
     elemento = elemento.enlace_proximo();
    }
    return temp;
   } else {Vector temp = this.clon();return temp;}
  } else if (str.equals("parteImaginaria")) {
   Escalar tipoVector1 = new Escalar(0);
   EscalarComplejo tipoVector2 = new EscalarComplejo(0,0);
   if (this._clase.equals(tipoVector1.getClass())){
    Vector temp = this.clon();
    return temp;
```

```
   } else if (this._clase.equals(tipoVector2.getClass())){
    Vector temp = new Vector(tipoVector1.getClass().getName());
    ElementoLista elemento = this._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2 = (EscalarComplejo) elemento.dato();
     temp.añadeElemento(tipoVector2.parteImaginaria());
     elemento = elemento.enlace_proximo();
    }
    return temp;
   } else {Vector temp = this.clon();return temp;}
  } else {Vector temp = this.clon();return temp;}
 }

 // métodos que multiplican el Vector (Escalar o EscalarComplejo) por un escalar (Escalar o
 // EscalarComplejo). El tipo de dato "básico" (Double, Float, Long, Integer)en que es
dejado
 // el Vector resultado es el que tiene el Vector que llama a este método (aunque de manera
 // general puede haber elementos de diferente tipo "basico" en el Vector)
 public Vector multiplica(Escalar escalar)
  throws ClassNotFoundException {
  Vector temp = this.clon();
  Escalar tipoVector1 = new Escalar(0);
  EscalarComplejo tipoVector2 = new EscalarComplejo(0,0);
  for(int i=0; i<temp.tamaño(); i++){
   if (temp._clase.equals(tipoVector1.getClass())){
    tipoVector1 = (Escalar) temp.ObtenElementoDe(tipoVector1,i);
    temp.ReemplazaElementoEn(tipoVector1.multiplica(escalar),i);
   } else if (temp._clase.equals(tipoVector2.getClass())){
    tipoVector2 = (EscalarComplejo) temp.ObtenElementoDe(tipoVector2,i);
    temp.ReemplazaElementoEn(tipoVector2.multiplica(escalar),i);
   }
  }
  return temp;
 }
 public Vector multiplica(EscalarComplejo escalarComplejo)
  throws ClassNotFoundException {
  Escalar tipoVector1 = new Escalar(0);
  EscalarComplejo tipoVector2 = new EscalarComplejo(0,0);
  if (this._clase.equals(tipoVector1.getClass())){
   Vector temp = new Vector(tipoVector2.getClass().getName());
   ElementoLista elemento = this._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector1 = (Escalar) elemento.dato();
    temp.añadeElemento(escalarComplejo.multiplica(tipoVector1));
    elemento = elemento.enlace_proximo();
   }
   return temp;
  } else if (this._clase.equals(tipoVector2.getClass())){
   Vector temp = this.clon();
   for(int i=0; i<this.tamaño(); i++){
    tipoVector2 = (EscalarComplejo) temp.ObtenElementoDe(tipoVector2,i);
    temp.ReemplazaElementoEn(tipoVector2.multiplica(escalarComplejo),i);
   }
   return temp;
  } else {Vector temp = this.clon();return temp;}
 }

 // métodos que dividen el Vector (Escalar o EscalarComplejo) por un escalar (Escalar o
 // EscalarComplejo). El tipo de dato "basico" (Double, Float, Long, Integer) en que es
 // dejado el Vector resultado es el que tiene el Vector que llama a este método (aunque
 // de manera general puede haber elementos de diferente tipo "basico" en el Vector)
 public Vector divide(Escalar escalar)
  throws ClassNotFoundException {
  Vector temp = this.clon();
  Escalar tipoVector1 = new Escalar(0);
  EscalarComplejo tipoVector2 = new EscalarComplejo(0,0);
  for(int i=0; i<temp.tamaño(); i++){
   if (temp._clase.equals(tipoVector1.getClass())){
    tipoVector1 = (Escalar) temp.ObtenElementoDe(tipoVector1,i);
    temp.ReemplazaElementoEn(tipoVector1.divide(escalar),i);
   } else if (temp._clase.equals(tipoVector2.getClass())){
```

```
    tipoVector2 = (EscalarComplejo) temp.ObtenElementoDe(tipoVector2,i);
    temp.ReemplazaElementoEn(tipoVector2.divide(escalar),i);
   }
  }
  return temp;
 }
 public Vector divide(EscalarComplejo escalarComplejo)
  throws ClassNotFoundException {
  Escalar tipoVector1 = new Escalar(0);
  EscalarComplejo tipoVector2 = new EscalarComplejo(0,0);
  if (this._clase.equals(tipoVector1.getClass())){
   Vector temp = new Vector(tipoVector2.getClass().getName());
   ElementoLista elemento = this._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector1 = (Escalar) elemento.dato();
    temp.añadeElemento(tipoVector1.divide(escalarComplejo));
    elemento = elemento.enlace_proximo();
   }
   return temp;
  } else if (this._clase.equals(tipoVector2.getClass())){
   Vector temp = this.clon();
   for(int i=0; i<this.tamaño(); i++){
    tipoVector2 = (EscalarComplejo) temp.ObtenElementoDe(tipoVector2,i);
    temp.ReemplazaElementoEn(tipoVector2.divide(escalarComplejo),i);
   }
   return temp;
  } else {Vector temp = this.clon();return temp;}
 }

 // método que comprueba que dos Vectores sean del mismo tamaño sin importar sus tipos de
datos
 // (Escalar o EscalarComplejo)
 public boolean mismoTamañoVectores(Vector vector2){
  if (this.tamaño() == vector2.tamaño()) return true;
  else return false;
 }

 // método que comprueba que dos Vectores sean del mismo tipo de datos (Escalar
 // o EscalarComplejo)
 public boolean mismoTipoVectores(Vector vector2){
  if (this._clase.equals(vector2._clase)) return true;
  else return false;
 }

 // concatena un Vector con otro, el parámetro (vector2) es añadido al final del Vector
 // de la instancia actual, ambos deben ser del mismo tipo, si no son del mismo tipo de
 // datos (Escalar o EscalarComplejo) se regresa sin modificación  el Vector de la
 // instancia actual que llama al método.
 public Vector concatena(Vector vector2)
  throws ClassNotFoundException {
  Vector temp = this.clon();
  if (this.mismoTipoVectores(vector2) == true){
   ElementoLista elemento = vector2._cabeza;
   for (int i=0; i<vector2.tamaño(); i++){
    temp.añadeElemento(elemento.dato());
    elemento = elemento.enlace_proximo();
   }
  }
  return temp;
 }

 // método que revierte el Vector (voltea el Vector), el ultimo elemento se convierte en
 // el primero y el primer elemento pasa a ser el último
 public void revierte()
  throws ClassNotFoundException {
  Vector vector = new Vector(this._clase.getName());
  ElementoLista elemento = this._cabeza;
  for (int i=0; i<this.tamaño(); i++){
   vector.principioElemento(elemento.dato());
   elemento = elemento.enlace_proximo();
  }
```

```
  elemento = vector._cabeza;
  for (int i=0; i<this.tamaño(); i++){
   this.ReemplazaElementoEn(elemento.dato(),i);
   elemento = elemento.enlace_proximo();
  }
 }

 // método que copia un Vector en otro a partir de una posición dada. Si el índice es una
 // localización negativa y en magnitud más grande que el tamaño del Vector a copiar
(vector2)
 // o el indice es demasiado grande, mayor que el tamaño del Vector en el que se
 // copia (instancia actual que llama al método) entonces se añade al principio y al final
 // respectivamente. Si el índice esta dentro del Vector, reemplaza a partir de esa posición
 // con el Vector parámetro (vector2), si éste es más grande que el Vector donde se
 // reemplazan los elementos, entonces los añade al Vector resultante. Ambos vectores deben
 // ser del mismo tipo de datos (Escalar o EscalarComplejo), si no regresa el Vector de la
 // instancia actual
 public Vector copiaORen(Vector vector2, int indice)
  throws ClassNotFoundException {
  Vector subVect;
  Vector temp = this.clon();
  if (indice > temp.tamaño()) indice = temp.tamaño();
  if ((indice < 0) && (Math.abs(indice) > vector2.tamaño())) indice = -(vector2.tamaño());
  if (temp.mismoTipoVectores(vector2) == true){
   if (indice < 0) {subVect = vector2.subVector(0, Math.abs(indice)-1);}
   else {subVect = vector2.subVector(0, -1);}
   ElementoLista elemento = vector2._cabeza;
   for (int i=0; i<vector2.tamaño(); i++){
    if (indice+i < temp.tamaño())
     temp.ReemplazaElementoEn(elemento.dato(),indice+i);
    else
     temp.insertaElementoEn(elemento.dato(),indice+i);
    elemento = elemento.enlace_proximo();
   }
   subVect = subVect.concatena(temp);
   return subVect;
  } else return temp;
 }

 // método que copia un Vector en otro a partir de una posición dada. Si el índice es una
 // localización negativa y en magnitud más grande que el tamaño del Vector a copiar
(vector2)
 // o el indice es demasiado grande, mayor que el tamaño del Vector en el que se
 // copia (instancia actual que llama al método) entonces se regresa un Vector vacío.
 // Si el índice esta dentro del Vector, reemplaza a partir de esa posición
 // con el Vector parámetro (vector2), si éste es más grande que el Vector donde se
 // reemplazan los elementos, entonces no se copian los restantes elementos. Ambos Vectores
 // deben ser del mismo tipo de datos (Escalar o EscalarComplejo), si no
 // regresa el Vector de la instancia actual
 public Vector copiaANDen(Vector vector2, int indice)
  throws ClassNotFoundException {
  int indiceinicio,indicefin;
  Vector subVect;
  Vector temp = this.clon();
  if (temp.mismoTipoVectores(vector2) == true){
   ElementoLista elemento = vector2._cabeza;
   for (int i=0; i<vector2.tamaño(); i++){
    temp.ReemplazaElementoEn(elemento.dato(),indice+i);
    elemento = elemento.enlace_proximo();
   }
   if (indice < 0) indiceinicio = 0; else indiceinicio = indice;
   if (indice + vector2.tamaño() < temp.tamaño()) indicefin = indice + vector2.tamaño()-1;
   else indicefin = temp.tamaño();
   subVect = temp.subVector(indiceinicio,indicefin);
   return subVect;
  }
  return temp;
 }

 // método que copia un vector en otro a partir de una posición dada. Si el índice es una
```

```
 // localización negativa y en magnitud más grande que el tamaño del vector a copiar
(vector2)
 // o el indice es demasiado grande, mayor que el tamaño del Vector en el que se
 // copia (instancia actual que llama al método) entonces se regresa el Vector sin cambio.
 // Si el índice esta dentro del Vector, reemplaza a partir de esa posición
 // con el Vector parámetro (vector2), si éste es más grande que el Vector donde se
 // reemplazan los elementos, entonces no se copian los restantes elementos. Ambos Vectores
 // deben ser del mismo tipo de datos (Escalar o EscalarComplejo), si no regresa el Vector
de
 // la instancia actual
 public Vector copiaEn(Vector vector2, int indice)
  throws ClassNotFoundException {
  Vector temp = this.clon();
  if (temp.mismoTipoVectores(vector2) == true){
   ElementoLista elemento = vector2._cabeza;
   for (int i=0; i<vector2.tamaño(); i++){
    temp.ReemplazaElementoEn(elemento.dato(),indice+i);
    elemento = elemento.enlace_proximo();
   }
  }
  return temp;
 }

 // método que copia un vector en otro.El índice siempre es cero, reemplaza a partir
 // de esa posición con el Vector parámetro (vector2), si éste es más grande que el
 // Vector donde se reemplazan los elementos, entonces no se copian los restantes
 // elementos. Ambos Vectores deben ser del mismo tipo de datos (Escalar o EscalarComplejo),
 // si no regresa el Vector de la instancia actual
 public Vector copia(Vector vector2)
  throws ClassNotFoundException {
  return this.copiaEn(vector2,0);
 }

 // método que inserta un vector en otro a partir de una posición dada, regresa falso si la
 // localización es negativa o demasiado grande (mayor que el tamaño del Vector en el que se
 // inserta), si esta dentro del Vector el indice inserta en esa posición el Vector
parámetro
 // (vector 2). Ambos Vectores deben ser del mismo tipo de datos (Escalar o
EscalarComplejo),
 // si no regresa false y no inserta nada en el Vector de la instancia actual (el que llama
 // al método)
 public boolean insertaEn(Vector vector2, int indice)
  throws ClassNotFoundException {
  boolean inserto = false;
  if (indice < 0) return inserto;
  if (this.mismoTipoVectores(vector2) == true){
   ElementoLista elemento = vector2._cabeza;
   for (int i=0; i<vector2.tamaño(); i++){
    inserto  = this.insertaElementoEn(elemento.dato(),indice+i);
    elemento = elemento.enlace_proximo();
   }
  }
  return inserto;
 }

 // método que obtiene un subVector a partir de un Vector, empezando en indiceinicio y
 // terminando en indicefin
 public Vector subVector(int indiceinicio, int indicefin)
  throws ClassNotFoundException {
  Vector resultado = new Vector(_clase.getName());
  ElementoLista elemento = _cabeza;
  indicefin++;
  if (indiceinicio < 0) indiceinicio = 0;
  if (indicefin > this.tamaño()) indicefin = this.tamaño();
  for (int i=0; i<indicefin; i++){
   if (i>=indiceinicio) resultado.añadeElemento(elemento.dato());
   elemento = elemento.enlace_proximo();
  }
  return resultado;
 }
```

```
 // método que suma un Vector con otro (ambos del mismo tamaño) y regresa el resultado para
 // guardarlo en cualquier Vector (puede ser uno de los sumandos), pueden ser de diferente
tipo
 // de datos (Escalar o EscalarComplejo), hace la operación con el tipo de los datos
"básicos"
 // del Vector de mayor presición, y luego convierte el resultado al tipo de datos "básicos"
 // la instancia que llama a este método
 public Vector suma(Vector vector2)
   throws ClassNotFoundException {
  if (this.mismoTamañoVectores(vector2) == false) {Vector temp = this.clon();return temp;}
  Escalar tipoVector1a = new Escalar(0);
  Escalar tipoVector1b = new Escalar(0);
  EscalarComplejo tipoVector2a = new EscalarComplejo(0,0);
  EscalarComplejo tipoVector2b = new EscalarComplejo(0,0);
  if (this._clase.equals(tipoVector1a.getClass())){
   if (vector2._clase.equals(tipoVector1a.getClass())){
    Vector temp = new Vector(tipoVector1a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    ElementoLista elemento2 = vector2._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector1a = (Escalar) elemento1.dato();
     tipoVector1b = (Escalar) elemento2.dato();
     temp.añadeElemento(tipoVector1a.suma(tipoVector1b));
     elemento1 = elemento1.enlace_proximo();
     elemento2 = elemento2.enlace_proximo();
    }
    return temp;
   } else if (vector2._clase.equals(tipoVector2a.getClass())) {
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    ElementoLista elemento2 = vector2._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector1a = (Escalar) elemento1.dato();
     tipoVector2a = (EscalarComplejo) elemento2.dato();
     temp.añadeElemento(tipoVector1a.suma(tipoVector2a));
     elemento1 = elemento1.enlace_proximo();
     elemento2 = elemento2.enlace_proximo();
    }
    return temp;
   } else {Vector temp = this.clon();return temp;}
  } else if (this._clase.equals(tipoVector2a.getClass())){
   if (vector2._clase.equals(tipoVector1a.getClass())){
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    ElementoLista elemento2 = vector2._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2a = (EscalarComplejo) elemento1.dato();
     tipoVector1a = (Escalar) elemento2.dato();
     temp.añadeElemento(tipoVector2a.suma(tipoVector1a));
     elemento1 = elemento1.enlace_proximo();
     elemento2 = elemento2.enlace_proximo();
    }
    return temp;
   } else if (vector2._clase.equals(tipoVector2a.getClass())) {
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    ElementoLista elemento2 = vector2._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2a = (EscalarComplejo) elemento1.dato();
     tipoVector2b = (EscalarComplejo) elemento2.dato();
     temp.añadeElemento(tipoVector2a.suma(tipoVector2b));
     elemento1 = elemento1.enlace_proximo();
     elemento2 = elemento2.enlace_proximo();
    }
    return temp;
   } else {Vector temp = this.clon();return temp;}
  } else {Vector temp = this.clon();return temp;}
 }

 // método que resta el Vector de la instancia actual - el Vector parámetro (vector2)
 // (ambos del mismo tamaño) y regresa el resultado para guardarlo en
```

```
 // cualquier Vector (puede ser uno de los sustraendos) y si son de diferente tipo
 // hace la operación con el tipo de los datos del Vector de mayor presición, y luego
 // convierte el resultado al tipo de la instancia que llama a este método
 public Vector resta(Vector vector2)
   throws ClassNotFoundException {
  if (this.mismoTamañoVectores(vector2) == false) {Vector temp = this.clon();return temp;}
  Escalar tipoVector1a = new Escalar(0);
  Escalar tipoVector1b = new Escalar(0);
  EscalarComplejo tipoVector2a = new EscalarComplejo(0,0);
  EscalarComplejo tipoVector2b = new EscalarComplejo(0,0);
  if (this._clase.equals(tipoVector1a.getClass())){
   if (vector2._clase.equals(tipoVector1a.getClass())){
    Vector temp = new Vector(tipoVector1a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    ElementoLista elemento2 = vector2._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector1a = (Escalar) elemento1.dato();
     tipoVector1b = (Escalar) elemento2.dato();
     temp.añadeElemento(tipoVector1a.resta(tipoVector1b));
     elemento1 = elemento1.enlace_proximo();
     elemento2 = elemento2.enlace_proximo();
    }
    return temp;
   } else if (vector2._clase.equals(tipoVector2a.getClass())) {
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    ElementoLista elemento2 = vector2._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector1a = (Escalar) elemento1.dato();
     tipoVector2a = (EscalarComplejo) elemento2.dato();
     temp.añadeElemento(tipoVector1a.resta(tipoVector2a));
     elemento1 = elemento1.enlace_proximo();
     elemento2 = elemento2.enlace_proximo();
    }
    return temp;
   } else {Vector temp = this.clon();return temp;}
  } else if (this._clase.equals(tipoVector2a.getClass())){
   if (vector2._clase.equals(tipoVector1a.getClass())){
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    ElementoLista elemento2 = vector2._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2a = (EscalarComplejo) elemento1.dato();
     tipoVector1a = (Escalar) elemento2.dato();
     temp.añadeElemento(tipoVector2a.resta(tipoVector1a));
     elemento1 = elemento1.enlace_proximo();
     elemento2 = elemento2.enlace_proximo();
    }
    return temp;
   } else if (vector2._clase.equals(tipoVector2a.getClass())) {
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    ElementoLista elemento2 = vector2._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2a = (EscalarComplejo) elemento1.dato();
     tipoVector2b = (EscalarComplejo) elemento2.dato();
     temp.añadeElemento(tipoVector2a.resta(tipoVector2b));
     elemento1 = elemento1.enlace_proximo();
     elemento2 = elemento2.enlace_proximo();
    }
    return temp;
   } else {Vector temp = this.clon();return temp;}
  } else {Vector temp = this.clon();return temp;}
 }

 // método que multiplica un Vector con otro (ambos del mismo tamaño, si no es del mismo
tamaño
 // regresa cero) y regresa el resultado en un escalar y si son de diferente tipo de dato
 // (Escalar o EscalarComplejo) hace la operación con el tipo de dato "básico"
 // de mayor presición, y luego convierte el resultado al tipo de la instancia que llama a
 // este método y el valor que regresa es de este tipo (Escalar o EscalarComplejo)
```

```
public Object multiplica(Vector vector2)
  throws ClassNotFoundException {
 if (this.mismoTamañoVectores(vector2) == false) {return new Escalar(0);}
 Escalar tipoVector1a = new Escalar(0);
 Escalar tipoVector1b = new Escalar(0);
 EscalarComplejo tipoVector2a = new EscalarComplejo(0,0);
 EscalarComplejo tipoVector2b = new EscalarComplejo(0,0);
 if (this._clase.equals(tipoVector1a.getClass())){
  if (vector2._clase.equals(tipoVector1a.getClass())){
   Vector temp = new Vector(tipoVector1a.getClass().getName());
   Escalar resultado = new Escalar(tipoVector1a._clase);
   ElementoLista elemento1 = this._cabeza;
   ElementoLista elemento2 = vector2._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector1a = (Escalar) elemento1.dato();
    tipoVector1b = (Escalar) elemento2.dato();
    resultado = tipoVector1a.multiplica(tipoVector1b).suma(resultado);
    elemento1 = elemento1.enlace_proximo();
    elemento2 = elemento2.enlace_proximo();
   }
   return resultado;
  } else if (vector2._clase.equals(tipoVector2a.getClass())) {
   Vector temp = new Vector(tipoVector2a.getClass().getName());
   EscalarComplejo resultado = new EscalarComplejo(tipoVector2a._real._clase);
   ElementoLista elemento1 = this._cabeza;
   ElementoLista elemento2 = vector2._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector1a = (Escalar) elemento1.dato();
    tipoVector2a = (EscalarComplejo) elemento2.dato();
    resultado = tipoVector1a.multiplica(tipoVector2a).suma(resultado);
    elemento1 = elemento1.enlace_proximo();
    elemento2 = elemento2.enlace_proximo();
   }
   return resultado;
  } else {return new EscalarComplejo(0,0);}
 } else if (this._clase.equals(tipoVector2a.getClass())){
  if (vector2._clase.equals(tipoVector1a.getClass())){
   Vector temp = new Vector(tipoVector2a.getClass().getName());
   EscalarComplejo resultado = new EscalarComplejo(tipoVector2a._real._clase);
   ElementoLista elemento1 = this._cabeza;
   ElementoLista elemento2 = vector2._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector2a = (EscalarComplejo) elemento1.dato();
    tipoVector1a = (Escalar) elemento2.dato();
    resultado = tipoVector2a.multiplica(tipoVector1a).suma(resultado);
    elemento1 = elemento1.enlace_proximo();
    elemento2 = elemento2.enlace_proximo();
   }
   return resultado;
  } else if (vector2._clase.equals(tipoVector2a.getClass())) {
   Vector temp = new Vector(tipoVector2a.getClass().getName());
   EscalarComplejo resultado = new EscalarComplejo(tipoVector2a._real._clase);
   ElementoLista elemento1 = this._cabeza;
   ElementoLista elemento2 = vector2._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector2a = (EscalarComplejo) elemento1.dato();
    tipoVector2b = (EscalarComplejo) elemento2.dato();
    resultado = tipoVector2a.multiplica(tipoVector2b).suma(resultado);
    elemento1 = elemento1.enlace_proximo();
    elemento2 = elemento2.enlace_proximo();
   }
   return resultado;
  } else {return new EscalarComplejo(0,0);}
 } else {return new EscalarComplejo(0,0);}
}

// método que crea un Vector de tipo EscalarComplejo a partir de dos Vectores Escalares,
// la instancia actual que llama al método será la parte real y el parámetro (vector2)
// será la parte imaginaria. Si no son ambos de tipo Escalar o no son del mismo tamaño
// entonces regresa el Vector de tipo EscalarComplejo vacio.
public Vector aComplejo(Vector vector2)
```

```
  throws ClassNotFoundException {
 Vector escalarcomplejo = new Vector("miguel.math.EscalarComplejo");
 Escalar tipoVector1a = new Escalar(0);
 Escalar tipoVector1b = new Escalar(0);
 if (this.mismoTamañoVectores(vector2) == false) {return escalarcomplejo;}
 if (this._clase.equals(tipoVector1a.getClass())){
  if (vector2._clase.equals(tipoVector1a.getClass())){
   ElementoLista elemento1 = this._cabeza;
   ElementoLista elemento2 = vector2._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector1a = (Escalar) elemento1.dato();
    tipoVector1b = (Escalar) elemento2.dato();
    escalarcomplejo.añadeElemento(new EscalarComplejo(tipoVector1a,tipoVector1b));
    elemento1 = elemento1.enlace_proximo();
    elemento2 = elemento2.enlace_proximo();
   }
  }
 }
 return escalarcomplejo;
}

// método que calcula la magnitud de cada uno de los elementos en el vector.
// El vector debe ser del tipo EscalarComplejo, si no lo es: regresa el vector
// de la instancia actual que llama a este método igual. NOTA: esta operación
// no esta definida en matemáticas de vectores, se implementa aquí por eficiencia
// computacional, lo cual significa ahorro de tiempo y memoria.
public Vector magnitud()
  throws ClassNotFoundException {
 Vector escalar = new Vector("miguel.math.Escalar");
 EscalarComplejo tipoVector2a = new EscalarComplejo(0,0);
 if (this._clase.equals(tipoVector2a.getClass())){
  ElementoLista elemento1 = this._cabeza;
  for(int i=0; i<this.tamaño(); i++){
   tipoVector2a = (EscalarComplejo) elemento1.dato();
   escalar.añadeElemento(tipoVector2a.magnitud());
   elemento1 = elemento1.enlace_proximo();
  }
  return escalar;
 } else {return this;}
}

// método que calcula la fase de cada uno de los elementos en el vector.
// El vector debe ser del tipo EscalarComplejo, si no lo es: regresa el vector
// de la instancia actual que llama a este método igual. NOTA: esta operación
// no esta definida en matemáticas de vectores, se implementa aquí por eficiencia
// computacional, lo cual significa ahorro de tiempo y memoria.
public Vector fase()
  throws ClassNotFoundException {
 Vector escalar = new Vector("miguel.math.Escalar");
 EscalarComplejo tipoVector2a = new EscalarComplejo(0,0);
 if (this._clase.equals(tipoVector2a.getClass())){
  ElementoLista elemento1 = this._cabeza;
  for(int i=0; i<this.tamaño(); i++){
   tipoVector2a = (EscalarComplejo) elemento1.dato();
   escalar.añadeElemento(tipoVector2a.fase());
   elemento1 = elemento1.enlace_proximo();
  }
  return escalar;
 } else {return this;}
}

// método que suma un escalar con un Vector y regresa el resultado para
// guardarlo en cualquier Vector. Pueden ser de diferente tipo (Escalar o EscalarComplejo),
// hace la operación con el tipo de los datos "básicos" del Vector o del escalar de mayor
// presición, y luego convierte el resultado al tipo de datos "básicos"
// de la instancia que llama a este método. NOTA: esta operación
// no esta definida en matemáticas de vectores, se implementa aquí por eficiencia
// computacional, lo cual significa ahorro de tiempo y memoria. Para hacer la operación
// matemáticamente apropiada, debe generarse con el escalar un vector del mismo
// tamaño que el vector con el que se desea sumar el escalar y entonces sumar los dos
// vectores.
```

```
 public Vector suma(Escalar escalar)
   throws ClassNotFoundException {
  Escalar tipoVector1a = new Escalar(0);
  EscalarComplejo tipoVector2a = new EscalarComplejo(0,0);
  if (this._clase.equals(tipoVector1a.getClass())){
   Vector temp = new Vector(tipoVector1a.getClass().getName());
   ElementoLista elemento1 = this._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector1a = (Escalar) elemento1.dato();
    temp.añadeElemento(tipoVector1a.suma(escalar));
    elemento1 = elemento1.enlace_proximo();
   }
   return temp;
  } else if (this._clase.equals(tipoVector2a.getClass())){
   Vector temp = new Vector(tipoVector2a.getClass().getName());
   ElementoLista elemento1 = this._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector2a = (EscalarComplejo) elemento1.dato();
    temp.añadeElemento(tipoVector2a.suma(escalar));
    elemento1 = elemento1.enlace_proximo();
   }
   return temp;
  } else {Vector temp = this.clon();return temp;}
 }

 // método que suma un escalar complejo con un Vector y regresa el resultado para
 // guardarlo en cualquier Vector. Pueden ser de diferente tipo (Escalar o EscalarComplejo),
 // hace la operación con el tipo de los datos "básicos" del Vector o del escalar complejo
 // de mayor presición, y luego convierte el resultado al tipo de datos "básicos"
 // de la instancia que llama a este método. NOTA: esta operación
 // no esta definida en matemáticas de vectores, se implementa aquí por eficiencia
 // computacional, lo cual significa ahorro de tiempo y memoria. Para hacer la operación
 // matemáticamente apropiada, debe generarse con el escalar un vector del mismo
 // tamaño que el vector con el que se desea sumar el escalar y entonces sumar los dos
 // vectores.
 public Vector suma(EscalarComplejo escalarcomplejo)
   throws ClassNotFoundException {
  Escalar tipoVector1a = new Escalar(0);
  EscalarComplejo tipoVector2a = new EscalarComplejo(0,0);
  if (this._clase.equals(tipoVector1a.getClass())){
   Vector temp = new Vector(tipoVector2a.getClass().getName());
   ElementoLista elemento1 = this._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector1a = (Escalar) elemento1.dato();
    temp.añadeElemento(tipoVector1a.suma(escalarcomplejo));
    elemento1 = elemento1.enlace_proximo();
   }
   return temp;
  } else if (this._clase.equals(tipoVector2a.getClass())){
   Vector temp = new Vector(tipoVector2a.getClass().getName());
   ElementoLista elemento1 = this._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector2a = (EscalarComplejo) elemento1.dato();
    temp.añadeElemento(tipoVector2a.suma(escalarcomplejo));
    elemento1 = elemento1.enlace_proximo();
   }
   return temp;
  } else {Vector temp = this.clon();return temp;}
 }

 // Este método obtiene el valor absoluto de cada uno de los elementos en el vector de tipo
 // miguel.math.Escalar, el parámetro (cadena) puede ser cualquier cadena no importa
"don'tcare".
 // Si el vector es de tipo miguel.math.EscalarComplejo se pasa como parámetro una cadena
que
 // indica a que parte de cada número complejo en el vector se le va a sacar valor absoluto,
las
 // cadenas válidas son: "parteReal", "parteImaginaria", "ambas". Si no se escriben
exactamente
 // las cadenas entonces se regresa un vector clon de la instancia que llama al método
(vector
```

```
 // actual).
 public Vector absoluto(String cadena)
   throws ClassNotFoundException {
  Escalar tipoVector1a = new Escalar(0);
  EscalarComplejo tipoVector2a = new EscalarComplejo(0,0);
  if (this._clase.equals(tipoVector1a.getClass())){
   Vector temp = new Vector(tipoVector1a.getClass().getName());
   ElementoLista elemento1 = this._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector1a = (Escalar) elemento1.dato();
    temp.añadeElemento(tipoVector1a.valorAbsoluto());
    elemento1 = elemento1.enlace_proximo();
   }
   return temp;
  } else if (this._clase.equals(tipoVector2a.getClass())){
   if (cadena.equals("parteReal")==true){
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2a = (EscalarComplejo) elemento1.dato();
     temp.añadeElemento(new
EscalarComplejo(tipoVector2a.parteReal().valorAbsoluto(),tipoVector2a.parteImaginaria()));
     elemento1 = elemento1.enlace_proximo();
    }
    return temp;
   } else if (cadena.equals("parteImaginaria")==true){
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2a = (EscalarComplejo) elemento1.dato();
     temp.añadeElemento(new
EscalarComplejo(tipoVector2a.parteReal(),tipoVector2a.parteImaginaria().valorAbsoluto()));
     elemento1 = elemento1.enlace_proximo();
    }
    return temp;
   } else if (cadena.equals("ambas")==true){
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2a = (EscalarComplejo) elemento1.dato();
     temp.añadeElemento(new
EscalarComplejo(tipoVector2a.parteReal().valorAbsoluto(),tipoVector2a.parteImaginaria().valo
rAbsoluto()));
     elemento1 = elemento1.enlace_proximo();
    }
    return temp;
   } else {Vector temp = this.clon();return temp;}
  } else {Vector temp = this.clon();return temp;}
 }

 // Este método que recorta entre un máximo y un mínimo cada uno de los elementos en el
vector
 // de tipo miguel.math.Escalar, los valores máximo y mínimo se asignan a los elementos en
el
 // vector que son mayores que máximo y menores que mínimo respectivamente, pero los tipos
 // "basicos" de máximo y mínimo son convertidos a los tipos "basicos" de los elementos en
el
 // vector a los que reemplazan. El parámetro (cadena) puede ser cualquier cadena no importa
 // "don'tcare". Si el vector es de tipo miguel.math.EscalarComplejo se pasa como parámetro
 // una cadena que indica a que parte de cada número complejo en el vector se le va a
recortar,
 // como se explicó antes. Las cadenas válidas son: "parteReal", "parteImaginaria", "ambas".
 // Este método funciona como un operador de recorte (clip) en un vector para confinar todos
 // los valores en el vector entre los valores máximo y mínimo.
 public Vector recorta(String cadena, Escalar maximo,Escalar minimo)
   throws ClassNotFoundException {
  Escalar tipoVector1a = new Escalar(0);
  Escalar tipoVector1b = new Escalar(0);
  EscalarComplejo tipoVector2a = new EscalarComplejo(0,0);
  if (this._clase.equals(tipoVector1a.getClass())){
   Vector temp = new Vector(tipoVector1a.getClass().getName());
```

```
   ElementoLista elemento1 = this._cabeza;
   for(int i=0; i<this.tamaño(); i++){
    tipoVector1a = (Escalar) elemento1.dato();
    tipoVector1a = tipoVector1a.menor(maximo);
    tipoVector1a = tipoVector1a.mayor(minimo);
    temp.añadeElemento(tipoVector1a);
    elemento1 = elemento1.enlace_proximo();
   }
   return temp;
  } else if (this._clase.equals(tipoVector2a.getClass())){
   if (cadena.equals("parteReal")==true){
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2a = (EscalarComplejo) elemento1.dato();
     tipoVector1a = tipoVector2a.parteReal();
     tipoVector1a = tipoVector1a.menor(maximo);
     tipoVector1a = tipoVector1a.mayor(minimo);
     temp.añadeElemento(new EscalarComplejo(tipoVector1a,tipoVector2a.parteImaginaria()));
     elemento1 = elemento1.enlace_proximo();
    }
    return temp;
   } else if (cadena.equals("parteImaginaria")==true){
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2a = (EscalarComplejo) elemento1.dato();
     tipoVector1a = tipoVector2a.parteImaginaria();
     tipoVector1a = tipoVector1a.menor(maximo);
     tipoVector1a = tipoVector1a.mayor(minimo);
     temp.añadeElemento(new EscalarComplejo(tipoVector2a.parteReal(),tipoVector1a));
     elemento1 = elemento1.enlace_proximo();
    }
    return temp;
   } else if (cadena.equals("ambas")==true){
    Vector temp = new Vector(tipoVector2a.getClass().getName());
    ElementoLista elemento1 = this._cabeza;
    for(int i=0; i<this.tamaño(); i++){
     tipoVector2a = (EscalarComplejo) elemento1.dato();
     tipoVector1a = tipoVector2a.parteReal();
     tipoVector1a = tipoVector1a.menor(maximo);
     tipoVector1a = tipoVector1a.mayor(minimo);
     tipoVector1b = tipoVector2a.parteImaginaria();
     tipoVector1b = tipoVector1b.menor(maximo);
     tipoVector1b = tipoVector1b.mayor(minimo);
     temp.añadeElemento(new EscalarComplejo(tipoVector1a,tipoVector1b));
     elemento1 = elemento1.enlace_proximo();
    }
    return temp;
   } else {Vector temp = this.clon();return temp;}
  } else {Vector temp = this.clon();return temp;}
 }


}
```

Related Papers.

1. Ashton, A. and A. C. Marchant, "A Scanning Interferometer for Wavefront Aberration Measurements," *Appl. Opt.*, **8,** 1953 (1969).
2. Bates, W. J., "A Wavefront Shearing Interferometer," *Proc. Phys. Soc. Lond.*, **59**, 940 (1947).
3. Briers, J. D., "Self-Compensation of Errors in a Lateral Shearing Interferometer," *Opt. Commun.*, **4**, 69 (1971).
4. Brown, D. S., "A Shearing Interferometer with Fixed Shear and Its Application to Some Problems in the Testing of Astro-Optics," *Proc. Phys. Soc. Lond. B*, **67**, 232 (1954).
5. Brown, D. S., "The Application of Shearing Interferometry to Routine Optical Testing," *J. Sci. Instrum.*, **32**, 137 (1955).
6. Cornejo-Rodriguez, A., "Ronchi test," in *Optical Shop Testing*, D. Malacara, ed. (Wiley, New York, 1992), pp. 321.
7. De Vany, A. S., "Some Aspects of Interferometric Testing and Optical Figuring," *Appl. Opt.*, **4**, 831 (1965).
8. De Vany, A. S., "Quasi-Ronchigrams as Mirror Transitive Images of Shearing Interferograms," *Appl. Opt.*, **9**, 1477 (1970).
9. Dickey, F. M. and T. M. Harder "Shearing Plate Optical Alignment," *Opt. Eng.*, **17**, 295 (1978).
10. Donath, E. and W. Carlough, "Radial Shearing Interferometer," *J. Opt. Soc. Am.*, **53**, 395 (1963).
11. Drew, R. L., "A Simplified Shearing Interferometer," *Proc. Phys. Soc. Lond. B*, **64**, 1005 (1951).
12. Dutton, D., A. Cornejo and M. Latta, "A Semiautomatic Method for Interpreting Shearing Interferograms," *Appl. Opt.*, **7**, 125 (1968).
13. Fried, D. L., "Least-squares fitting a wave-front distortion estimate to an array of phase difference measurements," *J. Opt. Soc. Am.*, **67**, 370 (1977).
14. Gates, J. W., "Reverse Shearing Interferometry," *Nature*, **176**, 359 (1955).
15. Ghiglia, D. C. and L. A. Romero, "Direct phase estimation from phase differences using fast elliptic partial differential equation solvers," *Opt. Lett.*, **14**, 1107 (1989).
16. Gollub, G. H. and C. F. Van Loan, *Matrix computations* (Johns Hopkins U. Press, Baltimore, M. D., 1990).
17. Gorshkow, V. A. and V. G. Lysenko, "Study of Aspherical Wavefronts on a Lateral Shearing Interferometer," *Sov. J. Opt. Technol.*, **47**, 689 (1980).
18. Grindel, M. W., "Testing Collimation Using Shearing Interferometry," *Proc. SPIE*, **680,** 44, (1986).
19. Hariharan, P. and D. Sen, "Triangular Path Macro-Interferometer," *J. Opt. Soc. Am.*, **49**, 1105 (1959).
20. Hariharan, P. and D. Sen, "Cyclic Shearing Interferometer," *J. Sci. Instrum.*, **37**, 374 (1960).
21. Hariharan, P., "Simple Laser Interferometer with Variable Shear and Tilt," *Appl. Opt.*, **14**, 1056 (1975).
22. Hariharan, P., "Lateral and Radial Shearing Interferometers: A Comparison," *Appl. Opt.*, **27**, 3594 (1988).
23. Hudgin, R. H., "Wave-front reconstruction for compensated imaging," *J. Opt. Soc. Am.*, **67**, 375 (1977).

24. Hunt, B. R., "Matrix formulation of the reconstruction of phase values from phase differences," *J. Opt. Soc. Am.*, **69**, 393 (1979).
25. Joenathan, C., R. K. Mohanty and R. S. Sirohi, "Lateral Shear Interferometry with Holo Shear Lens," *Opt. Commun.*, **52**, 153.(1984)
26. Kanjilal, A. K., P. N. Puntambekar, "Compact Cyclic Shearing Interferometer: Part Two," *Opt. Laser Technol.*, **16**, 311 (1984).
27. Kanjilal, A. K., P. N. Puntambekar and D. Sen, "Compact Cyclic Shearing Interferometer: Part One," *Opt. Laser Technol.*, **16**, 261 (1984).
28. Kasana, R. S. and K. J. Rosenbruch, "Determination of the Refractive Index of a Lens Using the Murty Shearing Interferometer," *Appl. Opt.*, **22**, 3526 (1983).
29. Kasana, R. S. and K. J. Rosenbruch, "The Use of a Plane Parallel Glass Plate for Determining the Lens Parameters," *Opt. Commun.*, **46**, 69 (1983).
30. Komissaruk, V. A., "Investigation of Wavefront Aberrations of Optical Systems Using Three Beam Interference," *Opt. Spectrosc.*, **16**, 571 (1964).
31. Komissaruk, V. A., "The Displacement Interferogram in the Case of a Wavefront Having Rotational Symmetry," *Sov. J. Opt. Technol.*, **36**, 456 (1969).
32. Komissaruk, V. A. and N. P. Mende, "A Polarization Interferometer with Simplified Double-Refracting Prisms," *Opt. Laser Technol.*, **13**, 151 (1981).
33. Korwan, D., "Lateral Shearing Interferogram Analysis," *Proc. SPIE*, **429**, 194 (1983).
34. Langenbeck, P., "Modifying a Shear Interferometer to Obtain a Neutral Reference Beam," *J. Opt. Soc. Am.*, **61**, 172 (1971).
35. Lohmann, A. and O. Bryngdahl, "A Lateral Wavefront Shearing Interferometer with Variable Shear," *Appl. Opt.*, **6**, 1934 (1967).
36. Malacara, D., Testing of Optical Surfaces, Ph. D. Thesis, Insitute of Optics, University of Rochester, New York (1965).
37. Malacara, D. and M. Mendez, "Lateral Shearing Interferometry of Wavefronts Having Rotational Symmetry," *Opt. Acta.*, **15**, 59 (1968).
38. Malacara, D., A. Cornejo and M. V. R. K. Murty, "A Shearing Interferometer for Convergent or Divergent Beams," *Bol. Inst. Tonantzintla*, **1**, 233 (1975).
39. Malacara, D. and S. Mallick, "Holographic Lateral Shear Interferometer," *Appl. Opt.*, **15**, 2695 (1976).
40. Malacara, D., "Twyman-Green interferometer," in *Optical Shop Testing*, D. Malacara, ed. (Wiley, New York, 1992) pp. 51.
41. Malacara, D. and S. L. DeVore, "Interferogram evaluation and wavefront fitting," in *Optical Shop Testing*, D. Malacara, ed. (Wiley, New York, 1992), pp. 455-500.
42. Mantravadi, M. V. "Newton, Fizeau, and Haidinger interferometers," in *Optical Shop Testing*, D. Malacara, ed. (Wiley, New York, 1992), pp. 1-49.
43. Mantravadi, M. V., "Lateral shearing interferometers," in *Optical Shop Testing*, D. Malacara, ed. (Wiley, New York, 1992), pp. 123-172.
44. Marroquin, J. L. and M. Rivera, "Quadratic regularization functionals for phase unwrapping," *J. Opt. Soc. Am.*, **12**, 2387 (1995).
45. Murty, M. V. R. K., "The Use of a Single Plane Parallel Plate as a Lateral Shearing Interferometer with a Visible Gas Laser Source," *Appl. Opt.*, **3**, 531 (1964).
46. Murty, M. V. R. K., "Some Modifications of the Jamin Interferometer Useful in Optical Testing," *Appl. Opt.*, **3**, 535 (1964).
47. Murty, M. V. R. K., "Fabrication of Fixed Shear Cube Type Shearing Interferometer," *Bull. Opt. Soc. India*, **3**, 55 (1969).

48. Murty, M. V. R. K., "A Compact Lateral Shearing Interferometer Based on the Michelson Interferometer," *Appl. Opt.*, **9**, 1146 (1970).
49. Murty, M. V. R. K., "A Simple Method of Introducing Tilt in the Ronchi and Cube Type of Shearing Interferometers," *Bull. Opt. Soc. India.*, **5**, 1 (1971).
50. Murty, M. V. R. K. and R. P. Shukla, "Liquid Crystal Wedge as a Polarizing Element and its Use in Shearing Interferometry," *Opt. Eng.*, **19**, 113 (1980).
51. Murty, M. V. R. K. and R. P. Shukla, "Parallel Plate Interferometer for the Precise Measurement of Radius of Curvature of a Test Plate & Focal Length of a Lens System," *Ind. J. Pure Appl. Phys.*, **21**, 587 (1983).
52. Noll, R. J., "Phase estimates from slope-type wave-front sensors," *J. Opt. Soc. Am.*, **68**, 139 (1978).
53. Nyssonen, D. and J. M. Jerke, "Lens Testing with a Simple Wavefront Shearing Interferometer,"*Appl. Opt.*, **12**, 2061 (1973).
54. Rimmer, M. P., "A Method for Evaluating Lateral Shearing Interferograms," Itek Corp. Internal Report 72-5802-1 (1972).
55. Rimmer, M. P. and J. C. Wyant, "Evaluation of Large Aberrations Using a Lateral-Shear Interferometer Having Variable Shear," *Appl. Opt.*, **14**, 142 (1975).
56. Rooyen, E. Van, "Design for a Variable Shear Prism Interferometer," *Appl. Opt.*, **7**, 2423 (1968).
57. Rooyen, E. Van and V. H. G. Houten, "Design of a Wavefront Shearing Interferometer Useful for Testing Large Aperture Optical Systems," *Appl. Opt.*, **8**, 91 (1969).
58. Saunders, J. B., "A Simplified Wavefront Shearing Interferometer and Unique Method of Evaluating any Converging Wavefront," *J. Opt. Soc. Am.*, **51**, 1467 (1961).
59. Saunders, J. B., "Measurement of Wavefronts without a Reference Standard. Part I: The Wavefront Shearing Interferometer," *J. Res. Nat. Bur. Stand.*, **65b**, 239 (1961).
60. Saunders, J. B., "Measurement of Wavefronts without a Reference Standard. Part II: The Wavefront Reversing Interferometer," *J. Res. Nat. Bur. Stand.*, **66b**, 29 (1962).
61. Saunders, J. B., "Wavefront Shearing Prism Interferometer," *J. Res. Nat. Bur. Stand.*, **68C**, 155 (1964)
62. Saunders, J. B., "Some Applications of the Wavefront Shearing Interferometer," *Jap. J. Appl. Phys.*, **4**, Supp. 1, 99 (1965).
63. Saunders, J. B., "A Simple Inexpensive Wavefront Shearing Interferometer," *Appl. Opt.*, **6**, 1581 (1967).
64. Saunders, J. B., "A Simple Interferometric Method for Workshop Testing of Optics," *Appl. Opt.*, **9**, 1623 (1970).
65. Schwider, J., "Superposition Fringe Shear Interferometry," *Appl. Opt.*, **19**, 4233 (1980).
66. Schwider, J., "Continuous Lateral Shearing Interferometer," *Appl. Opt.*, **23**, 4403 (1984).
67. Sen, D. and P. N. Puntambekar, "Shearing Interferometer for Testing Corner Cubes and Right Angle Prisms," *Appl. Opt.*, **5,** 1009 (1966).
68. Sirohi, R. S. and M. P. Kothiyal, "Double Wedge Plate Shearing Interferometer for Collimation Test," *Appl. Opt.*, **26**, 4054 (1987).
69. Takajo, H. and T. Takahashi, "Least-squares phase estimation from the phase difference," *J. Opt. Soc. Am. A*, **5**, 416 (1988).
70. Thikonow, A. N., "Solution of incorrectly formulated problems and the regularization method," *Sov. Math. Dokl.*, **4**, 1035 (1963).

71. Venkata, B. and D. P. Juyal, "A 10μm CO_2 Laser Interferometer Using Shearing Technique," *Appl. Opt.*, **25**, 764 (1986).
72. Wan D. S. and D. T. Lin, "Ronchi test and a new phase reduction algorithm," *Appl. Opt.*, **29**, 3255 (1988).
73. Wyant, J. C., "Double Frequency Grating Lateral Shear Interferometer," *Appl. Opt.*, **12**, 2057 (1973).
74. Yokozeki, S. and T. Suzuky, "Shearing Interferometer Using the Grating as the Beam Splitter," *Appl. Opt.*, **10**, 1575 (1971).

Internet address, where we can find bibliographic and papers related:

1. pac.carl.org
2. spie.org/search