UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Máster Universitario de Ingeniería en Telecomunicación.

Trabajo Fin de Máster

Desarrollo de una Herramienta Docente para la Simulación de la Propagación en Fibras Ópticas.

Jorge Florentino Durán Poblador

Convocatoria Mayo-Junio, 2016

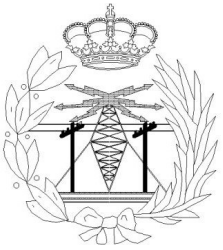
UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Máster Universitario de Ingeniería en Telecomunicación.



Trabajo Fin de Máster

Desarrollo de Una Herramienta Docente Para la Simulación de la Propagación en Fibras Ópticas.

Autor: Jorge Florentino Durán Poblador.

Tutor: Rafael Gómez Alcalá.

Tribunal Calificador

Presidente: Jesús Rubio Ruiz.

Secretario:Juan Francisco Izquierdo.

Vocal: Yolanda Campos Sierra .

Quiero dedicar este trabajo con el que concluyo el máster de Telecomunicaciones, porque se le agotó la vida cuando aún tenía mucho que vivir y se fue en medio de trabajos, y exámenes, y clases y exposiciones sin que pudiera saber que se me iba, a mi madre que fue guía en momentos tristes y referente en los alegres, que dirigió mis primeros pasos por los intrincados caminos que nos asolan. Que Dios la tenga en su seno y que su llama repleta de talento y brillo nunca se apague ni se aparte de nuestro lado.

A Nesi Poblador.

Índice general

Índice de figuras

Índice de cuadros

Abstract

To perform simulation of pulse propagation in optical fibers by solving the Schrödinger equation both scalar and vector is necessary the incorporation of methods of numerical approximation such as the split-step Fourier symmetrized and techniques that increase efficiency of calculation and leverage the resources of the machine to the maximum possible, such as CUDA library for parallelizing algorithms using techniques GPGPU.

Keywords

Parallelization, GPGPU, kernel, threads, blocks, pulse, fiber optic, soliton, dispersion, nonlinearity, attenuation, chirpeo, peak potential, FWHM, GVD, TOD, XPM, SPM, polarization, birefringence, ellipticity, phase, scalar equations, coupled equations.

Introduction

A field study of crucial importance in the development of optical communications is the propagation of the optical pulse in different media simulating the conditions of the optical fibers or free space, for which advanced computational and mathematical methods of numerical resolution are needed. Among them, those that solve the differential equation of pulse propagation in temporal component require some computing capabilities important that can be minimized through systems parallelization able to perform several procedure at once or divide a single procedure to be performed in multiple processors simultaneously. Thus, a sharing of resources that streamlines calculations and minimizes processing time is achieved. Within these methods are those using computing capabilities of modern graphics cards for general purpose calculations, it is the paradigm called GPGPU (General porpuse Computing on GPU).

Objectives

The main objective of this work is the adaptation of the PyOFTK library in its development of pulse propagation in optical fibers, for educational use in the laboratory and as a basis for the introduction to the study of physical phenomena that govern it. To this parallelization the split-step Fourier method symmetrized for the resolution scalar Schöringer equation describing the propagation of an optical pulse through a nonlinear dispersive optical fiber. Various examples and practices teaching purpose where the resolution of this equation both sequentially and in parallel be drawn is used then. As a complementary objective will be a study or analysis of time spent on each of the methods: sequential in cpu, gpu parallel CUDA scheduled. The same steps for the development of a method to solve the equations of Schödinger coupled, or vector equations with distinction birefringence of the fiber in the two axes of polarization, for which the SSPROP program developed by the University Maryland will be used.

Capítulo 1

Background/State of the Art

Among the fields of study in which we want to guide this work, there are two distinct that interact to obtain the required results. First, parallel computing that minimizes computation times and the processing load, and within it, oriented architectures computing general purpose graphics processing units. Second, the propagation of optical pulse in nonlinear fibers and dispersive for what is used, as has already been mentioned, the PyOFTK library based on the SSPROP program conducted by the University of Maryland [9], and within this, the study and solution of the Schrödinger equation, highlighting the generation of stable solitons that allow the spread of information in telecommunication systems over long distances. Therefore, the state of the art will be divided into the following sections.

1.1. General Purpose GPU Architectures (GPGPU)

[2]

For the study and development of techniques and procedures in teaching laboratories, it has positioned the Python programming language as one of the most popular because of its simplicity and ruggedness that gives it a versatility lacking other languages that operate at lowest level C , C ++, java, etc.

CUDA technology, designed by the company Nvidia graphics devices on computers to take advantage of the resources offered by the gpu of their graphics cards, offers a number of features to operate in parallel with the CPU using the programming with threads.

To deal with some credit multiparalelism techniques with use of GPU, it is necessary to understand how previously incorporated graphics card and how it is used to combine use with central processing units, CPU. To do this, it is to divide this section into two dependent and crucial for the development of work sections: Structure of a GPU and GPU computing techniques general purpose, GPGPU.

Traditionally graphics processors were focused on rendering, or transformation of a 3D image in another 2D through specialized steps executed in a predetermined order. This system was given the name *pipeline*, for programming with pipes, and is segmented into four main stages:

- Transforming vertices.
- Coupling primitive and rasterization.
- Color and textures on fragments.

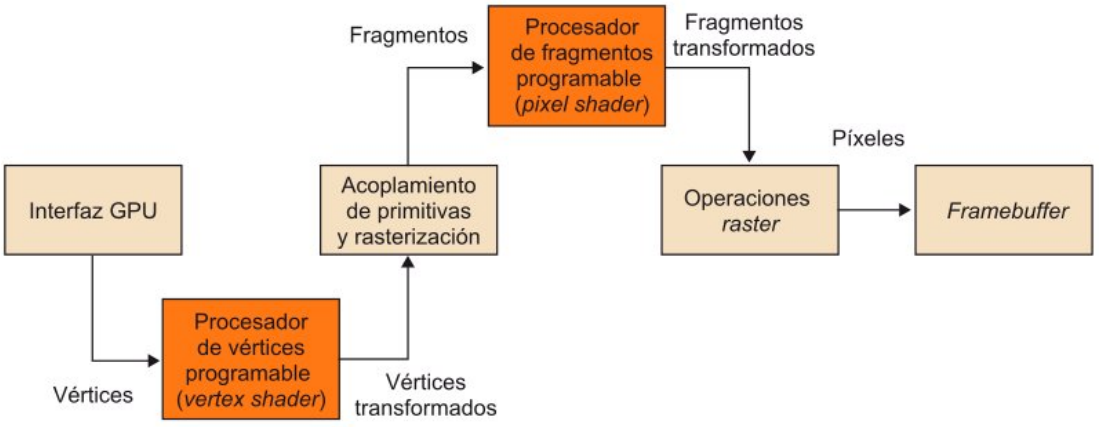


Figura 1.1: Programmable graphics pipeline.[2]

- Raster Operations.

Graphics processors allow use multiple programmable units, but typically are the vertex processing and fragment processing those dedicated to programming itself.

Both processors have a very similar operation.

- Dedicated to *vertices* processing carried out several steps since the gpu captures the image in 3D:

  1. Load the attributes associated with the vertices in three different types of internal registers:

     a) Vertices attribute information for each vertex.

     b) Temporary to provisional calculations.

     c) Output, where the new attributes are stored transformed.

  2. Execute Instruction program sequentially each located in a reserved video memory area. The operations to be carried out include:

     a) floating point mathematical operations on vectors.

b) Operations with hardware for denial of vectors and swizzling (conversion of references based on names pointers to memory locations).

c) Exponential, logarithmic and trigonometric functions.

d) flow control operations to implement loops and conditionals.

Vertex processors that enable these transactions operate in SIMD or MIMD mode.

- fragments programmable processors added to the above operations, others on textures that provide access to images by using specific coordinates that return the sign read. Only they work in SIMD mode.

All these operations performed by the pipeline and to the specification of objects, require the use of specific libraries, generally independent of the hardware used and implemented on various platforms, features and code portability. Among the most popular OpenGL and Direct3D are; and other alternatives may be SDL, Allegro and Render Ware.

Because the texture cache memory is read-only, the system can access your data an unlimited number of times but can not change them until reading data from the DRAM memory. This makes the use of traditional pipelines for general purpose programming.

In modern graphics it tends to implement a kind of advanced calculations accommodate both processor fragments vertices. This unified view has led to the widespread use of GPU computing and purposes global development multiparallelism with which lighten the burden of calculating the CPU.

To meet the growing demand in the video game industry much more complex processed to increase the sense of realism and allow real-time interaction, designers resorted initially to significantly increase the clock speed of the processor what

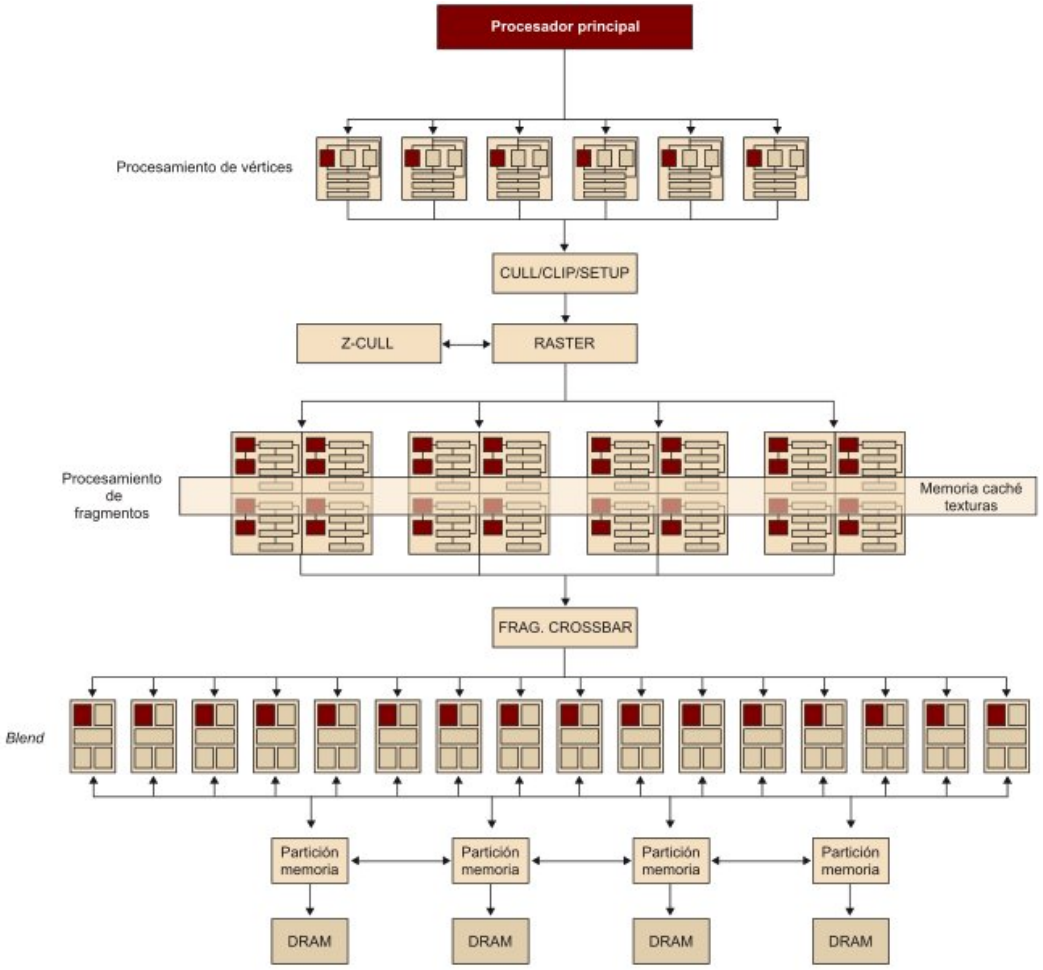


Figura 1.2: Structure of a GPU Nvidia GeForce6. [2]

had rigged a set of handicaps are difficult to resolve, such as the physical limit on the integration of transistors, high power density with temperature limitations and energy of the CMOS circuits, and limitations on the level of parallelism with the consequent increase in the size of the pipeline that impoverish yields .

To address these limitations, the industry chose to confer graphics cards to devices of a large number of nuclei with which prioritize parallel programming on the sequential saving energy and therefore runtime. This has enabled, as progress has been made in this philosophy massive parallelization of data in gpus, the

difference in performance with respect to CPU utilization has increased considerably:

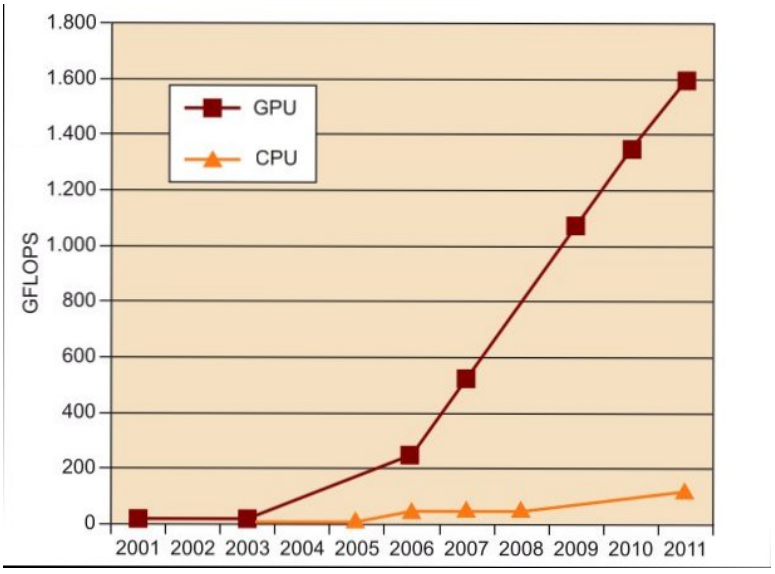


Figura 1.3: GPU vs CPU.[2] Pág 18.

The evolution in the programming of the GPU to general purpose mainly because the processors of the graphics of early 2000 using programmable arithmetic units (shakers) fully controlled by programmers to return the color of each pixel on the screen and manipulate textures. The researchers realized that they could use these units to handle any type of numerical data with which to increase performance, so that many efforts were dedicated in developing interfaces and environments of general purpose programming. This new programming technology was called GPGPU, general-purpose computing on graphics processing unit. Thus you can establish a comparison between the CPU memory read and reading textures on the GPU.

The writing is performed by the rendering process or copying data *framebuffer* to texture cache memory, declaring them as a set of data organized in one, two or three dimensions.

Each core has to process a complete data stream before the next kernel can start executing the resulting data, which is performed with the technique *render-to-texture* GPU similarly to how you would in memory of the CPU, except that the operation in cache can not be stored in the same, since it is read only. This will require to move the partial results of the operation to DRAM main memory from where fragments returned to the processor to perform new operations.

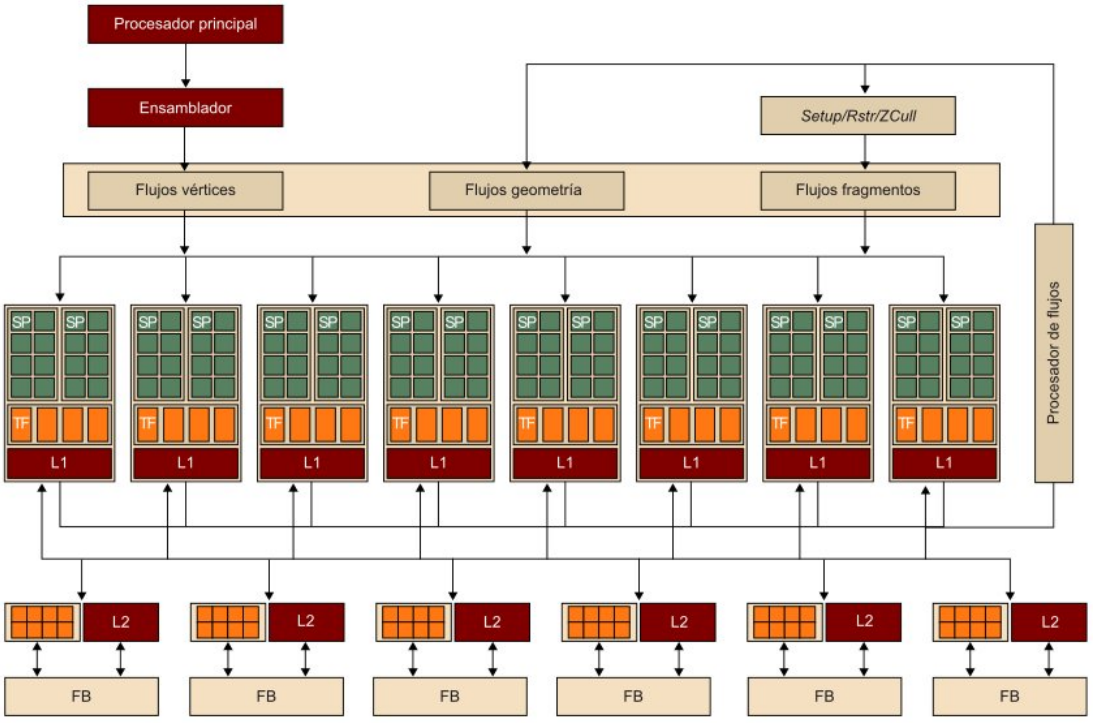


Figura 1.4: Tesla unified architecture, Nvidia G80.

To improve these operations, since shared process technology where the same processor can execute operations vertices identification and segmentation, graphs have been developed aimed at the general computer such as Nvidia Tesla. Specifically, the Nvidia G80 with architecture Tesla and adjusted tandard IEEE 754, was a quantum leap in graphics processing power and a substantial increase in performance, which led to the increase in computing power floating-point influence

significantly in use in general purpose computing.

The number of pipeline stages is considerably reduced and passed a sequential model to a cyclic pattern in the processing of the shaders.

In a first step the data is distributed by a specialized hardware so that it can be used as many functional units. In a second step, the output data are governed by a global controller flow to perform calculations in a coordinated manner and to select and type (vertex, segment or geometry) going to each processing unit GPU.

The processing core shaders comprises eight processing blocks that, in turn, are formed by sixteen main processing units (*streaming processor, SP*). Each block has a flow planner is responsible for deciding the internal management of data cache Level 1 (L1), access units and own textures filtering.

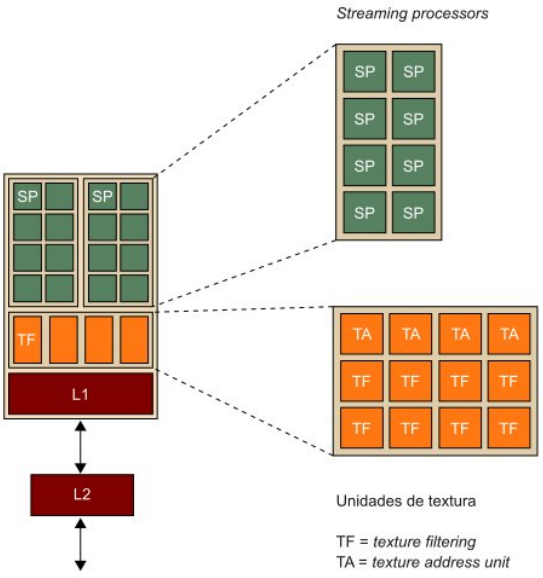


Figura 1.5: Structure of a block in a GPU G80 shaders.

The SP perform mathematical operations or routing and data transfer memory. Each is an arithmetic and logic, ALU unit, which operates on 32-bit precision,

following the IEEE 754 standard.

In turn, each block is composed of two groups of eight SP so that the internal scheduler has the same operation for both, each taking a certain number of clock cycles depending on the type of flow is running.

In addition to the three types of previously indicated flows, blocks allow the execution of flows data transfer memory, which have TF (texture filtering) and TA (texture addressing) units because of overlap as possible transfer of data. They can also share read-only information with other blocks through the second level cache (L2). Therefore, a block has four types of memory: A set of records of 32 local bits and three read-only caches (data or shared, constant and textures).

The main memory gets a combined 384-bit interface divided into six parts, each with a 64-bit interface. In this way data transfer speeds between very high central memory and processor are achieved.

The latest GPU implementations maximize the area dedicated to the ongoing implementation and floating point performance. Thus, *Fermi technology* provides a level greater than previous technologies with the GPU composed of a uniform computational units with little additional support elements set abstraction.

This structure is formed by a set of *streamming multiprocesors (SM)* with sharing L2 cache. In turn, each SM is composed of 32-48 cores (Cores) depending on the type of chip, 16 units load and store (LD/ST), 4-8 special function units (SFUs), a block of 64 Kb high speed in the chip and an interface for the L2 cache. Each core can execute an integer or floating point instruction per clock cycle. Also includes an interface with the central processor, a scheduler flows and multiple interfaces DRAM. [4]

Each *core CUDA* SM manages an integer operations and floating point for what has the corresponding units. The ALU supports 32-bit precision for all mathematical and logical instructions consistent with the requirements of the

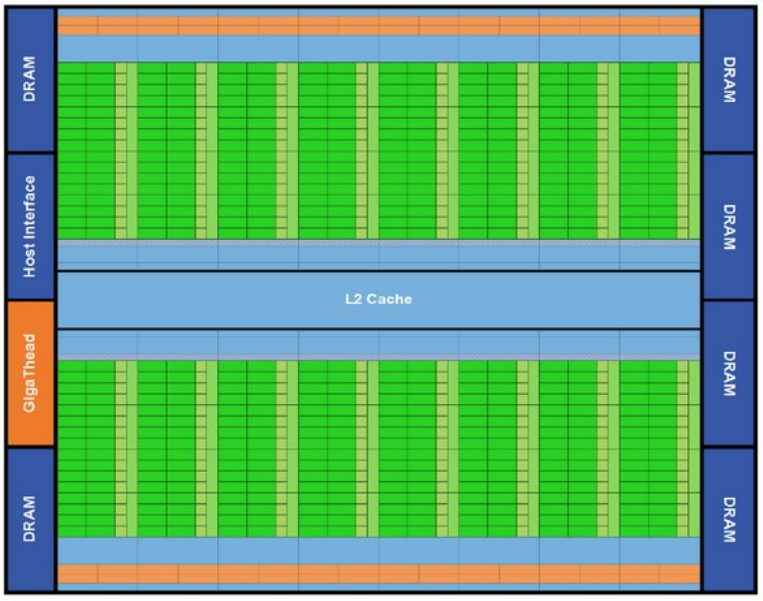


Figura 1.6: Fermi architecture of a block. [4]

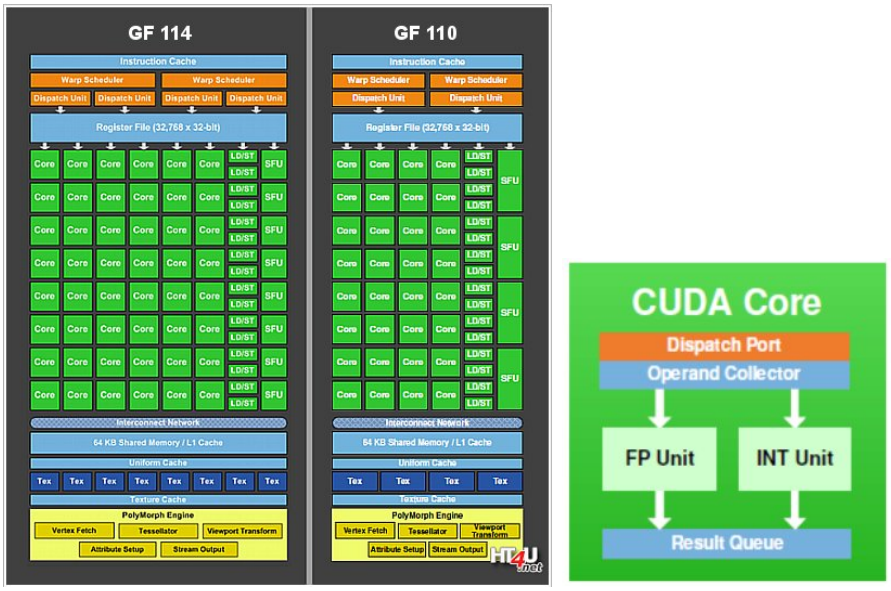


Figura 1.7: Streaming structure of a mulitprocessor and a CUDA Core. [4]

programming language used. It is also optimized to efficiently support 64-bit extended precision operations.

The GPU will be used throughout this work is the *Nvidia GeForce GTX 560 Ti*, with belonging to the Fermi GF114 chip technology. This graph was taken in March 2011 and is oriented parallel programming with CUDA library structure.

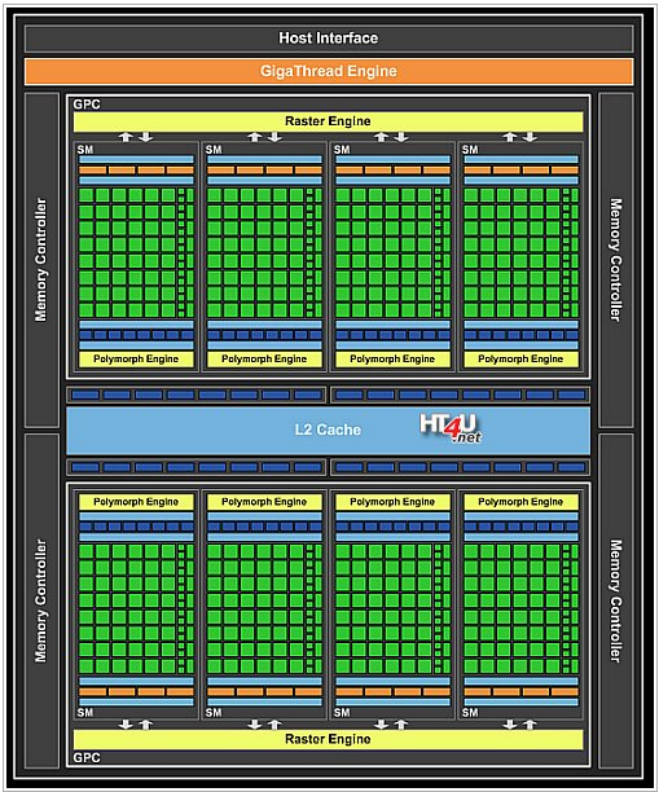


Figura 1.8: GPU GF114 structure. NVIDIA GeForce GTX 560 TI im Test. [6]

1.1.1. Programming Techniques general purpose GPU

As has become more necessary to use the extended computing load sharing and energy savings in complex calculations and simulation techniques, designers of graphics cards have oriented manufacturing GPU to multiparallelism and sharing processes with CPU also developing a range of libraries and methods for work in this field.

GPU Engine Specs:		
Graphics card version	GTX 560 Ti	GTX 560 Ti(448 cores)
CUDA Cores	384	448
Streamming Multiprocesors SM	8	8
Cores per SM	48	56
Graphics Clock (MHz)	822	732
Processor Clock (MHz)	1645	1464
Texture Fill Rate (billion/sec)	52.5	41
Memory Specs:		
Memory Clock (MHz)	4008 Gbps	1900
Standard Memory Config	1024	1280
Memory Interface	GDDR5	GDDR5
Memory Interface Width	256-bit	320-bit
Memory Bandwidth (GB/sec)	128	152
Feature Support:		
OpenGL	4.1	4.1
Bus Support	PCI-E 2.0 x 16	PCI-E 2.0 x 16
Certified for Windows 7	Yes	Yes
Supported Technologies	3D Vision, Surround, CUDA, PhysX, SLI	3D Vision, Surround, CUDA, PhysX, SLI
Microsoft DirectX	12 API	12 API
SLI Options	2-way2a	3-way2b
Display Support:		
Multi Monitor	Yes	Yes
Maximum Digital Resolution	2560x1600	2560x1600
Maximum VGA Resolution	2048x1536	2048x1536
HDCP	Yes	Yes
HDMI3	Yes	Yes
Standard Display Connectors	Mini HDMI, Two Dual Link DVI	Two Dual Link DVI-I, Mini HDMI
Audio Input for HDMI	Internal	Internal
Standard Graphics Card Dimensions:		
Length	9 inches	10.5 inches
Height	4.376 inches (111 mm)	4.376 inches (111 mm)
Width	Dual-slot	Dual-slot
Thermal and Power Specs:		
Maximum GPU Temperature	99 C	97 C
Maximum Graphics Card Power	170 W	210 W
Minimum System Power Requirement	500 W	550 W
Supplementary Power Connectors	Two 6-pin	Two 6-pin

Cuadro 1.1: Specifications for the GeForce GTX 560 Ti.

The most widespread techniques today were developed by very different but with a common objective reasons. First, *CUDA* was created by the company NVIDIA to output their products in the emerging field of GPGPU.

CUDA

Compute Unifed Device Architecture includes both architecture specifications as a specific programming model derived from C.

Overall programming CUDA is oriented to the use of threads or data streams encapsulated into blocks of two or three dimensions, which in turn are grouped in cells or GRID. A block can consist of one, two or more SM depending on the concrete implementation of the device and a SM can provide until eight blocks. Each SM is composed of a set of cores CUDA or SP (streamming procesors), sharing control logic and instruction cache. The global GPU memory is DRAM type GDDR (graphics double data rate) oriented graphics handling and storage. However, when used for general purpose use data functions as an external memory but high bandwidth high latency period compared to the system memory.

G80 graphical view in the previous section supports up to 768 SM flows by making a total of over 12000 streams per chip. The GT200 architecture, on the contrary, to have 240 massively parallel cores, can withstand up to 1024 flows by SM and in total, more than 30000 streams per chip.

CUDA belongs to the SIMD model, *Single Instruction stream, Multiple Data Stream* based on applying a single instruction (arithmetic) to a variable number of data simultaneously.

A CUDA program consists of several phases executed by the CPU or the GPU depending on the level of parallelism required.

NVIDIA compiler *(nvcc)* provides the code on the CPU (in ANSI C) and device (GPU) during the process. Although the code on the GPU is also ANSI

C, it has extensions to define data parallelization functions called *kernel*. For the code to run on the GPU must be compiled again with *nvcc*. If there were no available devices or kernel more appropriate for the CPU, you can run it using tools provided by CUDA. The type of file that handles CUDA to compile the code is very varied and used one or the other depending on the type of deployment you want to perform:

.cu	Código fuente CUDA con código de la CPU y funciones de la GPU.
.cup	Código fuente CUDA procesado.
.c	Código fuente C.
.cc, .cxx, .cpp	Código fuente C++.
.gpu	Archivo intermedio GPU.
.ptx	Archivo ensamblador intermedio ptx.
.o, .obj	Archivo de objeto.
.a, .lib	Archivo de biblioteca.
.res	Archivo de recurso.
.so	Archivo de objeto compartido

Cuadro 1.2: Input files on a CUDA compiler.

To make this process must follow specific steps in the DRAM memory device:

1. Reserve memory space on the GPU..
2. Transfer data from the CPU to the memory space previously reserved in the GPU.
3. Invoking kernel execution.
4. Transfer the results of running the kernel to the central processor.
5. Cleanup of GPU memory.

Memories global and constant type are those in which the main processor can transfer and receive data bidirectionally, by grid. Since the GPU can access memory in different ways:

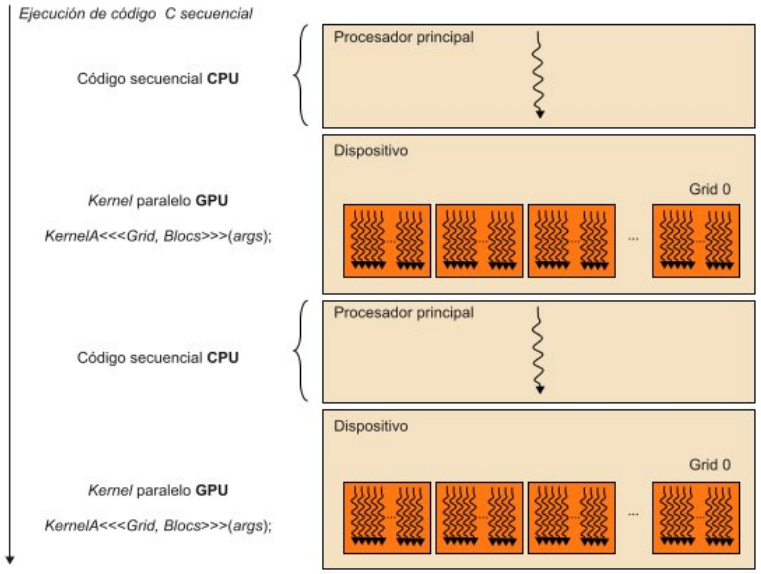


Figura 1.9: Process of compiling a code using Kernel.

- By grid:
  - · Read / write access to global memory.
  - · Read only constant memory.
- By flow (thread):
  - · Read / write access to the records.
  - · Read / write access to local memory.
- Per block:
  - · Read / write access to shared memory.

With the function *cudaMalloc()* allocates global memory interface device. It has two parameters: the first to give the address of the pointer to the object once allocated memory space; the second to indicate the object's size in bytes.

With the *cudaMemset()* function initializes a given value. Three parameters: address, value and quantity.

With the *cudaFree()* function interface frees the memory space previously

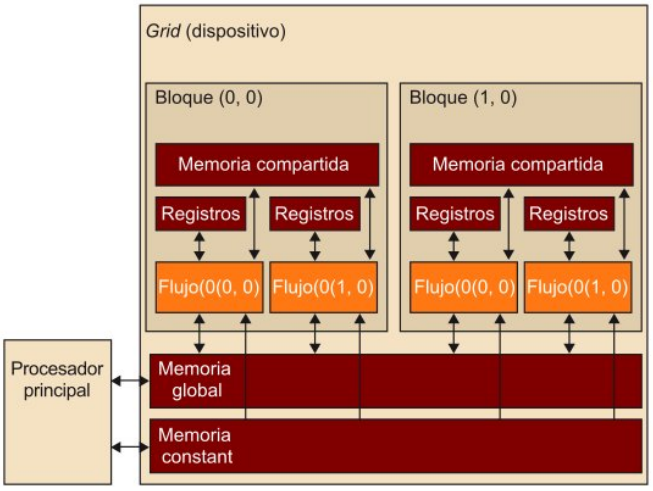


Figura 1.10: CUDA memory processes.

assigned to the object.

With the *cudaMemcpy()* function interface transfers the required data from the CPU to the device (*cudaMemcpyHostToDevice*), transfers the data from the CPU memory to the same memory of the CPU (*cudaMemcpyHostToHost*) transfers data from the device memory to the CPU memory (*cudaMemcpyDeviceToHost*) and transfers data from the memory of the GPU itself (*cudaMemcpyDeviceToDevice*).

These low-level functions are implicit in the generations of api both PyCUDA and Reikna, which are the libraries that are to be used in carrying out the simulations, greatly simplifying the process of allocating and freeing memory gpu.

The different types of variables using the GPU memories are defined in the following table:

With the keyword *_Global_* predating the name of a function, we are indicating that it is running on a *kernel* and must be called from the CPU to reserve memory space thus generating the corresponding grid.

With the keyword *_device_*, the declared function (CUDA device function) runs

Declaración de variables	Tipo de memoria	Ámbito	Ciclo de vida
Por defecto	Registro	Flujo	kernel
Vectores por defecto	Local	Flujo	kernel
shared, int SharedVar;	Compartida	Bloque	Bloque
device, int GlobalVar;	Global	Grid y host	Aplicación
constant, int ConstVar;	Constante	Grid y host	Aplicación

Cuadro 1.3: Types of variables used in the memories of the GPU.

exclusively on the CUDA device and can only be called from *kernel* or another depending on the GPU. They can not have or recursive calls, or indirectly through pointers.

With the keyword *_host_* we indicate that the function is a C function executed in the central processor, which can only be called from another function of it. If nothing is specified all functions in CUDA belong to the central processor.

When the keywords *_device_* and *_host_* are used simultaneously in the declaration of a variable, the processor makes two copies, one that runs on the host and the other does in the device.

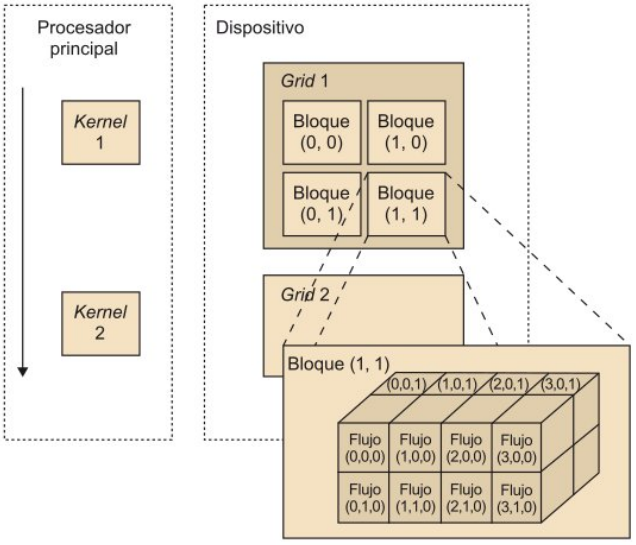


Figura 1.11: Data structure used by CUDA.

Because a *kernel* can be executed by a large number of threads, you must assign an identifier that differentiates one type of threads of others. To do CUDA has keywords to reference them, ejm.:*threadIdx.x*. During execution, the identifier is replaced by the value it deserves.

A grid consists of a variable number of threads organized hierarchically in two levels. The top level contains one or more blocks of threads.

All blocks of a grid having the same number of threads and must be organized the same way.

Each block within a grid has a single coordinate in a two-dimensional space specified by the keywords *blockIdx.x and blockIdx.y*.

A block is arranged in a three dimensional space with a maximum of 512 threads.

The coordinates of the threads within a block are identified by three indices: *threadIdx.x, threadIdx.y and threadIdx.z*.

When a call to an *kernel*, the processor must first indicate the dimensions of the grid and block within the grid it is performed. The data type using both parameters is *dim3*, ie a three-dimensional vector therefore be indicated the three numbers for each dimension even though the grid has only two dimensions. The third dimension of the grid in the vector *dimGrid* is indifferent, any number can be used. The block will use the vector *dimBlock* to indicate its dimensions.

When you call *kernel* must indicate the dimensions of the grid and blocks it contains. In the library Pycuda this is not necessary, by default assigns the largest possible number of threads to each block, resizing the plot in the most comfortable way to work for the kernel. Also, optionally, you can specify the number of bytes to reserve in shared memory via *Ns* is of type *size_t* This is done as follows:

```
dim3 dimGrid(xg,yg,0);
```

```
dim3 dimBlock(xb,yb,zb);

size_t Ns = 200;

Func <<< dimGrid;   Ns, dimBlock >>> (parameter)
```

Within an execution it is also possible to synchronize threads not to start a process not end until all the above process. The function used for this purpose is *_syncthreads()*.

These systems "low level"have been overcome in the development of algorithms for a higher level of abstraction, where computing becomes independent of the number of grid, wires or blocks through the selection of a particular context. PyCUDA default, if the Python programming language is used, uses a file included in the library with the specific characteristics among which, the type of dispositvo and threads to be used in each run. If the file is not used by default, you can specify the device type in the code including this its own characteristics. This is the procedure to be used for the production of our parallelized algorithms.

1.2. Advanced Optical Systems.

1.2.1. Evolution of Optical Communication Systems. [12]

After the invention of the laser in 1960, trying to exploit the superior ability of information regarding other previous transmission systems (tenfold to microwaves), they were conducted countless tests through atmospheric optical spectrum. However, due to its high cost and large attenuation problems caused by atmospheric agents such as clouds, fog, rain, dust ... these systems never were implemented. By then he had already begun to study the optical fiber as a new transmission path guided with very mixed results, since in those very early times as the sixties still had attenuations of the order of the 1000dB/Km, unsustainable

to carry out a satisfactory transmission of information. It was not until 1970 when these losses were achieved reduced below 20dB/km, at the same time GaAs semiconductor lasers were developed with performance at room temperature. These two events accounted for the starting signal for optical communications as we know them today.

From that moment and until the date of today's optical communications systems have advanced at a rate of double the product L · B (L = length of the fiber, B = capacity in bps) year after year, so that they can make five divisions or generations depending on the progress involved in these developments:

- In the first generation optical systems, already available between 1977 and 1979, operating over multimode fiber at wavelengths near 850nm wave front window with lasers GaAs semiconductor at speeds of 34 to 45 Mbps with capability to transmit up to 10 km without the use of repeaters, compared to coaxial cable needed to place one every km, which meant the gradual replacement of one another in the long-distance lines. It is, furthermore, that this distance between reptidores could be increased considerably if the optical systems would work in the 1310 nm region, the second window or wavelength of zero dispersion, where the fiber exhibited attenuation below 1dB/km checked. This led to the development of InGaAsP semiconductor lasers and detectors operating this system.

- It was in the second generation of optical systems in the early eighties, when carried out such progress, even though the transmission rate in multimode fibers was limited below 100Mbps. With the advent of the single-mode fiber in 1981, which operated beam to a single length of more channeled in the core wave transmissions up to 2Gbps on above 40km distances were achieved and thus for 1988 were marketed systems operating at 1.7 Gbps at distances

up to 50km. Still, these losses caused a major hindrance in transmissions over very long distances and it was not until the migration to third window, 1550nm, in 1979 they managed to reduce approximately 0.2 dB/km with the added handicap of the great dispersion of the pulse.

- In the third generation of optical systems this problem could be solved by creating zero-dispersion shifted fiber at 1550nm or limiting the spectrum of lasers to a single longitudinal mode. And so, that laboratory experiments were performed with rates up to 4 Gbps over distances exceeding 100 km in 1989, which led to systems operating at 2.5Gbps they were commercially available as early as 1992; then a 10 Gbps systems were introduced. The big disadvantage of these systems was the need to use optoelectronic repeaters to regenerate the signal periodically, which caused occasional delays, limiting transmission speeds, etc, and it was not until the advent of EDFAs (Erbium Doped Fiber Amplifier) in 1990 direct optical signals amplifications were achieved greatly reducing transmission times and synchronization failures.

- In fourth generation systems WDM technique effective optical amplification access without signal separation emerged, which led to a significant reduction in transmission costs. Thanks to this could double the capacity of the system every half a year while with the use of EDFAs, transmissions at higher than 11000km 5Gbps speeds, optimal for transoceanic communications distances were achieved.

- In the fifth generation of optical systems, it seeks to increase the number of possible channels and transmission rates of each through the appearance of photonic integrated circuits and silicon chips that will allow the implementation of the electronics and optics in one chip. Within these systems can

include both DWDM as OTDM, made possible by the advent of ultrashort pulse lasers that minimize the effects of ISI and aliassing respectively. When getting very stable Femtosecond pulses of the order of the distance between pulses guard can be minimized considerably which favors, while a higher transmission rate without associated errors. Thus, the optical systems division time through the generation of stable solitons are called to offer a higher bit rates with minimal distortion of the signal. It will emphasize the following subsection in such systems and a comparison is made with systems wavelength division with use of continuous wave lasers.

Today, companies like Huawei, Alcatel Lucent and Ericsson platforms offer a 40Gbps DWDM employing up to 160 channels over C and L band with separations between channels 50 Ghz. Most of these systems are prepared to support transmission rates of 100 Gbps. In February 2009, Verizon became the first service provider to employ 100 Gbps technology by Nortel technology on its existing DWDM network, a link from Paris to Frankfurt 893 km. This technology uses coherent detection and DP-QPSK (dual polarization quadrature phase shift keying) modulation.

1.2.2. The Soliton Advanced Optical Systems. [8]

Solitons are travelers intensive pulses without distortion of amplitude or shape along its displacement, unless the own attenuation medium. The first to study this type of pulses was J. Scott Russell in 1834 in the city of Edinburgh. This hydraulic engineer observed in the channel this city a wave in the water that moved without distortion aparante which helped him to perform a series of studies on the subject very useful for further advances. Historically these traveling waves have caused major damage to vessels and have already been sighted on numerous

occasions since ancient times, sources like Cristobal Colón himself testified of their existence, and of particular interest are those produced in the Amazonas River capable of traveling along thousands of kilometers unchanged personals causing serious losses both materials.

For optical communications such transmission systems were not useful until the appearance of lasers emissions train ultrashort pulses with regular and more specifically to the advent of lasers lock modes (mode-locking) amplification and switching gain (gain-switching). The first allow to obtain very short pulses with very good characteristics at the expense of a complex and inflexible repetition frequency in implementation. The latter generate picosecond pulses on the order of the repetition rate required a much simpler way. With this inrush could multiplexing techniques performed in the time domain (OTDM, Optical Time Division Multiplexing), in hybrid systems multiplexed in both time and wavelength (DWDM / OTDM) or division multiplexing code (OCDMA). [13]

In practice, the implementation of systems using optical solitons has two restrictions that make it unviable principle:

- A periodic amplification due to attenuation of the medium realize, which affects certain decremental effects. The first of these effects is the amplification of the pulse intensity, because the shape remains unchanged, carries a widening and the resulting bit error. Another amplification effect by the ASE (amplified spontaneous emission) is the specific noise soliton systems because of random changes of the center frequency of the pulses. This, in principle, should not be a problem because solitons can alter their frequency without affecting its form or energy, however changes in pulse frequencies can be converted into changes in arrival times called temporal jitter (Gordon-Haus jitter). To solve this problem can be incorparar certain

bandpass filters that change the center frequency of the soliton to the same filter, so are called filters guidance. The problem with this system is that the ASE noise accumulates in the passband of the filter chain inpracticable making long distance communication where you have to use a large number of them. A solution to this problem is to slightly modify the center frequency between filters so that the jitter does not accumulate in a fixed frequency. This technique called *sliding-frequency* enables transoceanic soliton transmission systems.

- A fiber with very low value of anomalous chromatic dispersion, $D < 0{,}2ps/nm-km$, other than those used by conventional WDM systems and also carries a higher rate of attenuation is used. This causes the increase of the number of repeaters and the distance between them, leading to an increase in system costs. To this must be added that the WDM systems using conventional solitons cause, due to XPM (cross-phase modulation), similar to those of Gordon-Haus time jitter, which makes it inadvisable to use to implement systems comuncaciones optical long distances. These three problems can be minimized with chirpeados utlización RZ pulses, also called solitons controlled dispersion soliton DC. First fibers can be used widely used controlled dispersion and marketing and small chromatic dispersion. Secondly the distance between amplifiers widens to the 60-80km compatible with the specific amplifiers WDM systems. Finally, XPM effects are reduced due to the high local chromatic dispersion and consequently the temporal jitter ready. Similarly and for the same reason, the Gordon-Haus jitter is also reduced, which makes it unnecessary to use guided slipped frequency filters. The NRZ systems, widely used in optical communications systems, generally operate in the linear regime, which leads that are not sensitive to

excessive chromatic dispersion, plus they have PMD penalties higher than the RZ systems. In contrast, systems with DC or RZ soliton I have a minor problem chirpeado PMD, so that they are superior in all sections of the fiber achieving higher transmission speeds to 40Gb / s much longer distances.

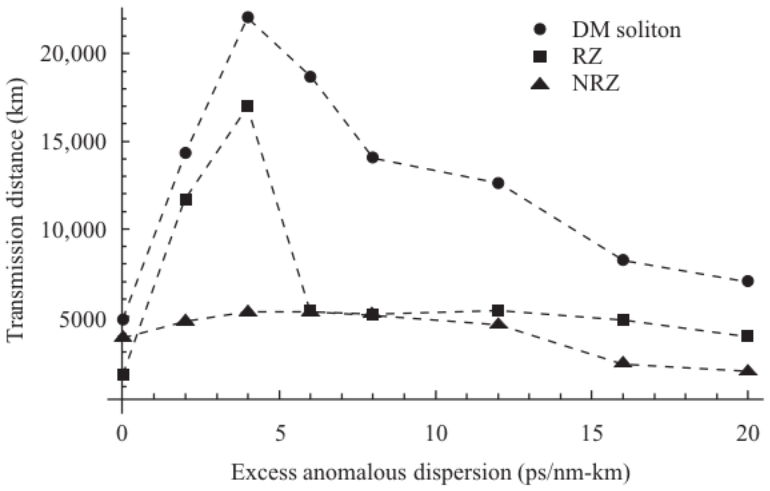


Figura 1.12: Comparison between NRZ, RZ and RZ chipped systems (soliton DC) [8]

Today the system more implanted optical communications is division multiplexing wavelength (WDM) while not generally used ténicas emission temporal solitons, can I be improved transmission capacity over long distances due to above characteristics. Because the optical pulses travel at different group velocity, collisions may occur that adversely affect the dispersion thereof with the corresponding bit error. By working with solitons, even if collisions occur, the shape and intensity of the optical pulses are not affected, being much more immune to these problems. This technique media access is based on the division into channels at wavelengths different but very close, around a central commonly found in third window, 1550nm. For m channels with equal bit rate B, the system capacity will

be *mxB*. WDM has the potential to exploit the large bandwidth offered by fiber optics both second window and third window, as shown in the following graph:

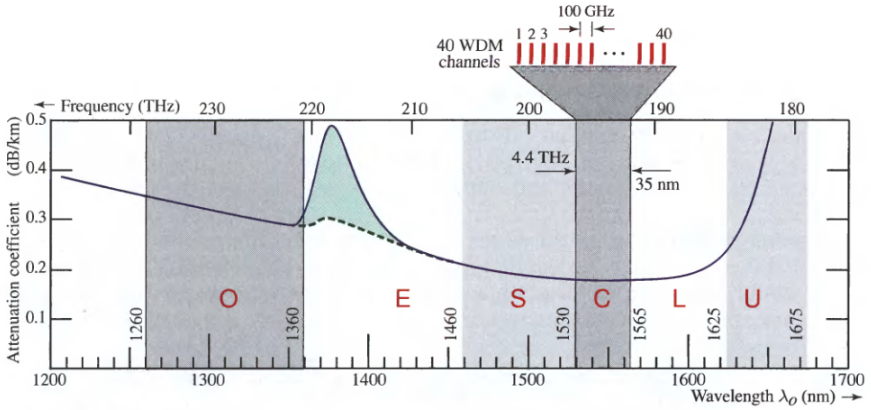


Figura 1.13: Optical spectrum channels forty third window, band C, with a spacing of 100GHz [7]

The C band is used for transmission CWDM, coarse WDM according to ITU-T G.694.2 the standard, with a bandwidth of 4.4THz, while the L band is used for DWDM, dense WDM ITU-T G. according 694.1, system much more progress in the last decade, with a bandwidth of 10.6THz hundred different channels corresponding to the same bit rate. In total, the third window that typically operate WDM systems having a bandwidth of 15THz, while in the second window around 1330nm, corresponding to the band O, original zero dispersion, the width is of 14THz. [14]

The system, however, best suited to the use of solitons is OTDM, Optical Time Division Mulitplexing wherein the optical time to multiplex and demultiplex the information is used. Data flows are mulitplexados slow stream to a higher speed at the entrance of the channel and are extracted groups slower fast flow pulses through a demultiplexer function. Functionally it is identical to ETDM except that the operations are performed entirely in the optical domain at high

speeds. The widths of the pulses should be such that bit rates of several tens of hundreds of Gbps be achieved. The laser which can generate pulses of order better PS is the mode-locked laser or by using continuous wave lasers with an external modulator. But, considering that the temporary pulses are very short, the frequency spectrum must be very large, which will influence the effects of chromatic dispersion. To solve this, they are commonly used pulse return to zero (RZ). In a system bit interleaved multiplexing of N channels it is achieved by a technique of delay can be implemented optically simplified combination of channels using an optical coupler Nx1. Each channel occupies a time frame $T_B = 1/(N_B)$ and requires a pulse duration of less than T_B carrier. A slotted OTDM system transmits data packets from each channel. Note that normally $\tau_p < \tau$, the first pulse width and second width between pulses, so there is a certain guard time between successive pulses [11]. One purpose of this guard time is to provide some tolerance in the operations of multiplexing and demultiplexing, another reason is to prevent undesirable interactions between adjacent pulses.

In a system OTDM bit interleaved optical multiplexer generates a train of periodic pulses which are then divided by a splitter in many replicas of the same soliton, as many as multiplexed channels n is. This sequence is delayed according to the number of caesarean channel a certain time, so that depending on the assigned channel will be placed in a specific position within the frame. Once positioned into place, data entry modulates the soliton train obtaining a sequence which is combined with another generated in the same way and with the synchronization pulses in order to facilitate demultiplexing. Where there is a one in the input sequence, a soliton is placed in its assigned position, while in those places where there is zero space will air the same white temporal size. For demultiplexing the input stream into two equal replicas is divided, the first is delayed by a multiple of the time corresponding to the channel number to which it belongs

pulse. It goes through an AND gate pulse to each frame of the soliton delayed occupying the same position in the train without delay and shall correspond to the channel you want demultiplexing in each frame rail.

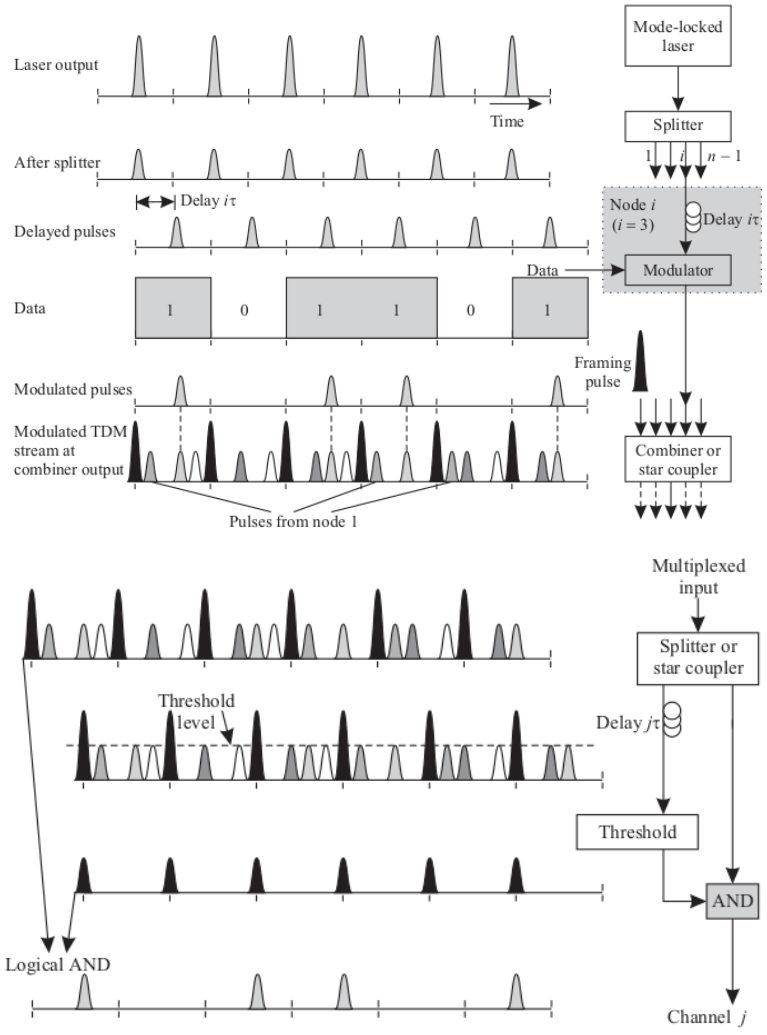


Figura 1.14: OTDM Optical Multiplexing and Demultiplexing transmission. [8]

Capítulo 2

Material and method

PyOFTK [15], it is a library made in Python and Numpy to help design systems numerical simulation of optical pulses in optical fiber. They are a set of functions and libraries open source and free license, which make up a job unfinished by Martin Laprise for the development of their graduation project. This work is based on the SSPROP program at the University of Maryland where it is developed in Matlab language, the split-step Fourier method symmetrized for solution of the equations of Schrödinger both scalar and coupled [9]. To a large extent, the library in question is a literal translation of the language Python open source software.

PyOFTK is structured in a series of Python files necessary to perform different simulations of propagation both fiber optic and free space. For this matter, as discussed above, a number of libraries for numerical calculation and management of arrays, as well as parallel computing.

2.1. Libraries necessary for using PyOFTK

To use the package PyOFTK it is necessary to install a series of numerical computation libraries for Python, performing Fourier transforms, and displays graphics and simulations. Similarly, the installation of APIs and development CUDA OpenCL python is required. Below is a brief description of each is made to understand more easily use program processes:

2.1.1. SciPy

SciPy is an open source library of tools and mathematical algorithms to Python born from the original collection of Travis Oliphant extension modules for Python, launched in 1999 under the name Multipack.

SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal processing and image resolution of ordinary differential equations and other tasks for science and engineering.

Its latest version is 0.16.1 SciPy of 24.10.2015.

2.1.2. NumPy

Package for scientific computing with Python object with a powerful N-dimensional array of sophisticated functions (broadcasting), tools for integrating C/C++ and Fortran, useful linear algebra, Fourier transform and random number capabilities.

Besides the obvious scientific uses, Numpy also be used as an efficient multi-dimensional generic data container. This allows seamlessly integrate NumPy with a wide variety of databases. Numpy is licensed under the BSD license, which allows reuse with few restrictions.

Its latest version is 1.10.1 NumPy

2.1.3. Matplotlib

It is a Python library for rendering 2D figures and high quality graphics. It can be used on multiple platforms such as python scripts, Python, Matlab and Mathematica ipython, web servers and several toolkits GUI.

You can generate areas, histograms, power spectra, bar graphs, error, scatter diagrams, etc.

Stable release: 11/03/2015 2.0 Matplotlib

2.1.4. Python-VTK

VTK (Visualization Toolkit) is a toolkit for visualization of 3D computer graphics system consisting of a free object-oriented software for image processing.

It supports a wide variety of visualization algorithms such as scalar, vector, tensor, textures and volumetric methods.

It supports numerical modeling techniques such as implicit, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation. Python-VTK provides shared libraries that allow use VTK from Python scripts.

The latest version is 5.10.1 Python-VTK 28/09/2015

2.1.5. PyFITS

This module is a Python library that provides access to files FITS (Flexible Image Transpor System) commonly used in astronomy to store images and tables.

2.1.6. Python-tables

Library that represents the relational database as a table system that allows multidimensional arrays make use of a large number of high-speed data access.

This memory is continuously optimized and the data can be used in real time for performing simulations and numerical approximation methods. It is built on top of the HDF5 library using language Python and Numpy package.

2.1.7. SWIG

Simplified Wrapper and Interface Generator is an open source tool used to connect programs and libraries written in C or C ++ with encryption languages like Lua, Perl, PHP, Python, R, Ruby, etc. In this way the program can be modified much more quickly through a scripting language rather than in C.

Output can also be in XML format or Lisp S-expressions.

It was created by Dave Beazly in 1996 although it is currently supported by an active group of volunteers led by William Futon. It has been done under the GNU license.

2.1.8. FFTW2 [16]

Library for computing Fast Fourier Transforms in C and C ++ in one or more dimensions with real and complex data and an input of arbitrary size. Also it includes processed in parallel for both shared memory and distributed memory. This software typically performed transformed faster than other programs available to developers. An important feature is that the FFTW not a fixed algorithm used for calculation of the transform, since the DFT adapts to the hardware details for better yields. The information is contained in a data structure called PLAN produced by the scheduler. Take the FFTW computes the transformed according to the obtained corresponding algorithms PLAN which, in turn, has been passed to the executor along with a set of data. The pattern planficador / executor applies to the four operating modes FFTW:

1. FFTW. Transforms a complex dimension.
2. FFTWND. Transform two-dimensional complex.
3. RFFTW. Transformed dimensional real data.
4. RFFTWND. Transformed multidimensional real data.

Each comes with its own planner and executor.

Also can be programmed to perform FFTW transformed irregular where one type of fast algorithm to prioritize certain arrays whose size can be factored in certain small primes, and using routine slower general purpose other factors.

2.1.9. FFTW3 [17]

The new version of FFTW is not compatible with earlier versions FFTW2 or because they have different semantics. Therefore no compatibility between the two wraps can be performed efficiently. Since data formats themselves are identical and style Planner / Executor is essentially the same, updating the other version itself is simple. In FFTW3, unlike previous versions, no separate header files for real and complex transformations. The biggest difference is in the division of planning / execution of work. In particular FFTW3 arrays, jump parameters, etc. to create the plan that is executed after specifically for these arrays and these parameters are specified. Before much of the information is specified at runtime and is now specified in planning time. On the other hand, in this version all types of transforms have the same type of fftw_plan plan and all are destroyed by fftw _destroy_plan..

2.1.10. reikna [18]

It is a library with several algorithms GPU to operate PyCUDA and PyOpenCL. Allows separation of cores to perform calculations such as matrix multipli-

cation, random number generation, etc., simple transformations on their values access input and output such as scaling, box, etc. It also allows the separation of the stages of preparation and execution so that maximizes the execution priority on the preparation in large simulations. Finally makes a partial abstraction from CUDA /OpenCL.

Latest version: reikna0.6.6 of 11.05.2015.

2.1.11. PyCUDA [19]

Since the base CUDA programming is written in C/C++, it is necessary an API wrapper or wrappers for CUDA in order to adapt it to Python. Already exist prior to the implementation of PyCUDA several of these APIs, however, this library is essential due to abstraction in the use of kernel or the use of arrays in GPU, resulting in greater convenience for developers. Moreover, error checking is automatic, all CUDA errors translate into Python exceptions. The language used by PyCUDA, often called RAII in C ++, makes it much easier to write code correction so as to achieve link the cleaning of the object to its lifetime, that is, until not end the context in which it works a particular object will not be released all its memory. Latest version: 2014.1 PyCUDA.

2.2. Split-Step Fourier method for resolution symmetrized scalar Schrödinger equation

The differential equation describing the propagation in a nonlinear optical fiber and guides planar light wave is called nonlinear differential equation Schrödinger (NLSE). In theoretical physics, this equation is a nonlinear variation of the Schrödinger equation. In addition, this equation appears in the study of small wave amplitudes in the nonviscous water surface; beam propagation of plane waves in localized regions of the ionosphere; soliton propagation alpha-helix of Davydov, responsible for transporting energy along molecular channels; and many others. Development to reach the Schrödinger equation and a summary of the theory of propagation of optical pulses in dispersive and nonlinear optical fibers, they can be found in the appendix A.

$$\frac{\partial A}{\partial z} = j\beta_1 \frac{\partial A}{\partial t} - \frac{j\beta_2}{2}\frac{\partial^2 A}{\partial t^2} - \frac{\beta_3}{6}\frac{\partial^3 A}{\partial t^3} j - \gamma |A|^2 A$$

This equation has no analytical solution if assessed globally, but you can reach a numerical approximation if taken separately the two terms of the second term of equality. The first one is considered the nonlinear and dispersive part of the equation, while the second would be its linear part. This system is the one that follows the method `Split-Step Fourier` for solving this equation. In general, the method consists of calculating the dispersion parameter on one side and the nonlinearity of the other, so that calculations are rotated along the entire length of the fiber. Thus, the dispersive part is calculated in the frequency domain corresponding parts eliminating nonlinearities and nonlinear part in the time domain if the dispersive terms in the equation are eliminated. Multiplying the input pulse first by the dispersive factor found after performing the Fourier transform and the

result is again multiplied by the nonlinear factor once the inverse Fourier transform, alternating products and transformations along the fiber , you can reach a workable solution to a given distance, as can be seen in the following graph:

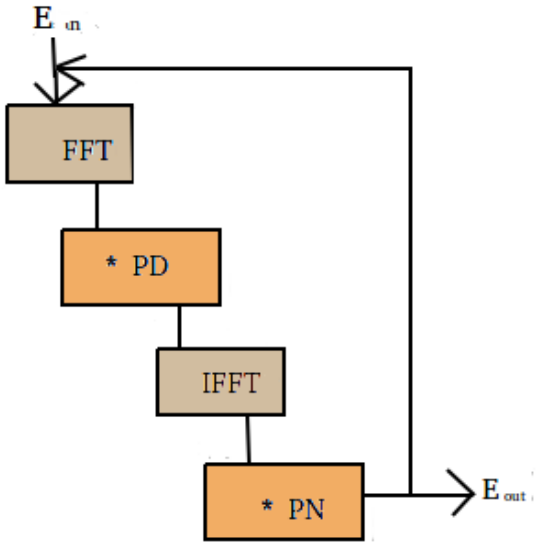


Figura 2.1: Scheme Split-Step Fourier.

- Linear Part or dispersive: $\frac{\partial A_D}{\partial z} = -\frac{j\beta_2}{2}\frac{\partial^2 A_D}{\partial t^2}$

This part is solved in the frequency domain, so that after performing the Fourier transform, we will be as follows:

$$\frac{\partial \widetilde{A}_D}{\partial z} = \frac{j\beta_2}{2}(\omega - \omega_0)^2 \widetilde{A}_D$$

Then this differential equation is solved by passing the denominator of the first term to the second and A_D from second to first, and integrating both with respect to z. We calculate interest for a certain h small very close to z:

$$\int \frac{d\widetilde{A}_D(\omega, z+h)}{\widetilde{A}_D(\omega, z+h)} = \int \frac{j\beta_2}{2}(\omega - \omega_0)^2 dz$$

$$L_n\widetilde{A}_D(\omega, z+h) = \frac{j\beta_2}{2}(\omega-\omega_0)^2(z+h)$$

$$\widetilde{A}_D(\omega, z+h) = e^{\frac{j\beta_2}{2}(\omega-\omega_0)^2(z+h)}$$

$$\widetilde{A}_D(\omega, z+h) = \widetilde{A}_D(\omega, z) * e^{\frac{j\beta_2}{2}(\omega-\omega_0)^2 h}$$

We must solve the nonlinear part in the time domain and pass it to the frequency domain to then multiplied by the solution of the dispersive part:
- No Lineal Part: $\frac{\partial A_N}{\partial z} = j\gamma\left|A\right|^2 A_N$

$$\int \frac{\partial A_N(t, z+h)}{A_N(t, z+h)} = \int j\gamma\left|A\right|^2 dz$$

$$L_n A_N(t, z+h) = j\gamma\left|A\right|^2 (z+h)$$

$$A_N(t, z+h) = e^{j\gamma|A|^2(z+h)}$$

$$A_N(t, z+h) = A_D(t, z+h)e^{j\gamma|A_D(t,z+h)|^2 h}$$

$$\widetilde{A}_N(\omega, z+h) = TF(A_N(t, z+h)) = \int_{-\infty}^{\infty} A_N(t, z+h)e^{-j(\omega-\omega_0)t}dt$$

If both terms are multiplied in the frequency domain the value of the offset distance h another signal would be obtained:

$$\widetilde{A}_D(\omega, z+2h) = e^{j\frac{\beta}{2}(\omega-\omega_0)^2 h} * \widetilde{A}_N(\omega, z+h)$$

It has to pass this result into the time domain again, through the inverse Fourier transform to return to displace another some distance h multiply provided by the nonlinear operator.

$$A_D(t, z+2h) = TFI(\widetilde{A}_D(\omega, z+2h)) = \int_{-\infty}^{\infty} \widetilde{A}_D(w, z+h)e^{j(\omega-\omega_0)t}d\omega$$

$$A_N(t, z + 2h) = A_D(t, z + 2h)e^{j\gamma|A_D(z+2h)|^2 h}$$

Thus the signal is moving along the z axis a certain amount until modified again the modulus value of A with said jump. two sizes have thus jump, the first to advance into the second to reach the next space segment, where it will perform the same process. many iterations will be made as to advance along the space depending on the size of h. Said size h influence the relative error of the simulation, the smaller their size, the lower the accumulated error.

When steps $\frac{h}{2}$ are performed instead of h performing two steps in each iteration, one to apply the linear operator and another to apply the nonlinear, you can give a solution as follows:

$$A(t, z + h) = e^{(j\frac{\beta}{2}(\omega-\omega_0)^2\frac{h}{2})} \;\; e^{(\int_z^{z+h} j\gamma|A|^2 dz)} \;\; e^{(j\frac{\beta}{2}(\omega-\omega_0)^2\frac{h}{2})} \; A(z, t)$$

This form is called *method split-step Fourier symmetrized* because nonlinear effects are in the center of the embodiment with dispersive operators applied before and after symmetrically. The biggest advantage in using this method with respect to the first is that the error is of order three in one jump size h. If the trapezoidal rule is used to approximate the integral, one can achieve greater precision in the nonlinear contribution method:

$$\int_z^{z+h} j\gamma \left|A\right|^2 dz \approx \frac{h}{2} j\gamma[|A(z)|^2 + |A(z + h)|^2]$$

The main problem of using this methodology is not known, in principle, the value of the nonlinear field in the middle of step h, ie $z+\frac{h}{2}$. It is necessary to perform an iterative process starting by replacing $|A(z + h)|$ by $|A(z)|$. Applying this value to the above equation for $A(t, z + h)$ with which to calculate the new nonlinear operator, always dependent on the field strength.

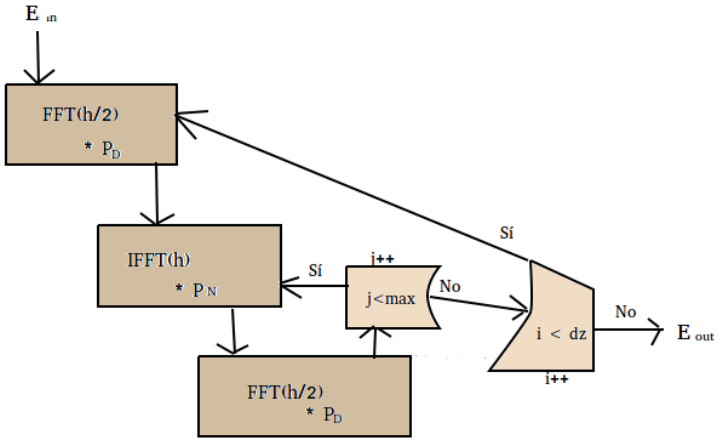


Figura 2.2: Algorithm for the implementation of the Split-Step Fourier method symmetrized.

2.3. Split-Step Fourier method for solving symmetrized Vector coupled Schrödinger equations

[9]

The vector solution of the Schrödinger equations coupled, where the propagation of an optical pulse in its two polarization components, horizontal and vertical described, is developed in Appendix B of this work following the description raised in reference book [1]. Therein two types of NLSC depending on whether the polarizations components are circular or elliptical studied. The first is more convenient if we consider an optical fiber with no lineal length of the same order that beat length or somewhat smaller; and, conversely, if the length of nonlinearity is much higher than the beat length, the second, in which case they may not disregard the additional terms of the equations is used.

Circular polarization:

$$\frac{\partial A_+}{\partial z} + \beta_1 \frac{\partial A_+}{\partial t} + j\frac{\beta_2}{2}\frac{\partial^2 A_+}{\partial t^2} + \frac{\alpha}{2}A_+ = \frac{j}{2}(\Delta\beta)A_- + j\frac{2\gamma}{3}\left(|A_+|^2 + 2|A_-|^2\right)A_+$$

$$\frac{\partial A_-}{\partial z} + \beta_1 \frac{\partial A_-}{\partial t} + j\frac{\beta_2}{2}\frac{\partial^2 A_-}{\partial t^2} + \frac{\alpha}{2}A_- = \frac{j}{2}(\Delta\beta)A_+ + j\frac{2\gamma}{3}\left(|A_-|^2 + 2|A_+|^2\right)A_-$$

Elliptical polarization:

$$\frac{\partial A_x}{\partial z} + \beta_{1x} \frac{\partial A_x}{\partial t} + j\frac{\beta_2}{2}\frac{\partial^2 A_x}{\partial t^2} + \frac{\alpha}{2}A_x = j\gamma(|A_x|^2 + B|A_y|^2)A_x$$

$$\frac{\partial A_y}{\partial z} + \beta_{1y} \frac{\partial A_y}{\partial t} + j\frac{\beta_2}{2}\frac{\partial^2 A_y}{\partial t^2} + \frac{\alpha}{2}A_y = j\gamma(|A_y|^2 + B|A_x|^2)A_y$$

$$B = \frac{2 + 2sin^2\Gamma}{2 + cos^2\Gamma}$$

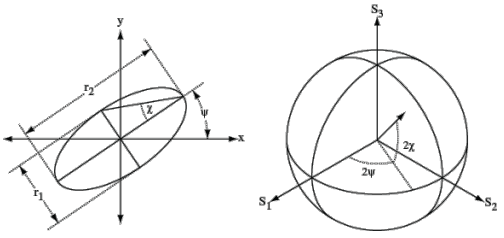


Figura 2.3: Polarization diagram elliptical ($\chi = \Gamma$). [9]

Following this criterion we have been implemented two methods of split-step Fourier symmetrized as is done in the SSPROP program at the University of Maryland [9], where the same steps as for the method climb with continue respective transformations related to the polarization of the light beam in the birefringent fibers. Each of the processes must be performed twice, once for the equation vertically and one for the landscape mode.

While, in the SSPROP program choosing angles ellipticity and tilt are carried out from a vector `psp` describing the polarization state of the fiber in implemented

in this work, have including separately each. Moreover, it has been considered the attenuation constant α depending on the frequency. Following the approach used in the SSPROP program, two different methods are performed

- If we find `circular method`, first it has to perform the transformation explained in the Annex to describe the vertical and horizontal components based on the two input components in the x-axis and y. In this way, the pulse is placed in the appropriate reference system within the birefringent fiber:

$$u0a = (u\_ini\_x + 1{,}0j * u\_ini\_y)/\sqrt{2}$$
$$uba = (1{,}0j * u\_ini\_x + u\_ini\_y)/\sqrt{2}$$

Dispersive part of the method must be made taking into account the additional component with $\frac{j}{2}(\Delta\beta)A_+(-)$, where $\Delta\beta z$ is equivalent to the angle Γ. This factor will condition the result greatly complicating the calculation of each of the resulting components. Thus we can consider the system of equations of the dispersive part as follows:

$$\frac{\partial A_+}{\partial z} = -j\beta_1\omega_0 A_+ - j\frac{\beta_2}{2}\omega_0^2 A_+ - j\alpha_1\omega_0 A_+ + j\frac{\alpha_2}{2}\omega_0^2 A_+ + \frac{j}{2}(\Delta\beta)A_-$$
$$\frac{\partial A_-}{\partial z} = -j\beta_1\omega_0 A_- - j\frac{\beta_2}{2}\omega_0^2 A_- - j\alpha_1\omega_0 A_- + j\frac{\alpha_2}{2}\omega_0^2 A_- + \frac{j}{2}(\Delta\beta)A_+$$

If this system solve coupled differential equations, we found that the coupling component prevents obtain dispersive parameters equivalent to those obtained in the method scalar:

$$\frac{\partial A_+}{A_+} = \left[-j\beta_1\omega_0 - j\frac{\beta_2}{2}\omega_0^2 - j\alpha_1\omega_0 + j\frac{\alpha_2}{2}\omega_0^2 + \frac{j}{2}(\Delta\beta)\frac{A_-}{A_+}\right]\partial z$$

$$\frac{\partial A_-}{A_-} = \left[-j\beta_1\omega_0 - j\frac{\beta_2}{2}\omega_0^2 - j\alpha_1\omega_0 + j\frac{\alpha_2}{2}\omega_0^2 + \frac{j}{2}(\Delta\beta)\frac{A_+}{A_-}\right]\partial z$$

We solve the integral for both equations and obtain the following results for later calculate dz/2 later.

$$A_+ = exp\left[-j\beta_1\omega_0 z - j\frac{\beta_2}{2}\omega_0^2 z - j\alpha_1\omega_0 z + j\frac{\alpha_2}{2}\omega_0^2 z\right] e^{\frac{j}{2}(\Delta\beta)\int \frac{A_-}{A_+}\partial z}$$

$$A_- = exp\left[-j\beta_1\omega_0 z - j\frac{\beta_2}{2}\omega_0^2 z - j\alpha_1\omega_0 z + j\frac{\alpha_2}{2}\omega_0^2 z\right] e^{\frac{j}{2}(\Delta\beta)\int \frac{A_+}{A_-}\partial z}$$

What a linear transformation matrix follows for a variation dz/2 distance after the initial length in each section z, according. [9].

$$\begin{pmatrix} A_+(\omega, z + dz/2) \\ A_-(\omega, z + dz/2) \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix} \begin{pmatrix} A_+(\omega, z) \\ A_-(\omega, z) \end{pmatrix}$$

The h parameters resulting change in the matrix are as follows:

$$h_{11} = \frac{1}{2}\left[(1 + sin(2\chi))h_a + (1 - sin(2\chi))h_b\right]$$

$$h_{12} = -\frac{j}{2}e^{j2\psi}cos(2\chi)(h_a - h_b)$$

$$h_{21} = \frac{j}{2}e^{-j2\psi}cos(2\chi)(h_a - h_b)$$

$$h_{22} = \frac{1}{2}\left[(1 - sin(2\chi))h_a + (1 + sin(2\chi))h_b\right]$$

For the nonlinear part we consider null dispersion coefficients and solve the resulting system of equations:

$$\frac{\partial A_+}{\partial z} = \frac{2j\gamma}{3}\left(|A_+|^2 + 2|A_-|^2\right)A_+$$

$$\frac{\partial A_-}{\partial z} = \frac{2j\gamma}{3}\left(|A_-|^2 + 2|A_+|^2\right)A_-$$

Since the part belonging to the other polarization component is the intensity does not affect the result of the equation so that the resolution is simplified enough if the method is used trapeze:

$$A_+(t, z+dz)_N = exp\left[\int_z^{z+dz} \frac{2j\gamma}{3}(|A_+|^2 + 2|A_-|^2)dz\right]$$

$$A_-(t, z+dz)_N = exp\left[\int_z^{z+dz} \frac{2j\gamma}{3}(|A_-|^2 + 2|A_+|^2)dz\right]$$

For a distance of $d\_z/2$ after the initial z, we have the following parameters nonlinearity:

$$A_+(z+dz)_N \approx exp\left[\frac{2j\gamma}{3}(|A_+|^2 + |A_+(+dz/2)|^2 + 2|A_-|^2 + 2|A_-(+dz/2)|^2)dz/2\right] A_+$$

$$A_-(z+dz)_N \approx exp\left[\frac{2j\gamma}{3}(|A_-|^2 + |A_-(+dz/2)|^2 + 2|A_+|^2 + 2|A_+(+dz/2)|^2)dz/2\right] A_-$$

Therefore, to obtain an expression of the complete method Split-Step Fourier symmetrized, as in the scalar case, half dispersive jump, followed by a non-linear jump and finally other means dispersive jump on each complete step is performed:

$$A_+(z+dz) = \left[h_{11}\hat{A}_+ + h_{12}\hat{A}_-\right] exp\left[\int_z^{z+dz} \frac{2j\gamma}{3}(|A_+|^2 + 2|A_-|^2)dz\right] \left[h_{11}\hat{A}_+ + h_{12}\hat{A}_-\right]$$

$$A_-(z+dz) = \left[h_{21}\hat{A}_+ + h_{22}\hat{A}_-\right] exp\left[\int_z^{z+dz} \frac{2j\gamma}{3}(|A_-|^2 + 2|A_+|^2)dz\right] \left[h_{21}\hat{A}_+ + h_{22}\hat{A}_-\right]$$

Once all necessary iterations, the optical output fields are obtained by means of the Jones matrix transposed to the input:

$$\begin{pmatrix} u\_out\_x \\ u\_out\_y \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -j \\ -j & 1 \end{pmatrix} \begin{pmatrix} A_+(L) \\ A_-(L) \end{pmatrix}$$

- For *elliptical polarization method* the procedure is the same but the scattering parameters and nonlinearity, and the transformation to the symmetry axis of the fiber birefringent vary considerably. First, as described in Appendix B, the transformation matrix multiplies the input values for the signals to change within the fiber:

$u0a = [cos(\psi)cos(\chi) - jsin(\psi)sin(\chi)]\, u\_ini\_x + [sin(\psi)cos(\chi) + jcos(\psi)sin(\chi)]\, u\_ini\_y$

$u0b = [-sin(\psi)cos(\chi) - jsin(\psi)sin(\chi)]\, u\_ini\_x + [sin(\psi)cos(\chi) + jcos(\psi)sin(\chi)]\, u\_ini\_y$

Since no dispersive coupling term, the nonlinearity parameter for each of the equations fits obtained for scalar equation:

$$A_a(\omega, z + dz/2)_D = h_a * A_a(\omega, z)$$
$$A_b(\omega, z + dz/2)_D = h_b * A_b(\omega, z)$$

Where h_a and h_b are given by:

$$h_a = exp\left[-\frac{\alpha_a(\omega)}{2}z + j\beta_a(\omega)z\right] \qquad h_b = exp\left[-\frac{\alpha_b(\omega)}{2}z + j\beta_b(\omega)z\right]$$

However, for the parameter of nonlinearity must consider ellipticity factor B:

$$B = \frac{2+2sin^2\chi}{2+cos^2\chi}.$$

Although the method of making the same, that is, by solving the integral trapezoid method, expressions are much more complex:

$$A_{a(+dz)} = exp\left[\int_z^{z+dz} \frac{2j\gamma}{3}(|A_a|^2 + B|A_b|^2)dz\right]$$

$$A_b(+dz) = exp\left[\int_z^{z+dz} \frac{2j\gamma}{3}(|A_b|^2 + B|A_a|^2)dz\right]$$

$A_{a(+dz)} = exp\left[\frac{-j\gamma}{3}\left((2 + cos^2(2\chi))(|A_a|^2 + |A_{a(+dz)}|^2) + (2 + 2sin^2(2\chi))(|A_b|^2 + |A_{b(+dz)}|^2)\right) dz/2\right] A_a$

$A_{b(+dz)} = exp\left[\frac{-j\gamma}{3}\left((2 + cos^2(2\chi))(|A_b|^2 + |A_{b(+dz)}|^2) + (2 + 2sin^2(2\chi))(|A_a|^2 + |A_{a(+dz)}|^2)\right) dz/2\right] A_b$

The double angle sines and cosines is obtained by trigonometric pursuant to cancel the denominator of the B and not be developed in this section. Moreover, the square cosine of the double angle of ellipticity becomes one for circular polarization, while the breast is canceled.

Once the iterations that are considered appropriate to advance the fiber a certain distance, the inverse transformation is performed to display the signals of the fiber at the other end:

$$\begin{pmatrix} u\_out\_x \\ u\_out\_y \end{pmatrix} = \begin{pmatrix} cos(\psi)cos(\chi) + jsin(\psi)sin(\chi) & -sin(\psi)cos(\chi) - jcos(\psi)sin(\chi) \\ sin(\psi)cos(\chi) - jcos(\psi)sin(\chi) & cos(\psi)cos(\chi) - jsin(\psi)sin(\chi) \end{pmatrix} \begin{pmatrix} A_a(L) \\ A_b(L) \end{pmatrix}$$

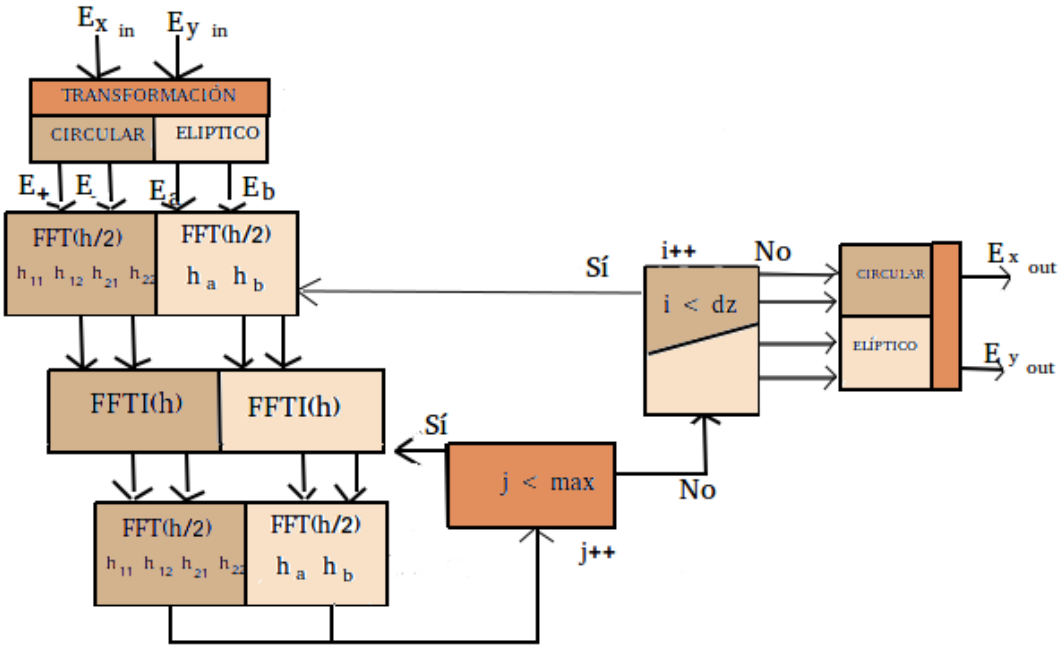


Figura 2.4: Diagram Split-Step Fourier symmetrized, ec. coupled.

Capítulo 3

Results and discussion

3.1. Functions and examples for using scalar Schrödinger equation.

3.1.1. Implementation of a function that solves the differential nonlinear Schrödinger equation using the split-step Fourier method symmetrized

As indicated in the introduction to this work, the programming language python with all libraries and APIs that support and supplement for mathematical resolution and graphic representation of functions in the implementation of a function that solves *the equation Schrödinger* be used that describes the propagation of a short along a non-dispersive optical fiber line and pulse. This function is already performed on the *ssf.py* PyOFTK belonging to the library, which will serve as a source for the realization of our practices file.

ssf.py contains various functions for execution in the CPU and many others to be executed in this language gpu using CUDA. The difference between the two

is subtle, lies in the incorporation or not the calculation of the phase to be used in certain examples, performing a loop *for* only for the calculation of *split-step Fourier* instead of two loops *for* nested simulating space jumps and small steps in the simulation between skip and jump, or the inclusion of the field strength calculated as the square modulus of the input variable. While the original file *ssf.py* repeated for each function all the code redundantly in the generated for this work has decided to simplify as far as possible without modifying the method call or the parameters introduced in order to reduce the code lines.

To perform a more thorough explanation of the method *split-step Fourier* and modifications made to the original file, we will divide this section into two sections: the first, we will address the function *SSF()* to be executed solely on the CPU, ie sequentially without utlización CUDA; in the second, the *ssfgpu()* function for execution on both CPU, the sequential part, and GPU, the part will be parallelized.

Functions implemented for execution on the CPU

Given that the function *ssfu()* performs a quick version of the method *split-step Fourier* with only a hop size, for making use of a simple *for*, and the other two functions, *SSF()* and *ssf _b()*, utlizan jump two sizes depending on the method used; it was decided to divide into two parts the general function that describes it. The first one, representing the top of the code, it will be common for all three functions and will be named *ssfUP()*; the second part will contain the iterative method section and will be different for *ssfu()* with a single loop *for*, function called *ssfDOWN2()*, that for the other two, *ssf()* and *ssf _b()*, with the two loops *for*, which will be called *ssfDOWN1()*.

To choose one or the other, depending on the circumstances, it was decided to include a new selective function with two *if-else* nested with which that of the

above described will be taken depending on the input parameter *phiNLOut* if you want to return the phase with the solution method or not. In the latter case, the second *if-else* will be held to select the second part of the code that is preferred, which has a single jump or whom he has two for which a new Boolean parameter called be used *down1*:

```
def ssf(u0, dt, dz, nz, alpha, betap, gamma, maxiter, to5, phiNLOut, down1):
   if phiNLOut:
      return ssf1(u0, dt, dz, nz, alpha, betap, gamma, maxiter, tol, phiNLOut, down1)
   else:
      if down1:
         return ssf_b(u0, dt, dz, nz, alpha, betap, gamma, maxiter, tol, phiNLOut, down1)
      else:
         return ssfu(u0, dt, dz, nz, alpha, betap, gamma, maxiter, tol, phiNLOut, down1)
```

Input parameters common to all three functions are:

u0, as input function consists of a vector of samples representing a pulse determinadoson.

dt, temporal variation of the samples for calculating the working frequency.

dz, It represents the spatial variation of short pitch in the z direction.

nz, is the number of steps in that long divided the sample and define the number of the first itariciones *for*.

alpha, It is the constant loss and will only have value if it is decided to include losses in the fiber, which would alter the Schrödinger equation by adding a new term, as we saw in the previous section. Normally you have value 0.0.

betap, It is the propagation constant and is represented by a vector of three elements, but may include all who wish. Only one of them will be different from 0.0, to indicate the type of dispersion is being used. As explained in the previous section, the first order is usually despised by not cause significant effects on the pulse, the second order is called VGD, velocity group dispersion, and the third order, TOD, dispersion third order which it should only be considered when the wavelength is located very close, on the order of nanometer wavelength of zero

dispersion, to 1,27μm.

gamma, It represents nonlinearities and is used to calculate the nonlinear part of the equation.

maxiter, is the number of short steps we divide each stride and define the number of iterations of the second *for*.

tol,indicate the tolerance that want to give the relative error between the value of the data in each slow pace of its value in each stride. It is a system so that the data should not diverge too much from the original value and error each jump is not spreading. Indicársele usually an error on the order of ten microns.

phiNLOut, is a boolean indicating whether you want to return or not the calculated phase.

down1, as mentioned above, is another Boolean to indicate whether you want to use the *ssfDOWN1()* function or the *ssfDOWN2()*.

The first function of the *ssfUP()* method comprises, creates a variable *nt* which assigns the length of the input signal; then uses this variable and the size between short steps *dt* to construct a vector *w* of equally spaced angular frequencies in the spectrum. To calculate this frequency uses the *wspace* function included in the of PyOFTK utilities.py file.

The first value is assigned to step half represented by the variable *Halfstep* (said half step, because the method that implements the function is the *split-step Fourier symmetrized* that leaps half of length to general method) it is half of the constant linear losses, in most cases 0.0 to treated optical fiber high speed and almost no losses.

It is important to mention that this method begins working in the frequency domain and therefore makes development in Taylor series of the propagation constant used in the construction of the dispersive part of the differential equa-

tion, replacing the parameters $\frac{\partial}{\partial t}$ for the value of the vector *w* of equally spaced frequencies previously calculated. For the development of Taylor uses a loop *for* that runs all terms of the vector *BETAP* introduced in the function parameters and fills the variable *Halfstep* with every new term.

The method, as explained in the previous section, conducts an independent solution of the linear part of the differential Schrödinger equation that returns the exponential of the dispersive component. The solution displaced a half step size, dz/2, will be equal to the solution in the origin multiplied by the displacement solution, which will be the multiplier factor in each of the linear method jumps:

$$\hat{A}_D(\omega, z + dz/2) = \hat{A}(\omega, z)e^{-j\frac{\beta_2}{2}dz/2}$$

The variable *Halfstep* finally represent this multiplier factor step. To apply to the input signal must be passed to the frequency domain using a fast Fourier transform, using the package *fftpack* library *scipy*. This is included in a variable that *ufft* with *Halfstep* will be returned for use in the next part of the method. The complete code for the function that implements this first part is as follows:

```
def ssfUP(u0, dt, dz, nz, alpha, betap, gamma, maxiter, tol = 1e-5):
   nt = len(u0)
   w = wspace(dt*nt,nt)
   halfstep = -alpha/2.0
   if len(betap) != nt:
      for ii in arange(len(betap)):
         if (ii % 2 == 0): m = -1
         else: m = 1
         halfstep = halfstep - 1.0j*m*betap[ii]*pow(w,ii)/factorial(ii)
   halfstep = exp(halfstep*dz/2.0)
   ufft = fftpack.fft(u0)
   return [ufft, halfstep]
```

The second part of the comprehensive approach is the one responsible for performing the iterations progress along the z axis, with a number *nz* of jumps and

a number of steps hopping resizable marked by the input parameter `maxiter`. Each jump will have a full length `dz` divided into several steps minimum dz/maxiter length. However, as stated above, in each iteration of the inner loop two-step method size `dz/2` are performed, as is the symmetrized method, one for the non-linear part and the other for the dispersive . The latter with the sole purpose of obtaining the value of intermediate pulse with which to rebuild the nonlinear factor.

First a variable `u1` is created with the same value as the input signal `u0`, so that you can override u1 in the loop of short steps to calculate intermediate nonlinear operators, retaining the u0 will only be modified at each hop.

This procedure implements the descriptive equation `split-step Fourier method` I symmetrized seen in the previous section in which three successive multiplications the input pulse are performed.

$$A(t, z+dz) = e^{(j\frac{\beta}{2}(\omega-\omega_0)^2\frac{dz}{2})} \;\; e^{(\int_z^{z+dz} j\gamma|A|^2 dz)} \;\; e^{(j\frac{\beta}{2}(\omega-\omega_0)^2\frac{dz}{2})} \; A(z,t)$$

The first, the dispersive operator for a semisalto `dz/2` calculated in the `ssfUP()` function and is saved in the variable `Halfstep`; the second, the non-linear operator represented by the exponential of the integral solution of the nonlinear Schrödinger equation after replacing, by the approach of the trapezius, the amount of said solution with its shifted by a dz jump:

$$\int_z^{z+dz} j\gamma \left|u(z')\right|^2 dz' \approx j\gamma[|u0|^2 + |u1|^2]dz/2$$

and the third, again the nonlinear operator for a dz/2 semisalto.

The use of the dispersive part in intermediate hops will only serve to calculate values of the signal in those jumps that find the new nonlinear factor dependent

on them. To carry out the first application of nonlinear operator, since the value of the signal one `dz/2` is not known after the jump in question, it replaces the `u1` by `u0` . The pulse will only modify its value due to the dispersive part in each of the iterations of the main loop, which is responsible for the displacement `dz`. the intermediate values of the function in the time domain is stored in a variable `uv`, while in the frequency domain will do so in the variable `ufft`.

To control the spacing between small displacement is not so large as to cause a significant difference in the values of the current and previous pulses, also impact on a wrong solution of the nonlinear operator, error handling is performed by calculating the relative error of the signal obtained at each iteration with respect to the preceding.

$$error = \frac{||uv - u1||^2}{||u1||^2}$$

For this calculation the standard two of the line of both is used, element by element, with the function of numpy, `linalg.norm()`.

If the error is greater than the value we have gone through parameter for tolerance, the loop is broken and passed directly to the next `dz` long after overwritten variable `u1` with the result obtained in this latest advance `uv`. Thus, it has to play with the value assigned to `dz`, because if it is high will shorten run times significantly but will decrease the accuracy of the method.

If you reach the total of iterations assigned to inner loop without having reached the limit imposed tolerance, we assume that has not reached the allowable value for the nonlinear operator, ie, convergence has not been the required minimum so that the previous and next signals depart too far to continue to perform the calculations. To control this situation, an exception convergence to break the loop with the resulting error is included. Therefore we play with these two criteria; on the one hand, the method allows us to reach the limit of tolerance

and, secondly, not to exceed.

When you have made all the intermediate steps for the implementation of factor nonlinearity, the nonlinear phase is calculated as the sum of nonlinear partial phases of all the jumps *dz* above. By últimpo the *u0* function is overwritten and is carried back all the way to a new leap in size *dz*.

```
def ssfDOWN1(u0, dz, nz, ufft, halfstep, gamma, maxiter, tol = 1e-5, phiNLOut = False):

    phiNL = 0.0
    u1=u0
    for iz in arange(nz):
       # primer operador dispersivo
       uhalf = fftpack.ifft(ufft*halfstep)
       for ii in arange(maxiter):
          # operador no lineal
          uv = uhalf * exp(1.0j*gamma*(pow(abs(u1),2.0) + pow(abs(u0),2.0))*dz/2.0)
          ufft = fftpack.fft(uv)
          # segundo operador dispersivo
          ufft = halfstep*ufft
          uv = fftpack.ifft(ufft)
          error = linalg.norm(uv-u1,2.0)/linalg.norm(u1,2.0)
          u1 = uv
          if (error <tol):
             break
          if (ii == maxiter):
             raise Exception, "Failed to converge",
       phiNL += gamma*dz*pow(abs(u1),2).max()
       u0 = u1
    if phiNLOut:
       return [u0, phiNL]
    else:
       return u0
```

The *ssfDOWN2()* function performs the same operation without symmetrise, with a step size *dt* instead of *dt/2*, so it is not necessary to use a second loop *for* to calculate intermediate values. At each hop, the pulse resulting from the previous iteration is multiplied, or received by the initial parameters, the spreading factor and the inverse Fourier transform is performed to pass the time domain. The obtained value is multiplied by the initial as an input parameter nonlinear factor performed with the pulse value in the previous iteration, or to

obtain pulse intensity and turns to perform the Fourier transform to go back to the domain frequency. We update the input variable *up* and we make another leap in size *dz*.

```
def ssfDOWN2(up, dz, nz, ufft, linearstep, gamma):
     for iz in arange(nz):
        # 1er Application de l'operateur lineaire
        uhalf = fftpack.ifft(linearstep*ufft)
        # Application de l'operateur nonlineaire
        uv = uhalf * exp(-1.0j*gamma*(pow(abs(up),2.0))*dz)
        ufft = fftpack.fft(uv)
        up = uv
     return uv
```

Of these two methods, the latter is obviously having a shorter run, a difference that will increase as you increase the number of iterations of progress whatever you want to apply.

The *ssfDOWN1()* function must make a second loop to calculate the intensity of the nonlinear factor in *dz/2*, for which can be made several iterations, not recommended more than four, with two would enough. In each were carried out six operations: multiplication by the nonlinear factor, Fourier transform, multiplication factor dispersive, Fourier inverse transform, error calculation and comparison with the tolerance imposed; amount should be multiplied by the number of iterations, as much *maxiter* with a default value of four and at least two, and adding the first multiplication factor of dispersive and Fourier inverse transform initial. In total, at least, this method will contain fourteen symmetrized operations in each jump *dz*, with a maximum of twenty-six.

With the second method not only symmetrized each full iteration will consist of four steps: two Fourier transforms, one direct and one reverse, and two multiplications, one by the dispersive factor and one by the nonlinear. However, this second method has a higher error associated previous and achieved much less accurately. As stated above, the accuracy of the method will be marked by the size of jump, a smaller size *dz* greater precision in the final results will be

obtained but, in return, should be made more iterations to travel the same stretch z coordinate.

Therefore, to achieve a solution more accurate pulse propagation along the optical fiber, we should use the symmetrized method and a small step size. However, if we choose these premises for the solution of the nonlinear Schrödinger equation, the time cost will be increased considerably.

One solution to this handicap is parallelize that part of the code that can be divided into temporary sections so that operations are conducted in installments rather than sequentially, where each section involving a thread in the whole process. Since these parallel threads perform the same operations, most likely they reach the next step at the same time, or the differences are minimal. This condition can be forced through a synchronization method, where the first of the threads to finish an operation expected to last to start the next time.

Functions implemented for execution on the GPU

Again, this section consists of several functions with virtually the same code except for minor differences related to the presentation, or not, of certain results. For the sake of saving code, it has decided to simplify these functions on an original single *ssfgpu1()* call to which all others. This feature is not divided into two parts, an upper and a lower, because the other functions that vary significantly not call any of them and is not necessary. The method call can be made in two ways: directly to each of the functions with the parameters given by default, or through a new feature called *ssfgpu()* with two *if-else* chained:

```
    def ssfgpu(u0, dt, dz, nz, alpha, betap, gamma, maxiter, tol = 1e-5, context = False,
phiNLOut = False, fullOutput=False):
        if context == False:
            return ssfgpuStd(u0, dt, dz, nz, alpha, betap, gamma, maxiter, tol)
        else:
            if fullOutput:
```

```
            return ssfgpuFull(u0, dt, dz, nz, alpha, betap, gamma, maxiter, tol, context,
phiNLOut)
        else:
            return ssfgpu2(u0, dt, dz, nz, alpha, betap, gamma, maxiter, tol, context,
phiNLOut)
```

The two new parameters to select the functions to be executed and which were not implemented in non parallelizing are *context*, which indicate whether implemented a context, or not, for execution on the GPU, and *fullOutput* for the call to the function *ssfgpuFull()*, which will return the intensity of the resulting pulse in addition to other data.

As *method split-step Fourier symmetrized* is the same as it is done in paralelizante or not, that part of the function that is of interest will be discussed obviating those already explained in the previous section.

First, if the parameter boolean *fullOutput* is True, we initialize the variable *Uarch* with the function of numpy *ones()*, where the values of the intensities of the input pulse will be saved. This function will fill about a matrix of nz * nt.

```
if fullOutput:
    uArch = numpy.ones((nz, nt))
```

It must verify that the values of the input pulse are in single-precision complex numbers, for which the incoming signal with itself is resized using the of PyCUDA astype function. This value is stored in a new variable *v1*, and the non paralelizante method:

```
u0 = u0.astype(numpy.complex64)
u1 = u0
```

The new variable created to save the dispersive factor also be resized using this method.

The *ssfgpu1()* function, in principle, worked with the library *fftpy()* to perform Fourier transforms on the gpu. The problem arose when using these libraries, it was that had become obsolete and did not recognize gpu. For this reason the methodology has been modified to work in the gpu with the *reikna*

library, created in PyCUDA for manipulating vectors and matrices, as well as functions and mathematical algorithms implemented in different blocks.

If the parameter *context* is selected as *False*, the driver of the graph will be initialized and an instance *pycuda.driver.context* will be created from the function *make_default_context()* of the library *tools* of PyCUDA. This instance will be chosen based on three conditions:

1. If the variable of the CUDA device environment is established, its integer value will be used as the device number.

2. If the *device.cuda* is in the root directory of the user, its integer value will be used as the device number.

3. Otherwise, all CUDA devices available will be chosen in order of priority.

If you do not believe this context, the devices will also be chosen by priority regardless of any rule. That is, files and existing environment variables are ignored.

```
if context == False:
    cuda.init()
    context = make_default_context()
```

Then he will call a procedure *reikna* to differentiate whether we are working with CUDA. This procedure is in the *reikna.cluda* folder and is named *cuda_api()*, to work with CUDA, and *OCL_api()*, to do with OpenCL. This procedure will be stored in an environment variable that call *api*. It is in this environment variable which must be created involving the Thread object context created. With this Thread specified context and queue in which the kernel that will be used will be stored. If no parameter is not specified, will use the available device.

```
api = cluda.cuda_api()
thr = api.Thread.create()
```

To use the function *FFT()* of reikna, you must first initialize a vector or matrix of the same size as the signal to be transformed. This matrix is filled

in a accordance with the procedure described above where he will format single complex number. Then it is created with a variable FFT data made to that matrix. These data can be modified at any time just to use the *compile(signal)* method and the signal we want to transform. For the fft work with the parameters used in the object *Thread()* created, enter it in *compile()*.

```
data = numpy.ones((1, nt), dtype=numpy.complex64)
fft = FFT(data)
thr_fft = fft.compile(thr)
```

To work in the gpu with vectors, first reserve a certain space in its global memory. For this operation to create Thread() object device for each of the variables that you want to work in parallel is used.

seven variables are created in principle for dispersive operator, the signal source, the copy of the source signal, the signal values each dt/2, the values of the intermediate signal in the time domain, the signal values intermediate in the frequency domain and nonlinear phase.

The first three are created in the gpu making an exact copy of existing in the CPU, for what the *to_device(buffer)* method of *driver* PyCUDA be used. With this method reserves memory space according to the saved copy in the buffer, then copy the buffer to the gpu returning an object that indicates where the data are located, *DeviceAllocation*.

By contrast, for the last four vectors, the method used is *empty _like (array_gpu) pycuda.gpuarray*, which creates a new gpuarray with the same properties as the sample passed.

```
gpu_halfstep = thr.to_device(halfstep)
gpu_u0 = thr.to_device(u0)
gpu_u1 = thr.to_device(u1)
gpu_uhalf = thr.empty_like(gpu_u0)
gpu_uv = thr.empty_like(gpu_u0)
gpu_ufft = thr.empty_like(gpu_u0)
phiNL1 = thr.empty_like(gpu_u0)
```

With the method created for performing the Fourier transform on the gpu,

we turn to the frequency domain the original pulse *gpu_u0* and store it in a new memory space called *gpu_ufft*, reserved for this purpose.

```
thr_fft(gpu_ufft, gpu_u0)
```

So far no operation has been no parallel using threads or streams created by reikna. In fact, all operations that can be found in the paralyzed part of the method iteratively describing the same sequence as that performed in the CPU functions except making calls kernels that must be previously defined. That is, the loops are made on the local host memory and operations in global memory device with the computational cost that may involve. However, spending minimal time since the communication between devices get great speeds, plus high-speed graphics processing outweigh these potential gaps.

For the implementation of *split-step Fourier method symmetrized*, the signal is fed into the frequency domain in the main loop. Is carried in him the call to the first kernel is called *halfstepKernel()* and is responsible to multiply those stored in *gpu_ufft* by the dispersive factor passed previously gpu values, *gpu_halfstep*.

```
   halfStepKernel = ElementwiseKernel("pycuda::complex<float>*u, pycuda::complex<float>*halfstep,
pycuda::complex<float>*uhalf",
       ühalf[i] = u[i] * halfstep[i]",
       "halfstep_linear",
       preamble="#include <pycuda-complex.hpp>")
```

This call to the kernel is done through the *ElementwiseKernel()* method PyCUDA.elementwise imported from the library. This kernel contains a number of input arguments either scalar or vector and scalar operations performed on each entry of its case if it were a table. The arguments are specified as a string in C language format operation is specified as an estate allocation in C without semicolons. Each of the input vectors are divided into a thread count defined by the variable [i]. the name to be recompiled kernel and a preamble that is filled with the header function that can contain the kind of language to use is specified,

in this case *pycuda-complex.hpp*, which is a C extension for use of complex numbers.

The data obtained in the execution of this kernel are passed to the time domain through an inverse Fourier transform by the *thr_fft()* method with the boobleano parameter *inverse* set to *true*, default is *false*. The result is saved in the variable *gpu_uhalf*.

In implementing the second loop *for*, to obtain the nonlinear factor and multiplying by the step signal obtained in the previous kernel it is called a kernel named *nlKernel()* using the same procedure *ElementwiseKernel()* with seven input arguments: four complex vectors that can be divided into sections controlled by wires, and three float floating point or integer. Because the language does not allow work with complex numbers, first they are separated into their real and imaginary parts of the vectors *u1(u1._M_re, u1._M_im)* and *u0(u0._M_re, u0._M_im)*. Nor does it allow the use of complex exponentials, such as nonlinear and dispersive factors, so that must be passed to the exponential rectangular mode using Euler's theorem of complex numbers $e^{j\phi} = cos\phi + jsen\phi$. For this intermediate variables are used *euler1* and *euler2*, the first for the real part of the nonlinear factor and the second for the imaginary part.

```
    nlKernel = ElementwiseKernel("pycuda::complex<float>*uhalf, pycuda::complex<float>*u0,
pycuda::complex<float>*u1, pycuda::complex<float>*uv, float gamma, float dz, float phinl",
        "
        float phinl[i] = gamma*dz*u1[i]
        float u0_int = pow(u0[i]._M_re,2) + pow(u0[i]._M_im,2);
        float u1_int = pow(u1[i]._M_re,2) + pow(u1[i]._M_im,2);
        float realArg = -gamma*(u1_int + u0_int)*dz;
        float euler1 = cos(realArg);
        float euler2 = sin(realArg);
        uv[i]._M_re = uhalf[i]._M_re * euler1 - uhalf[i]._M_im * euler2;
        uv[i]._M_im = uhalf[i]._M_im * euler1 + uhalf[i]._M_re * euler2;
        ",
        "halfstep_nonlinear",
        preamble="#include <pycuda-complex.hpp>")
```

The parameter *phinl* return nonlinear phase all intermediate data pulses.

Then the maximum value of each pulse to be added to all the above be taken. The vector *uv*, which is returned as a result of the multiplication *uhalf * factor_No_lineal* be calculated by separating its real part and its imaginary part.

The next step is to perform the Fourier transform of the resultant vector to carry out the second multiplication in the frequency domain by the dispersive factor. To perform this multiplication is a call made to the kernel *halfstepKernel*, as explained above. The multiplication obtained is passed back to the time domain using the method *thr_fft* with inverse set to *True*, where will the new intermediate pulse to calculate the nonlinear factor of the next iteration or overwrite the vector of the final pulse.

To calculate the relative error of each iteration using another call is made to a new kernel with the *ReductionKernel* method.

```
computeError = ReductionKernel(numpy.float32, neutral="0", reduce_expr = "ä+b ",
map_expr="pow(abs(a[i] - b[i]),2)",
   arguments="pycuda::complex<float>*a, pycuda::complex<float>*b",
   name=.error_reduction",
   preamble="#include <pycuda-complex.hpp>")
```

This has an argument called reduced _expr where the parameters to reduce indicating, another argument called map _expr that give expression to perform with the input parameters, the input parameters in arguments, the name of the kernel function and preamble to the header indicating the file source programming language. In this case, you indicate on the map _expr that the square of the line of input vectors is made; while the reduced _expr add the values obtained in the map _expresion, to get in the end the square of the standard two of that line.

$$||u1[i] - uv[i]||^2 = (u1[1] - uv[1])^2 + (u1[2] - uv[2])^2 + (u1[3] - uv[3])^2 + ... + (u1[i] - uv[i])^2$$

To calculate the relative error is passed to the CPU the vector obtained by the function *.get()* and the result is divided by the square of the standard two vector *gpu _u1()* whose I passed the cpu result is stored in the variable e _ini.

To calculate this last vector we use the same *ReductionKernel* with one of the parameters to zero.

Once each iteration of the inner loop must overwrite the variable *gpu_u1* with the data obtained in this iteration. This requires a call is made to a new kernel, *complexDeepCopy*, with the above described method ElementwiseKernel. This kernel has two arguments, the first output and the second input. As the C employing extended kernels not allowed to work with complex, equality between vectors element by element, part imaginary to real part imaginary part and real part must occur.

```
complexDeepCopy = ElementwiseKernel("pycuda::complex<float>*u1,
pycuda::complex<float>*u2",
   ü1[i]._M_re = u2[i]._M_re;u1[i]._M_im = u2[i]._M_im",
   "gpuarray_deepcopy",
   preamble="#include <pycuda-complex.hpp>")
```

To control that all possible iterations marked by *maxiter*, the program performs an exception indicating failure of convergence are not met. If it fails to exceed the tolerance marked in total iterations, the signal is not close to its previous value and will not be accurate enough to consider the correct simulation. In that case, you should change the input parameters.

the partial sum of the maximum value of the nonlinear phase is carried out with the above if the parameter *phiNLOut* is *True* and pulse intensity is calculated as the square of the magnitude of the signal *gpu_u1* after passing it to the CPU with the function *.get()* if the parameter *also* fullOutput find *True*. This intensity value will saving a vector *Uarch[]* along the loop jumps *dz*.

```
for iz in arange(nz):
   halfStepKernel(gpu_ufft, gpu_halfstep, gpu_ufft)
   thr_fft(gpu_uhalf, gpu_ufft, inverse=True)
   for ii in arange(maxiter):
      nlKernel(gpu_uhalf, gpu_u1, gpu_u0, gpu_uv, float(gamma), float(dz/2.0), float(phiNL1))
      thr_fft(gpu_uv, gpu_uv)
      halfStepKernel(gpu_uv, gpu_halfstep, gpu_ufft)
      thr_fft(gpu_uv, gpu_ufft, inverse=True)
```

```
        e_ini = computeError(gpu_u1, 0).get()
        error = computeError(gpu_u1, gpu_uv).get() / e_ini
        complexDeepCopy(gpu_u1, gpu_uv)
        if(error <tol):
           break
        if (ii == maxiter):
           if context:
              context.pop()
           raise Exception, "Failed to converge",
     complexDeepCopy(gpu_u0, gpu_u1)
     if (phiNLOut):
        phiNL += phiNL1.get().max()
     if fullOutput:
        uArch[iz] = pow(abs(gpu_u1.get()),2)
```

When you have made the number of jumps that had nz stipulated by parameter, you leave for the final value of the pulse gpu_u0 to the CPU, the context is passed closes created with context.pop() function, always the context input parameter to the function is to True; and the values obtained based on the parameters fullOutput phinLOut and returns.

```
u1 = gpu_u0.get()
if context == False:
   context.pop()
if phiNLOut:
   if fullOutput:
      return [u1, uArch, phiNL]
   else:
      return [u1, phiNL]
else:
   if fullOutput:
      return [u1, uArch]
   else:
      return u1
```

3.1.2. Ejemplos de utilización del archivo ssf.py en la librería PyOFTK

En la librería PyOFTK hay una sección dedicada a la realización de ejemplos correspondientes a las diferentes utilidades y campos de estudio que abarca. El primero de los ejemplos (*agrawal_fig3_1_ssf.py*, deducido del libro *"Fibras ópticas no lineales", Agrawal, 2001*) está dedicado a la presentación en una gráfica de un pulso gaussiano ensanchado por los efectos de la dispersión cromática. Para ello hace pasar el pulso creado con la función gaussianPulse() del archivo pulse.py por la función ssf() vista en el apartado anterior. Para la creación del pulso gaussiano en primer lugar realiza una linea temporal, linspace centrada en cero con un ancho temporal T de 32.0 y un número de muestras nt de dos elevado a diez. Por tanto la distancia entre muestras temporales será de $dt = T/nt = 32/(2^{10}) = 32/1024 = 0{,}03125$. La función gaussianPulse() tiene seis parámetros de entrada:

1. *t* = linea temporal, linspace(-T/2,T/2,nt).

2. *FWHM* = full-width at half-maximum intensity of pulse, por defecto es uno. Este ejemplo tiene un valor de 2.0.

3. *t0* = origen de tiempos para el centro del pulso. Este ejemplo lo tiene centrado en el 0.0.

4. *P0* = Intensidad pico del pulso, donde t=t0, por defecto es uno.

5. *m* = orden gaussiano, por defecto es uno.

6. *C* = chirp parameter, o efecto de chirrido. Por defecto será cero.

La expresión matemática que es utilizada para generar el pulso gaussiano es la siguiente:

$$u(t) = \sqrt{P_0}\ \ exp\left[-\left(\frac{(t-t_0)}{\frac{FWHM}{\sqrt{4log(2)}}}\right)^{2m}\frac{(1-jC)}{2}\right]$$

Para lo que está implementada la siguiente función:

```
def gaussianPulse(t, FWHM, t0, P0 = 1.0, m = 1, C = 0):
   t_zero = FWHM/sqrt(4.0*log(2.0))
   amp = sqrt(P0)
   real_exp_arg = -pow(((t-t0)/t_zero),2.0*m)/2.0
   euler1 = cos(-C*real_exp_arg)
   euler2 = sin(-C*real_exp_arg)
   return amp*exp(real_exp_arg)*euler1 + amp*exp(real_exp_arg)*euler2*1.0j
```

La salida de esta función, un pulso gaussiano de amplitud unidad y centrado en el origen, se hace pasar a la función ssf con un dz, o tamaño de paso, de 2.0; un nz, o número de saltos, de 2; sin pérdidas, es decir, con constante de atenuación α igual a cero, un valor de VGD o β_2 de 1.0, incluido en el array de tres elementos; un coeficiente de no linealidad o γ también nulo; un número de iteraciones para la convergencia del factor no lineal, maxiter, de 10; y, por último, una tolerancia de 10 micras o 1e-5. A estos argumentos originales, habría que añadir el booleano por si queremos que nos devuelva la fase no lineal, por defecto es False, y el método iterativo que se quiere emplear, el simplificado o el simetrizado, para lo que se emplea otro booleano llamado down1 que por defecto es True.

Dado que la constante de propagación elegida es la dispersión de velocidad de grupo, la gráfica del pulso resultante deberá salir ensanchada respecto al pulso gaussiano original:

Se puede comprobar que, puesto que no ha habido pérdidas de energía, cuanto más se ensanche el pulso, más debe disminuir su intensidad para que el área que abarca siga siendo la misma.

Estos pulsos presentan una dispersión normal puesto que se ha forzado una VGD positiva. En este caso, las frecuencias superiores a la central viajan a más velocidad y tienden a desplazarse hacia el flanco de subida del pulso temporal, mientras que las frecuencias menores viajan a menor velocidad desplazándose hacia el flanco de bajada del pulso temporal. Esto provoca que el pulso tienda a ensancharse.

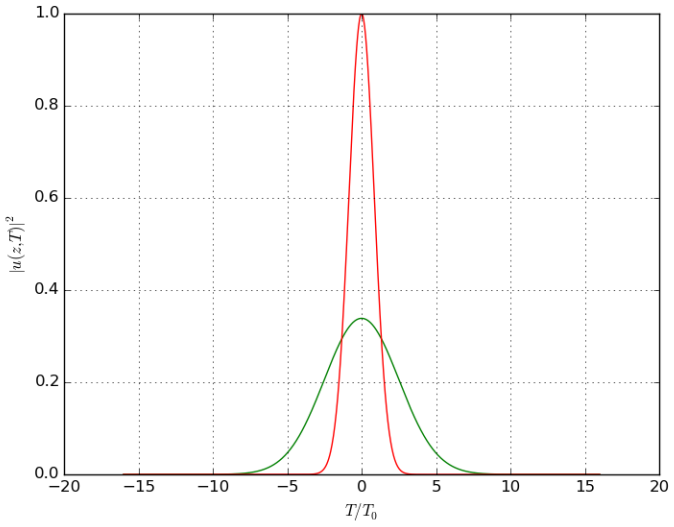


Figura 3.1: Pulso gaussiano y resultante con $\beta\_2 = 1{,}0$.

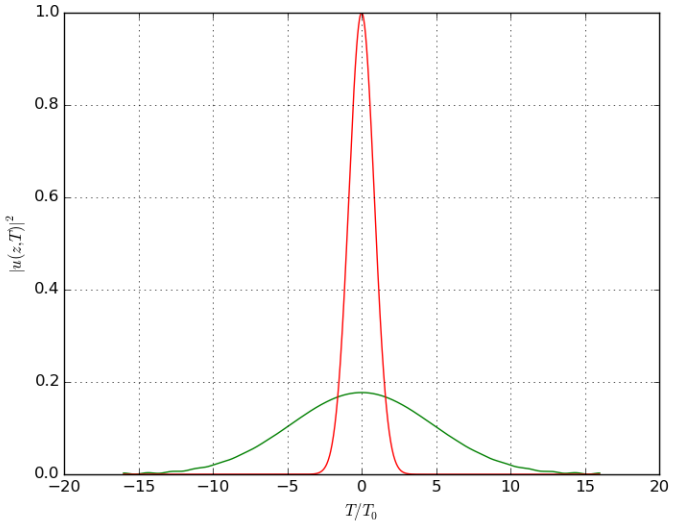


Figura 3.2: Pulso gaussiano y resultante con β_2=2.0.

El efecto contrario se produce en el régimen de dispersión anómala, donde la VGD es de signo negativo, la longitud de onda es mayor a la longitud de onda de dispersión cero. En este régimen es donde se producen los solitones y ondas

solitarias, que viajan a través de la fibra sin dispersión por un balance entre los efectos lineales y no lineales. Para que esto se produzca, la dispersión cromática de la fibra se debe compensar totalmente con la dispersión introducida por la no linealidad a través de la SMP, modulación de fase propia.

Si recordamos la ecuación de Schrödinger para anchos de pulsos muy cortos en fibras:

$$\frac{\partial A}{\partial z} = \frac{\alpha}{2}A - j\frac{\beta_2}{2}\frac{\partial^2 A}{\partial t^2} - j\gamma\left|A\right|^2 A$$

Donde A es la variación lenta de la amplitud de la envolvente del pulso y T es la línea temporal de referencia con el pulso moviéndose a la velocidad de grupo a lo largo del eje z. Dependiendo de la anchura del pulso T_0 y de la potencia pico P_0, los efectos dispersivos o de no linealidad dominarán a lo largo de la fibra. Se puede reescribir esta ecuación para una amplitud normalizada U, en función de una escala temporal normalizada τ y de dos longitudes escalares, L_D y L_{NL}, que representan aquellas longitudes donde la dispersión y no linealidad comienzan a ser importantes para la propagación del pulso:

$$U(z,\tau) = \frac{A(z,\tau)}{\sqrt{P_0}e^{-\alpha z/2}}$$

Al derivar A en la ecuación diferencial obtenemos términos de la constante de pérdidas α que se anulan con el ya existente.

$$\tau = \frac{T}{T_0};\quad L_D = \frac{T_0^2}{|\beta_2|};\quad L_{NL} = \frac{1}{\gamma P_0}$$

$$\frac{\partial U}{\partial z} = -j\frac{sgn(\beta_2)}{2L_D}\frac{\partial^2 U}{\partial \tau^2} - \frac{1}{L_{NL}}\left|U\right|^2 U$$

Dependiendo de las longitudes de dispersión y no linealidades en función de la longitud total de la fibra, se puede clasificar los efectos en cuatro categorías:

- Si $L << L_{NL}$ y $L << L_D$.

  Ninguno de los efectos dispersivos o no lineales juegan un papel importante en la transmisión del pulso. En este caso, se pueden despreciar ambos términos de la ecuación diferencial.

  Esto provoca que la variación temporal del pulso sea muy suave con una derivada segunda respecto al tiempo igual a uno, manteniendo la forma a lo largo de la propagación: $U(z,\tau) = U(0,\tau)$.

  Las características de la fibra no juegan un papel importante en este régimen, salvo en el caso de tener un coeficiente de pérdidas significativo.

  Este régimen es útil para los sistemas de comunicaciones ópticas.

  Si se conocen los valores de las longitudes de la fibra, de dispersión y de no linealidad, se pueden deducir la T_0 y P_0 para un sistema de comunicación. A partir de la curva de dispersión que relaciona la β_2 con la longitud de onda, se puede saber el valor de la primera; y el coeficiente de no linealidad dependerá del tipo de fibra a utilizar y del pulso inyectado a través de la frecuencia de portadora, el índice de no linealidad, la velocidad de la luz en el vacío y la apertura efectiva de la fibra.

  Para una fibra óptica estándar que trabaja en tercera ventana, con una línea de L=50km, en este régimen las longitudes de dispersión y de no linealidad deberán ser mayores a 500km:

$$\lambda = 1{,}55\mu m; \quad |\beta_2| \approx 20ps^2/km; \quad \gamma \approx 3W^{-1}km^{-1}$$

$$L_D = 500Km = \frac{T_0^2}{20ps^2/Km}$$

$$T_0 = \sqrt{500Km * 20ps^2/Km} = 100ps$$

$$L_{NL} = 500Km = \frac{1}{3W^{-1}Km^{-1}P_0}$$

$$P_0 = \frac{1}{500Km * 3(KmW)^{-1}} = 0{,}667mW$$

En la función del pulso gaussiano antes presentada, la T0 no se corresponde con esta T0, como es obvio. Esta T0 se refiere al ancho del pulso óptico caracterizado por FWHM del siguiente modo:

$$T_{zero} = \frac{FWHM}{\sqrt{4log2}}; \quad FWHM = 100ps * \sqrt{4log2} = 166{,}5ps$$

Si introducimos los datos obtenidos en el archivo del ejemplo, se puede ver la forma del pulso resultante para distintas posiciones a lo largo del eje z, con tal de variar la nz, o el número de pasos del método split step de Fourier simetrizado.

También se ha de tener en cuenta las unidades en las que trabaja el programa, ya que de introducir cantidades no proporcionales, los resultados serán erróneos. En primer lugar todas las presentadas en los cálculos precedentes son las mismas que las introducidas en la función, salvo el coeficiente de no linealidad, al que debemos transformar en $mW^{-1}Km^{-1}$ ya que la P_0 es presentada en mW, cuestión que se consigue dividiendo entre mil la cantidad aportada. Por tanto, introduciremos en la función un $\gamma = 0{,}003$.

Los datos resultantes para la envolvente del pulso en esta ventana espacial y este número de muestras, se muestran en la figura 3.3 para gráficas con distancias equiespaciadas cada 50nz.

Tenemos una dz=2km, puesto que estamos trabajando en unidades de kilómetros. Para estas unidades, se puede comprobar en la gráfica que no comienza a apreciarse la distorsión de la señal hasta los 100 primeros

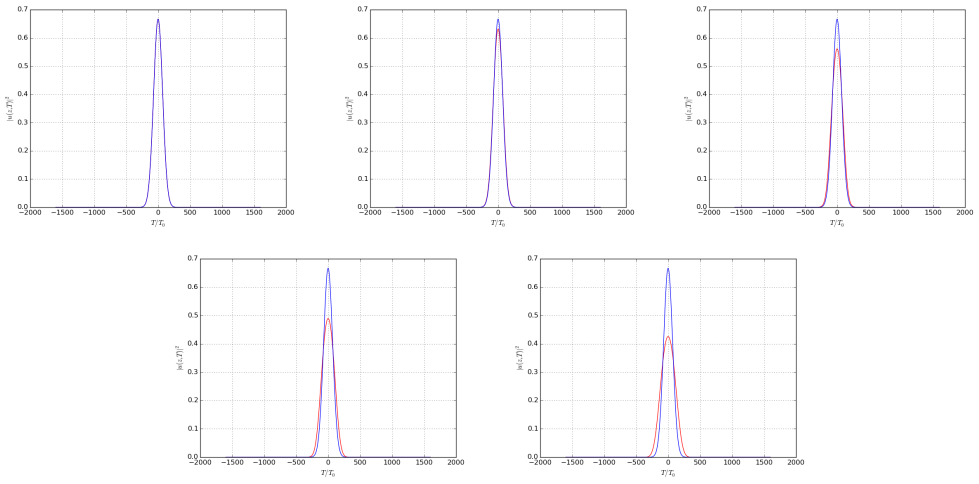


Figura 3.3: Pulso gaussiano y pulso en primer régimen de dispersión y no linealidad para nz=1, nz=50, nz=100, nz=150 y nz=200.

kilómetros, lo que corrobora con lo indicado para este primer régimen. La no linealidad es prácticamente despreciable para todas las distancias, mientras que la distorsión puede ser despreciada para una línea de 50km.

Para comprobar los resultados obtenidos haremos uso del software desarrollado por el grupo de fotónica y optoelectrónica de la Universidad de Rochester para la representación gráfica de distintos tipos de pulsos, utilizando el método split-step para la resolución de la ecuación de Schrödinger.

En primer lugar, tan pronto abrimos la aplicación NLSE, se presenta ante nosotros una ventana con cuatro secciones. En la primera de ellas, arriba a la izquierda, nos encontramos con la sección *Select The Fiber Parameters* con las constantes α de atenuación, γ de no linealidad, β_2 de DVG, dispersión de velocidad de grupo, y β_3 del TOD, dispersión de tercer orden. En el segundo recuadro, arriba a la derecha, nos encontramos con la sección *Select The Input Field* con los apartados de forma, como un combo con varios tipos de pulsos: gaussiano, secante e hipergaussiano.

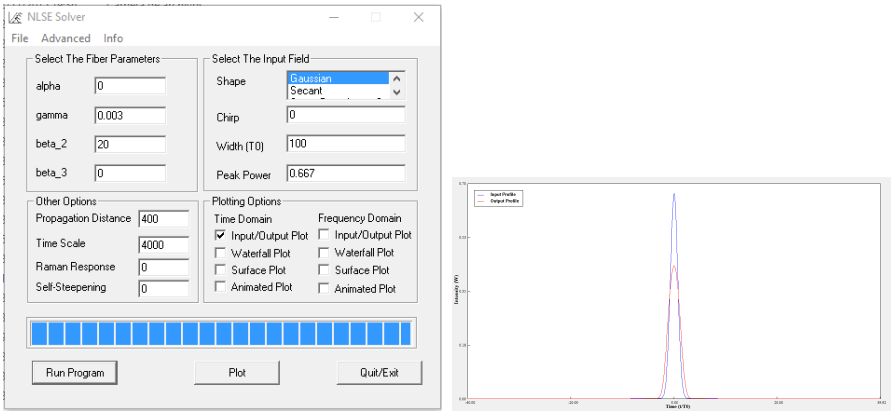


Figura 3.4: Pulso gaussiano y pulso en primer régimen de dispersión y no linealidad para nz=200 a través del programa NLSE de la Universidad de Rochester.

La siguiente opción será la de chirpeo que mantendremos a cero en todo instante, para continuar con la de ancho del pulso a amplitud 1/e, T0, y por último, la potencia pico. La tercera sección, `Other Options`, se compone de la distancia de propagación en km, de la escala de tiempos en ps, de la respuesta Raman que no vamos a utilizar, y la opción de autoparada. La cuarta y última sección, `Plotting Options`, queda dividida en dos subsecciones, la primera para el dominio del tiempo y la segunda para el domino de la frecuencia, con mismos items para ambas. La primera de las opciones, input/output Plot, será la elegida en todas las comprobaciones que realicemos, ya que se ajusta a las representadas anteriormente. La segunda, Waterfail Plot, se corresponde con una representación tridimensional de la propagación del pulso. La tercera, Surface Plot, una nueva representación tridimensional con progresión de forma en función de la propagación espacial. Por último, Animated Plot, representa una imagen gif o animada de la propagación del pulso en 2D en función de la distancia recorrida.

A tenor de los resultados obtenidos, podemos comprobar que en primer régimen de propagación del pulso la atenuación se ajusta a lo esperado,

ya que a 400km tanto la forma y amplitud en la aplicación realizada en PyOFTK como las de la Universidad de Rochester coinciden.

Dentro de este primer régimen de propagación, aunque podrían incluirse en cualquier de ellos, vamos a realizar un ejemplo práctico con el uso de la función `sellmeier(conger, wavelength)`, que calcula el índice de refracción de una fibra de silicio dopada con una cierta cantidad de GeO2 entrado por parámetro. El GeO2 suele emplearse para aumentar el índice de refracción en la fabricación de las fibras ópticas. A partir de Íos resultados obtenidos, emplearemos la ecuación resultante para la GVD en función de dicho índice de refracción. De esta forma, se puede caracterizar la fibra óptica a una determinada longitud de onda y unos determinados parámetros materiales. El código realizado puede consultarse en D. Los resultados obtenidos se recogen en dos gráficas, la primera que relaciona la GVD con la longitud de onda de trabajo , y la segunda que hace lo propio con el parámetros de dispersión D.

$$\beta_2 = \frac{\lambda^3}{n(2\pi c)^2}\left[4\lambda^2\left(\frac{B_1C_1}{(\lambda^2-C_1)^3}+\frac{B_2C_2}{(\lambda^2-C_2)^3}+\frac{B_3C_3}{(\lambda^2-C_3)^3}\right)\right]\ldots$$

$$\ldots-\frac{\lambda^3}{n(2\pi c)^2}\left[\left(\frac{B_1C_1}{(\lambda^2-C_1)^2}+\frac{B_2C_2}{(\lambda^2-C_2)^2}+\frac{B_3C_3}{(\lambda^2-C_3)^2}\right)+\frac{\lambda^2}{n}\left(\frac{B_1C_1}{(\lambda^2-C_1)^2}+\frac{B_2C_2}{(\lambda^2-C_2)^2}+\frac{B_3C_3}{(\lambda^2-C_3)^2}\right)^2\right]$$

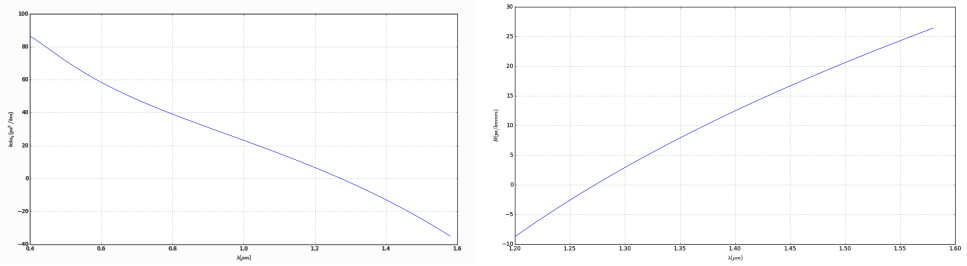


Figura 3.5: Gráficas de β_2 y del parámetro de dispersión D en función de la longitud de onda.

Se puede comprobar que, para unas impurezas de GeO2 con un valor de 5

introducidas en la función sellmeier, la GVD obtenida se ajusta considerablemente a la curva que relaciona dicho parámetro en el libro de referencia de Agrawal [1] en su página 9 (figure 1.5). Lo mismo se puede decir para el parámetro de dispersión. Lo que hace entender que tanto la función sellmeier() como el desarrollo de la ecuación que relaciona dicha función con β_2 son correctos y que la GVD puede indicarnos la longitud de onda de trabajo, si nos encontramos en segunda o tercera ventana, etc, siempre que conozcamos el tipo de fibra que se está utilizando, material y grado de impureza. A paritr de esta gráfica, se segurián utilizando los valores convenientes de la GVD para caracterizar los distintos regímentes de propagación o alguna característica necesaria, sin atender a la longitud de onda de trabajo, simplemente al grado de dispersión relacionado.

- Si $L \ll L_{NL}$ y $L \approx L_D$, el término de no linealidad puede ser despreciado en la ecuación de Schrödinger, a diferencia de los otros dos. En este caso, se tendrá en cuenta la relación entre las longitudes de dispersión y de no linealidad para calcular aproximadamente los valores de T_{zero} y de P_0, obteniendo uno en función del otro:

$$\frac{L_D}{L_{NL}} = \frac{\gamma P_0 T_{zero}^2}{|\beta_2|} \ll 1$$

Podemos encontrar para un pulso que se propaga en tercera ventana con los coeficientes de dispersión y no linealidad vistos en el régimen anterior, una potencia pico determinada para un ancho de pulso de 1ps:

$$P_0 \ll \frac{|\beta_2|}{\gamma T_{zero}^2} = \frac{20ps^2/km}{3W^{-1}km^{-1} * 1ps^2} = 6{,}667W$$

Esto supone una potencia muy por debajo del watio, que se puede hacer

coincidir con la del caso anterior, 0,667mW, para una anchura de pulso mucho menor.

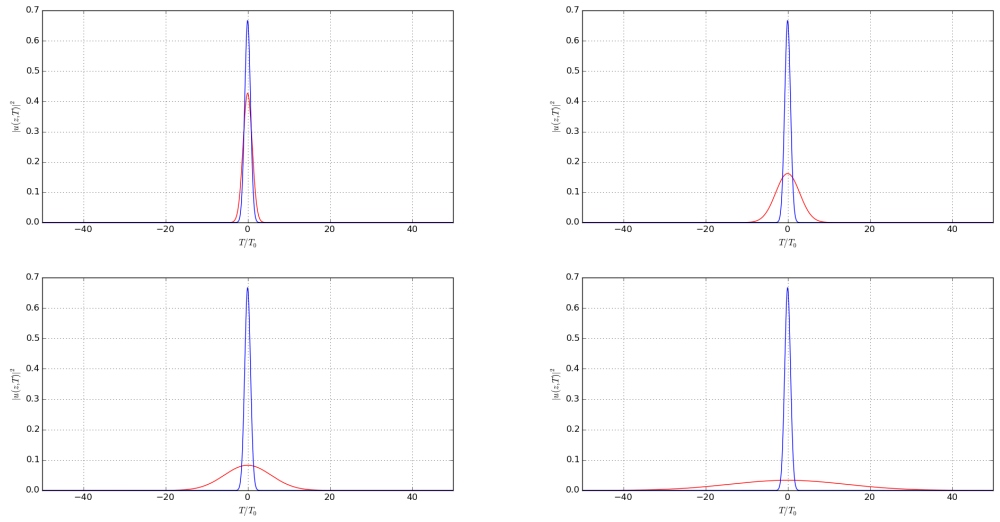


Figura 3.6: Pulso gaussiano y pulso en segundo régimen de dispersión y no linealidad para nz=30, nz=100, nz=200 y nz=500.

Comparamos de nuevo los resultados con el programa NLSE a 200 metros del origen, correspondiente a la segunda imagen, viendo que ambas se ajustan completamente, figura 3.6.

Para estos valores se puede comprobar que la dispersión es importante desde las primeras iteraciones, primeras distancias de la fibra óptica, y sin embargo la no linealidad no afecta hasta distancias próximas a los 50km. Una manera efectiva de comprobar este hecho es medir la anchura del pulso para una distancia dada. En ese caso, existe una relación entre la anchura del pulso origen, T0 y la del pulso ensanchado, T1, para la propagación de un pulso gaussiano en ausencia de no linealidad, suposición que podemos hacer para distancias menores de esos 50km. $U(z,T) = \sqrt{P_0}\frac{T_0}{\sqrt{T_0^2 - j\beta_2 z}} exp\left(-\frac{T^2}{2(T_0^2 - j\beta_2 z)}\right)$

Esta ecuación se corresponde con la del pulso original una cierta distancia

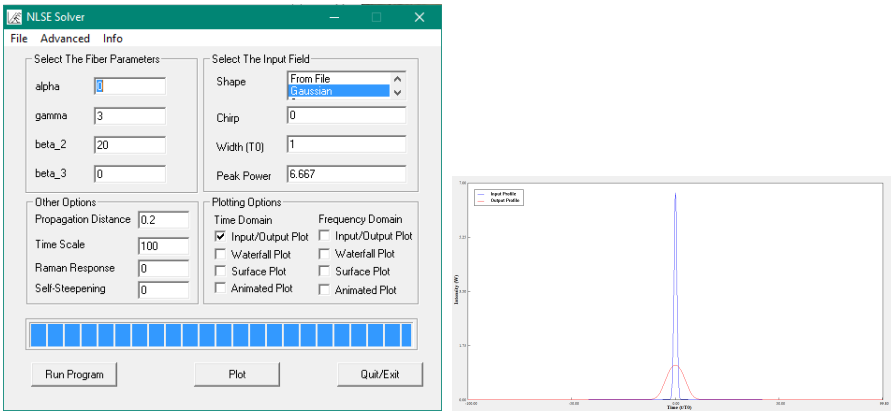


Figura 3.7: Pulso gaussiano y pulso en segundo régimen de dispersión y no linealidad para nz=100 (200 metros), a través del programa NLSE de la Universidad de Rochester.

z después de comenzar la propagación a través de la fibra. En este caso el ancho a intensidad mitad ha variado respecto al original en estos términos:

$$T_1(z) = T_0\sqrt{1+(z/L_D)^2};\quad L_D = \frac{T_0^2}{|\beta_2|} = 1/20 = 0{,}05$$

$$T_1(60m) = 1ps\sqrt{1+(0{,}06km/0{,}05km)^2} = \sqrt{2{,}2} \approx 1{,}4832ps$$

$$T_{FWHM-1}(50m) = 2{,}4696 \pm 41{,}4e(-5)ps$$

$$T_(200m) = 1ps\sqrt{1+(0{,}2km/0{,}05km)^2} = \sqrt{17} \approx 4{,}1231ps$$

$$T_{FWHM-1}(200m) = 6{,}865 \pm 6{,}96e(-6)ps$$

$$T_1(400m) = 1ps\sqrt{1+(0{,}4km/0{,}05km)^2} = \sqrt{65} \approx 8{,}0623$$

$$T_{FWHM-1}(400m) = 13{,}4237 \pm 1{,}77e(-6)ps$$

Para llevar a cabo las mediciones del pulso se utilizó un editor de imágenes con recuento de coordenadas en píxeles. Dado que la gráfica viene escalada tanto temporalmente como en amplitud, hallar los puntos de amplitud mitad y de corte con la envolvente del pulso es cuestión de manejar los cursores del programa. El valor obtenido en la fórmula T_1 debe multiplicarse por 1.665 para obtener el ancho del pulso a amplitud mitad, T_{FWHM}.

Para calcular los errores relativos se ha procedido del mismo modo que para la comparación de la tolerancia en la función ssf(): se resta la medida conseguida de la gráfica a la calculada a través de la ecuación, se halla el cuadrado de la norma dos de la resta y se divide por la norma dos del resultado obtenido en la ecuación. Los valores de error relativo obtenidos son muy pequeños porque los anchos de las gráficas se ajustan considerablemente a los de la ecuación, lo que demuestra la precisión que se consigue con el método Split-Step de Fourier simetrizado y que para estas longitudes las no linealidades pueden ser despreciadas. Un ejemplo del empleo del programa se presenta en la figura 3.7.

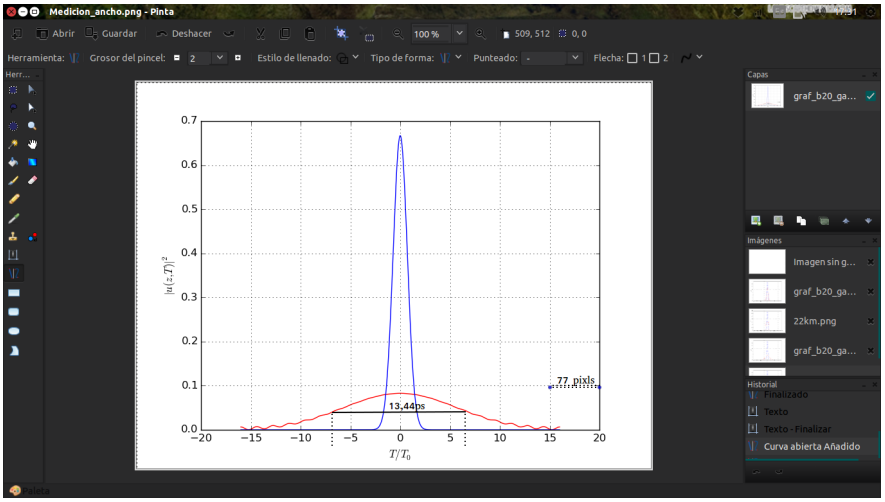


Figura 3.8: Medición para nz=200 (400m) con programa Pinta de Ubuntu.

- Si $L \ll L_D$ y $L \approx L_{NL}$, las dispersiones son insignificantes en comparación con los efectos de no linealidad. De este modo la SPM es la responsable del ensanchamiento del pulso con respecto a la GVD.

$$\frac{L_D}{L_{NL}} = \frac{\gamma P_0 T_0^2}{|\beta_2|} \gg 1$$

Aplicando la misma fórmula que en el caso anterior, fijando el ancho del pulso T0 en 100ps, que son los obtenidos en el primer régimen, obtenemos una P0 que debe ser mucho mayor a 0.667 mW. La suponemos del orden de 667mW para la realización de las simulaciones.

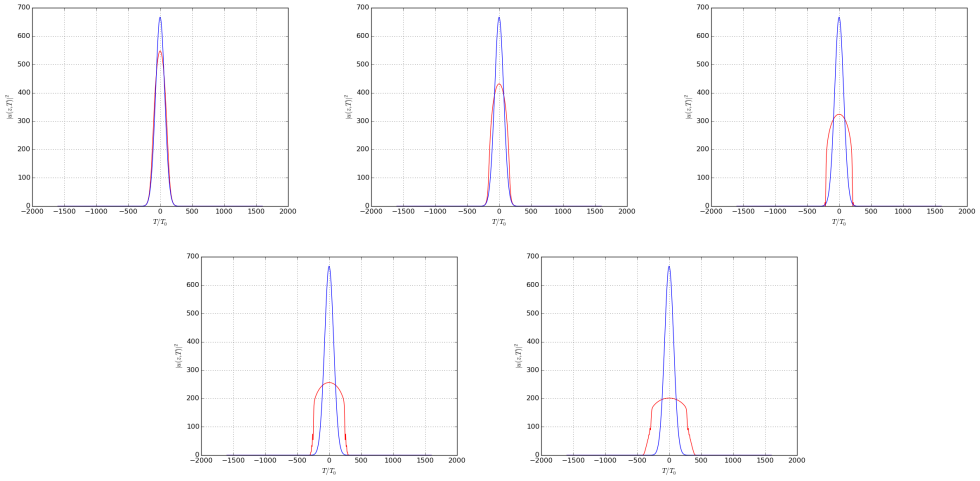


Figura 3.9: Pulso gaussiano y pulso en tercer régimen de dispersión y no linealidad para dz=0.002 y nz= 4000, nz=7000, nz=11000, nz=15000 y nz=20000.

La comparación con el programa de la Universidad de Rochester a 22km del origen vuelve a coincidir con la obtenida en la simulación de la función, figura 3.9.

Para poder representar bien el pulso se ha tenido que jugar con la escala de tiempos, ya que a grandes escalas aparece ruido creciente en la parte superior, como puede comprobarse en la imagen adjunta; y a pequeña escala, cercana a la anchura del pulso, los márgenes se comportan como interfaces que producen ondas estacionarias acumulando gran cantidad de ruido a medida que se avanza en la propagación. Es por eso, que la imagen resultante en el programa NLSE se encuentra representada a una escala de tiempos menor que la realizada en las imágenes procedentes de la simulación.

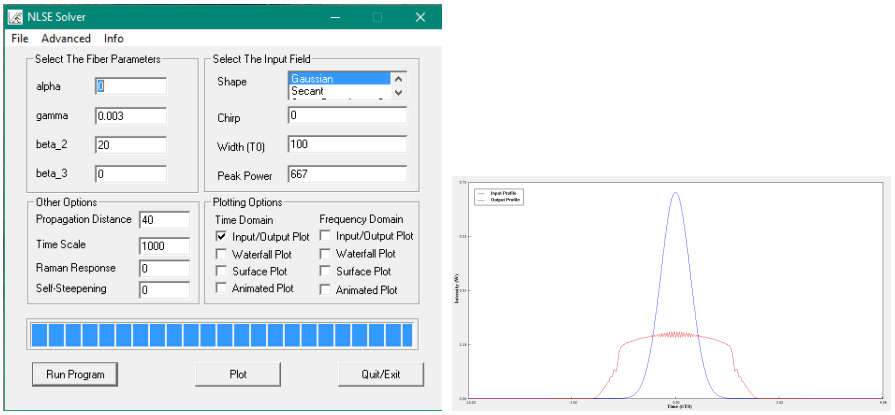


Figura 3.10: Pulso gaussiano y pulso en primer régimen de dispersión y no linealidad para nz=20000, 40km, a través del programa NLSE de la Universidad de Rochester.

El ensanchamiento del pulso es debido a las no linealidades, pero no para bajas intensidades, donde casi es inexistente. La expansión temporal se manifiesta a valores medios con una cierta verticalidad donde, para un mismo tiempo, los parciales se igualan. Aparecen ciertos lóbulos secundarios muy tenues a ambos márgenes que van acentuándose a medida que se propaga la onda a través de la fibra. Sin embargo, tanto el ensanchamiento como la aparición de lóbulos secundarios tienen una progresión muy lenta como puede comprobarse en las imágenes adjuntas. Este fenómeno es denominado rotura de la onda óptica. Se debe a que el gran número de barridos de frecuencias impuestas por la SPM inducida provoca que los efectos dispersivos incluso débiles afecten a la forma de la onda. Para dispersión normal se traduce en la expansión de la onda hasta convertirse en rectangular. El GVD inducido, por contra, tiene el efecto de producir una buena estructura cerca de los bordes.

El origen físico de las oscilaciones temporales cerca de los bordes del pulso está relacionado con la ruptura de onda óptica, donde GVD y SPM imponen

barridos de frecuencias en el pulso a medida que viaja por la fibra. Sin embargo, el chirp inducido por la GVD es lineal con el tiempo, mientras que el inducido por SPM no lo es a lo largo del pulso. Debido a la naturaleza no lineal de la composición del chirp, diferentes partes del pulso se propagan a diferentes velocidades. Las regiones iniciales y finales del pulso contienen luz a dos frecuencias diferentes que interfieren. Las oscilaciones cercanas a los bordes del pulso son el resultado de estas interferencias.

Respecto al ensanchamiento desigual producido en el pulso por el efecto cambinado de la SPM y de la GVD, provoca que la anchura a intensidad media FWHM ya no sea una medida objetiva del ensanchamiento real, ya que la forma de onda deja de ser gaussiana. En este caso, una alternativa es calcular el factor de ensanchamiento, `Broadening Factor`, a lo largo de la propagación del pulso en la fibra óptica. Para el cálculo de dicho factor se han utilizado tradicionalmente varias aproximaciones de la que vamos a hacernos eco en una gráfica comparativa en función de la longitud de la fibra. Para ello se implementarán las distintas soluciones analíticas propuestas sirviéndonos del archivo agrawal_fig3_1_ssf.py. Las cuatro soluciones propuestas parten del valor RMS de la anchura del pulso inicial σ_0 calculado como su valor cuadrático medio. También puede ser calculado dicho factor a través del valor cuadrático medio en cualquier longitud z a lo largo de la fibra. Por otro lado, distintas aproximaciones son simplificaciones más o menos factibles en función de las características analizadas. Para este caso, se hallarán en el tercer régimen de propagación. Se han tomado los datos dados en la página 65 del libro de referencia [1], con un ancho de pulso a caída 1/e, $T0 > 100ps$ y una potencia P0=1W. En realidad se han tomado los 100ps exactos para una Potencia P0=665mW. Estos datos nos arrojan

una longitud de dispersión de 500km para una longitud de no linealidad de 500m, bastante por debajo de los 80km que hemos tomado como longitud máxima de la línea. Se han realizado cuarenta y una medidas con una distancia de separación entre ellas de 2km.

$$\sigma_0 = \sqrt{\langle T_0^2\rangle - \langle T_0\rangle^2}; \quad \langle T_0^2\rangle = \frac{\int_{-\infty}^{\infty}|U(0,T)|^2t^2dt}{\int_{-\infty}^{\infty}|U(0,T)|^2dt}; \quad \langle T_0\rangle^2 = \left[\frac{\int_{-\infty}^{\infty}t|U(0,T)|^2dt}{\int_{-\infty}^{\infty}|U(0,T)|^2dt}\right]^2$$

$$\sigma_c = \sqrt{\langle T^2\rangle - \langle T\rangle^2}; \quad \langle T^2\rangle = \frac{\int_{-\infty}^{\infty}|U(z,T)|^2t^2dt}{\int_{-\infty}^{\infty}|U(z,T)|^2dt}; \quad \langle T\rangle^2 = \left[\frac{\int_{-\infty}^{\infty}t|U(z,T)|^2dt}{\int_{-\infty}^{\infty}|U(z,T)|^2dt}\right]^2$$

$$\sigma_{simp} = \sqrt{\sigma_0^2 + \tfrac{1}{2}\gamma\beta_2 S_p z^2}; \quad S_p = \tfrac{1}{W}\int_{-\infty}^{\infty}|U(0,T)|^4dt; \quad W = \int_{-\infty}^{\infty}|U(z,T)|^2dt$$

$$\frac{\sigma_{\phi max}}{\sigma_0} = \sqrt{1 + \sqrt{2}\phi_{max}\frac{L}{L_D} + \left(1 + \frac{4}{3\sqrt{3}}\phi_{max}^2\right)\frac{L^2}{L_D^2}}; \quad \phi_{max} = \gamma P_0 z; \quad L_D = \frac{T_{zero}^2}{|\beta_2|}$$

$$\frac{\sigma_{chirp}}{\sigma_0} = \sqrt{\left(1 + \frac{C\beta_2 z}{2\sigma_0^2}\right)^2 + \left(\frac{\beta_2 z}{2\sigma_0^2}\right)^2 + (1+C^2)^2\frac{1}{2}\left(\frac{\beta_3 z}{4\sigma_0^3}\right)^2}$$

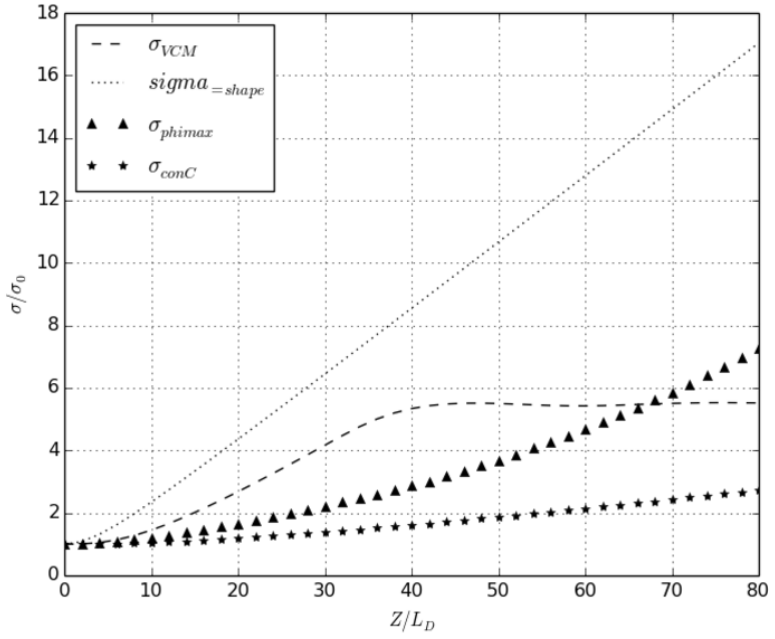


Figura 3.11: Factor de ensanchamiento del pulso en función de la longitud normalizada con la longitud de dispersión de la fibra.

Para realizar las integrales, teniendo en cuenta que nos encontramos con señales discretas, con un número nt de muestras, se calculan las sumas sucesivas de los valores obtenidos en cada una de ellas. De esta forma se obtiene la energía total del pulso inicial que no varía a lo largo de toda la longitud de la fibra óptica. La longitud L es igual a z y es calculada en la función como la multiplicación del número de saltos por el tamaño de cada salto, L=z=dz*nz.

Dado que se deben llamar a las señales en varias ocasiones, es conveniente realizar los cálculos en la gpu en paralelo a través de los paquetes PyCUDA y reikna. Sin embargo, otros no precisan utilizar las señales entrantes, ya que sólo operan con números de formato `float`. En estos casos, no es necesario llamar al kernel para realizarlas. La paralelización de la función se lleva a cabo para las sumas parciales correspondientes a la integración de la señal para los tres primeros términos. El método utilizado es un `reductionKernel`, en el que se introducen dos vectores y se calcula el cuadrado su multiplicación elemento a elemento, para luego realizar sumas sucesivas de todos ellos. Cuando sólo sea necesario realizar la suma de la potencia de un vector, el otro se introducirá como un vector de valores uno, realizado por medio de la función `ones` de `numpy`.

En el cálculo de la segunda y tercera fórmula, se multiplica el vector de salida al cuadrado con la línea de tiempos, para lo que se debe entrar una nueva línea de tiempos que sea la raíz cuadrada de la original considerándolas como vectores de números complejos. El resultado de todos los cálculos es representado en la gráfica adjunta que enfrenta el factor de ensanchamiento en función de la longitud z normalizada con la longitud de dispersión L_D.

En esta gráfica se puede observar que cada tipo de método empleado para el

cálculo del factor de ensanchamiento del pulso, arroja valores diferentes al resto. Sólo los pertenecientes al de la fase máxima y al que describe la forma simplificada a través de la Sp, parecen coincidir hasta los 16km, mientras que los correspondientes al valor cuadrático medio y el que incluye el coeficiente de shirp C y la β_3, lo hacen hasta los 10km aproximadamente. Este último se mantiene constante, con un ligero ascenso prácticamente inapreciable hasta los cuarenta kilómetros. Teniendo en cuenta que nos encontramos en tercer régimen de propagación, donde la L_D debe ser mucho mayor que la z, los valores comienzan a hacerse mucho más imprecisos a partir de un cierto límite donde comienzan a separarse de manera más acusada.

Puede comprobarse cómo el crecimiento del factor de ensanche deja de ser paulatino para el valor RMS cuando se alcanzan los 35km, que suponen el 7 % de la longitud de dispersión. Para valores mayores el factor se vuelve constante dado que se alcanza la ruptura óptica del pulso. En cambio, los resultados obtenidos para el factor calculado con la fase máxima y el que utiliza el C y el TOD para su cálculo, están relacionados por el cuadrado. Es decir, el que hemos llamado σ_{phimax} es el cuadrado del σ_{conC} en ausencia de C y TOD, como se han calculado.

En este caso, la expresión resultante se corresponde con la relación entre T1 y T0, o la anchura del pulso en caída 1/e al final de la fibra y al principio. Esto hace suponer que la expresión correcta para el factor de ensanche de la última expresión sería esa misma pero sin la raíz cuadrada, en cuyo caso los datos resultantes se ajustarían a los obtenidos mediante la de la fase máxima.

$$\frac{\sigma_{\phi_max}}{\sigma_0} = \left(\frac{T_1}{T_0}\right)^2$$

Lo mismo sucede para el valor obtenido con la fase máxima y el relativo

a la RMS del ancho temporal, también relacionados mediante el cuadrado según los resultados de la gráfica. Esto hace suponer de nuevo que el factor obtenido con la fase máxima sería correcto si le quitamos la raíz cuadrada, y en conjunto la relación entre los tres parámetros quedaría de la siguiente manera:

$$F_{RMS} = F^2_{\phi_{max}} = F^4_{chirp}$$

Con estas premisas, se repetirá el mismo ejercicio para una longitud máxima de la fibra óptica que sea un 7 % de la longitud de dispersión, es decir, hasta los 35km y eliminando la ráiz cuadrada de la expresión con fase máxima, además de elevar al cuadrado la relacionada con el chirp y el TOD.

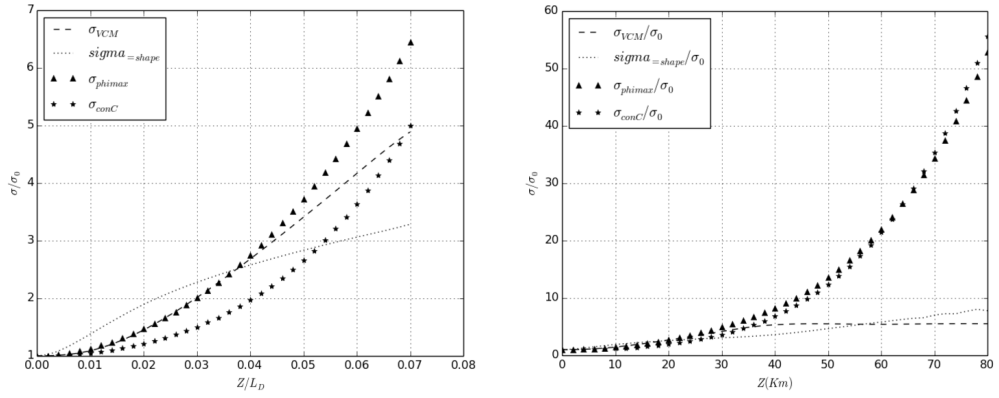


Figura 3.12: Factores de ensanche con las expresiones modificadas.

- Si $L_D, L_{NL} \ll L$, ambos efectos influyen conjuntamente a lo largo de la propagación del pulso y de forma muy diferente a como lo hacen cuando impera uno sobre otro. En este caso, si $\beta_2 \ll 0$, trabajando en régimen anómalo, puede servir para crear solitones; en dispersión normal, se puede utilizar para compresión del pulso.

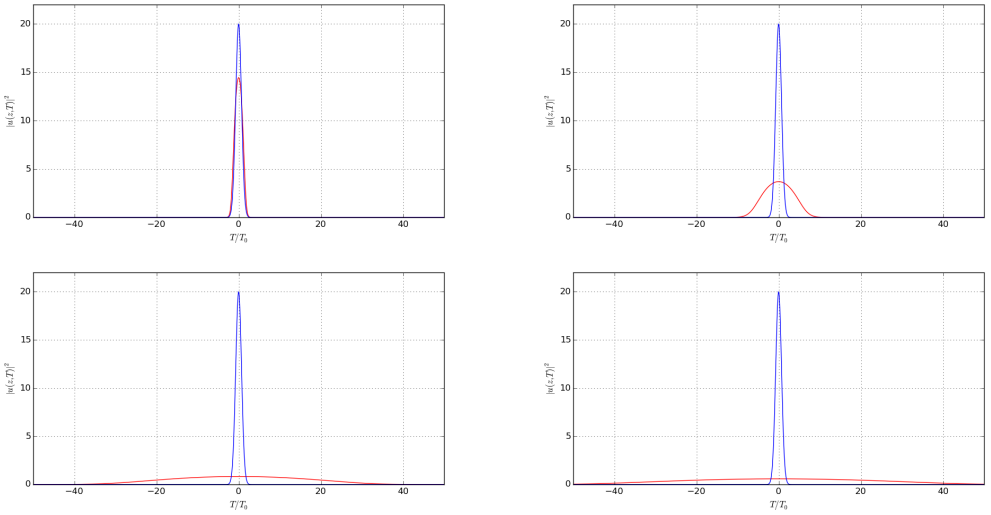


Figura 3.13: Pulso gaussiano y pulso en cuarto régimen de dispersión y no linealidad para dz=0.002, T0=1ps, P0=20W y nz=10, nz=60, nz=250, nz=350

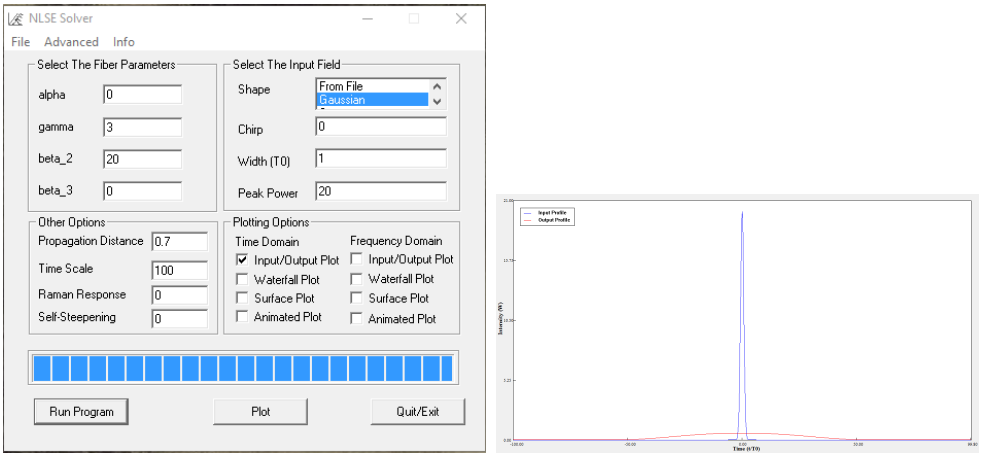


Figura 3.14: Pulso gaussiano en primer régimen con z=200, programa NLSE.

En este caso, para unas longitudes de dispersión y de no linealidad de unos 50m, se tienen una anchura de pulso T0 y una Potencia pico iguales a:

$$\lambda = 1{,}55\mu m; \quad |\beta_2| \approx 20ps^2/km; \quad \gamma \approx 3W^{-1}km^{-1}$$

$$L_D = 0{,}05km = \frac{T_0^2}{20ps^2/Km}$$

$$T_0 = \sqrt{0{,}05Km * 20ps^2/Km} = 1ps$$

$$L_{NL} = 0{,}05Km = \frac{1}{3W^{-1}Km^{-1}P_0}$$

$$P_0 = \frac{1}{0{,}05Km * 3(KmW)^{-1}} = 6{,}67W$$

Para garantizar dicho régimen y que los efectos de no linealidad se distingan suficientemente, se ha elegido una potencia pico P0 de 20W. A 20m ya el pulso comienza a ensancharse más en las potencias superiores con respecto a la base por efecto de la no linealidad, si bien este efecto no es tan acusado como en el régimen anterior puesto que la dispersión tiende a compensarla. Teniendo en cuenta esta interacción entre ambos efectos, se puede llegar a conseguir un pulso que mantenga la forma y la amplitud en su propagación a lo largo de la fibra. Sin embargo, para llevar a cabo un solitón, no es efectivo trabajar con pulsos gaussianos, ya que debe cumplir una serie de propiedades que estos no cumplen. Más adelante será abordado este tema con su respectiva práctica de propagación de pulso sechiperbólico a lo largo de una fibra óptica dispersiva y no lineal, para lo que nos serviremos de esta misma función de resolución de la ec. de Schörindger.

Cuando las longitudes de onda se encuentran muy próximas a la longitud de onda de dispersión cero, del orden del nanómetro, se hace necesario incluir el término β_3 denominado dispersión de tercer orden, TOD. En estos

casos la β_2 es igual a cero, sin embargo, en determinadas ocasiones, cuando el pulso es ultracorto, con una anchura T0 menor a 1ps, habrá que incluir ambos en la ecuación de Schrödinger dado que el parámetro $\frac{\Delta\omega}{\omega_0}$ no es lo suficientemente pequeño como para justificar la truncadura de la expansión de Taylor después de β_2. Para obtener la amplitud del pulso considerando estos dos términos, despreciamos las no linealidades en la ecuación diferencial:

$$\frac{\partial U}{\partial z} = -j\frac{\beta_2}{2}\frac{\partial^2 U}{\partial T^2} + \frac{\beta_3}{6}\frac{\partial^3 U}{\partial T^3}$$

Para trabajar con estos parámetros, la librería PyOFTK realiza un ejemplo práctico similar al del caso anterior denominado `agrawal_fig3_6_ssf.py` que trata de simular la figura 3.6 del Agrawal, con una longitud de la fibra de $z = 5L'_D$. En este archivo se llevan a cabo dos llamadas a la función ssf, la primera con la β_2 igual a cero, situación que ocurre, como ya se comentó, cuando la longitud de onda es la de dispersión cero; y la segunda, con ambos coeficientes de dispersión a uno.

En el primero de los casos existe un ligero desplazamiento del pulso hacia el margen derecho con aparición de pulsos secundarios a niveles superiores debido a que se está trabajando en régimen normal, si se hiciera en régimen anómalo los pulsos secundarios aparecerían de manera descendente hacia el margen izquierdo. En el segundo de los casos juega un papel preponderante el término L'_D, que se corresponde con la longitud de dispersión asociado con la dispersión de tercer orden, $L'_D = T_0^3/|\beta_3|$ en función de la GVD aplicada. Se considera un $L'_D = L_D$, lo que implica que $\beta_2 = \beta_3/T_0$. A medida que aumentamos el valor de la GVD, la L'_D se hace más pequeña en relación a la L_D, lo que conlleva que el pulso se aproxime más a una distribución gaussiana. En este caso, la medida del ancho del pulso conviene hacerla

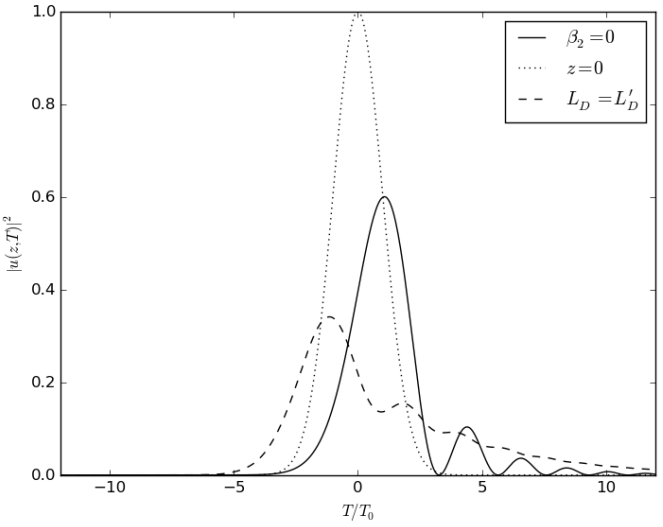


Figura 3.15: Pulso con dispersión de tercer orden.

utilizando el valor RMS del ancho calculado a partir de la tercera expresión vista en el apartado anterior donde se incluyen los efectos de la GVD, de la TOD y del chirp C.

3.1.3. Búsqueda de un solitón utilizando la ecuación escalar de Schrödinger

Como un ejemplo práctico de la utilización del método Split-Step para encontrar las soluciones de la ecuación no lineal de Schrödinger, se puede formalizar una función que busque un pulso no variable a lo largo del eje de propagación z. Para ello se deben realizar algunas consideraciones previas que acoten el problema. Para que este tipo de ondas se produzca la GVD debe ser negativa, es decir, debe trabajar en régimen anómalo de modo que se compense con la SPM producida por la constante γ positiva. Los efectos de no linealidad se compensan con los dispersivos para que la propagación del pulso ultracorto a largas distancias y

alta velocidad se produzca de igual manera que lo haría en un medio lineal, no dispersivo.

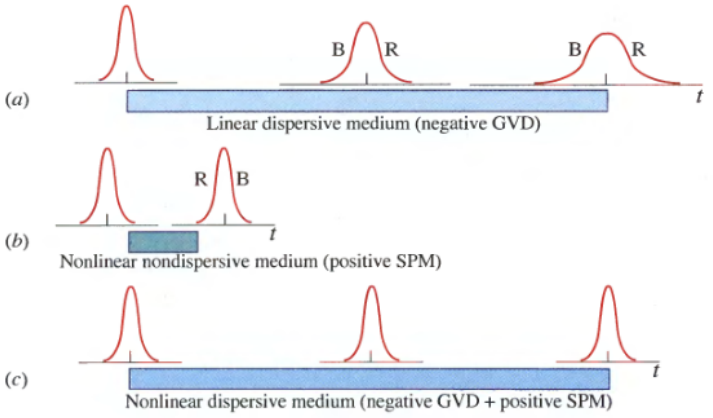


Figura 3.16: Propagación de pulsos ópticos invariantes en forma e intensidad. [7]

No se pudo dar una explicación física a este fenómeno hasta que no se desarrolló el método de scattering inverso para resolución de ecuaciones diferenciales, que parte de encontrar unas condiciones que expliquen nuestro problema para darles una solución satisfactoria. Para ello partimos de la ecuación propuesta, a la que realizamos una serie de cambios de variables [3]:

$$N = \sqrt{\gamma P_0 T_0^2/|\beta_2|}; \quad \zeta = \frac{z}{L_D} = \frac{z|\beta_2|}{T_0^2}; \quad \tau = \frac{T}{T_0}; \quad u = N\frac{A}{\sqrt{P_0}} = \sqrt{\frac{\gamma T_0^2}{|\beta_2|}}A$$

Estas expresiones las sustituimos en la ecuación de Schrödinger

$$\frac{\partial A}{\partial z} = -j\frac{\beta_2}{2}\frac{\partial^2 A}{\partial T^2} + j\gamma|A|^2A \qquad \frac{\partial u}{\partial \zeta} = \frac{j}{2}\frac{\partial^2 u}{\partial \tau^2} - j|u|^2u$$

Esta ecuación puede ser expuesta según el teorema de scattering en dos nuevas expresiones con un autovalor η común para ambas y dos amplitudes v_1 y v_2 de

u(ζ, τ):

$$\frac{\partial v_1}{\partial \tau} - juv_1 = -j\eta v_1 \quad \frac{\partial v_2}{\partial \tau} - ju^* v_2 = j\eta v_2$$

Por el método de scattering indirecto se encuentra la solución $u(\zeta, \tau)$ tras resolver una complicada integral. Si sólo existe un autovalor , N=1, el solitón es fundamental y su solución viene dada por:

$$u(\zeta, \tau) = sech(\tau) exp\left(\frac{j\zeta}{2}\right)$$

Si utilizamos esta expresión en el archivo ssf.py para encontrar unas condiciones donde el pulso no varíe, habremos dado con el solitón predicho. Para ello, en primer lugar se deberán cambiar de nuevo las variables a las que gobierna la función:

$$N = 1 = \sqrt{\gamma P_0 T_0^2 / |\beta_2|}$$

$$|\beta_2| = \gamma P_0 T_0^2; \quad \zeta = \frac{z|\beta_2|}{T_0^2} = z\gamma P_0; \quad \tau = \frac{T}{T_0}$$

$$u(z, T) = sech(\frac{T}{T_0}) exp\left(\frac{jz\gamma P_0}{2}\right)$$

Condición indispensable para que los dos factores se compensen es que las longitudes de dispersión y de no linealidad coincidan:

$$L_D = \frac{T_0^2}{|\beta_2|} = \frac{1}{\gamma P_0} = L_{NL}$$

Para un pulso de sech la relación entre el ancho a intensidad mitad FWHM y a 1/e (T0) es de 1.76 en lugar de los 1.665 correspondientes al pulso gaussiano. En este caso, se puede deducir que la potencia a imponer en el pulso de entrada sea

relación entre la GVD y la SPM a través de la γ:

$$P_0 = \frac{|\beta_2|}{\gamma T_0^2} \approx \frac{3{,}11|\beta_2|}{\gamma T_{FWHM}^2}$$

Posteriormente se habrá de realizar un bucle doble con dos while, el primero indicará el valor de la β_2 en régimen anómalo para las condiciones del cuarto tipo de propagación visto en el apartado precedente, donde la distancia de dispersión y de no linealidad son mucho menores a la longitud de la fibra, siempre bajo unas mismas condiciones de propagación, longitud de onda y ancho a intensidad media del pulso. Se hará variar tanto la GVD como la P0 para garantizar que se trabaje en condición de solitón. Por otra parte, para garantizar que la L sea mucho mayor a la longitud de dispersión y de no linealidad, que recordemos deben coincidir, se hará variar el salto espacial nz para que la z sea del orden de diez veces mayor a la longitud de no linealidad. Esto se puede conseguir teniendo en cuenta que calculamos la longitud total z a partir de la multiplicación del tamaño de salto por el número de saltos y que ambos parámetros son utilizados tanto para la generación del pulso de secante hiperbólica y del desarrollo del método split-step de Fourier simetrizado. De este modo, conforme se avanza en las iteraciones para una misma constante de dispersión, la diferencia entre la longitud de la fibra total, que se aumenta en cada iteración, y la longitud de dispersión o de no linealidad, que permanecen fijas, aumenta paulatinamente:

$$\lambda = 1{,}55nm; \quad \gamma = 3W^{-1}Km^{-1}; \quad T_{FWHM} = 1{,}76ps$$

$$P_0 \approx \frac{|\beta_2|}{3}W; \quad dz = 0{,}002Km; \quad nz = \frac{10}{P_0\gamma dz} = \frac{5000}{|\beta_2|}$$

Aunque la fft realizada en la cpu con la librería numpy hace los cálculos de manera independiente al número de muestras de la señal de entrada sin que

```
from numpy import *
 import numpy as np
 import numexpr as ne
 from scipy import *
 from pylab import *
 from time import *
 import os
 import PyOFTK
 T = 43.0


 nt = pow(2,13)
 dz = 0.002
 gamma = 3.0
 T0=0.0
 FWHM = 1*2*arccosh(sqrt(2))


 oo = 21.0
 dt = T/nt
 t = linspace(-T/2.0, T/2.0, nt)
```

```
while oo <= 40.0:
   betap=([0.0,0.0,-oo])
   P0 = pow(2*arccosh(sqrt(2)),2)*oo/(gamma*pow(FWHM,2))
   aa = 10/(P0*gamma*dz)
   bb = aa
   os.mkdir('gamma_3.0_T0_1ps/beta' + str(oo) + 'P0' + str(P0))
   while aa <= 100*bb:
      u_ini_x = PyOFTK.sechPulse(t, FWHM, T0, P0, gamma, dz, aa)
      u_out_x = PyOFTK.ssf(u_ini_x, dt, dz, aa, 0.0, betap, gamma, 10,
        1e-5, phiNLOut = False, down1 = True)
      plot(t, pow(abs(u_out_x),2), 'r', t, pow(abs(u_ini_x),2), 'b')
      ylabel("|u(z,T)|^2")
      xlabel("T/T_0")
      grid(True)
      savefig( 'gamma_3.0_T0_1ps/beta' + str(oo) +
        'P0' + str(P0) + '/figura_' + str(aa) + '.png')
      clf()
      aa += 10*bb
   oo += 1
```

suponga una pérdida de precisión en el resultado, en la fft realizada en la gpu a través de la librería reikna, que recordemos sustituye a la librería clásica pyfft de realización de transformadas de Fourier con python, sí que supone una merma importante en la precisión de los datos cuando el número de muestras introducidas no es potencia de dos. En este caso se varía el tipo de algoritmo empleado utilizando el de Bluestein, también llamado de la transformada z con chirp, mucho más ruidoso y de menor precisión, que trata el cálculo de la DFT como si fuese una convolución. Por tanto, se ha decidido modificar el número de muestras de entrada hasta 2^{16} y adaptar el rango de la línea de tiempos para hacerla coincidir con 128 THz ($\lambda = 2{,}34\mu m$) de modo que el número de muestras por salto sea exacto. A continuación se presentarán los solitones de primer rango, N=1, a lo largo de 1000km de fibra óptica para una GVD igual a $-1ps^2/Km$ con una potencia de 0.33W: Se puede observar cómo el pulso permanece constante a lo

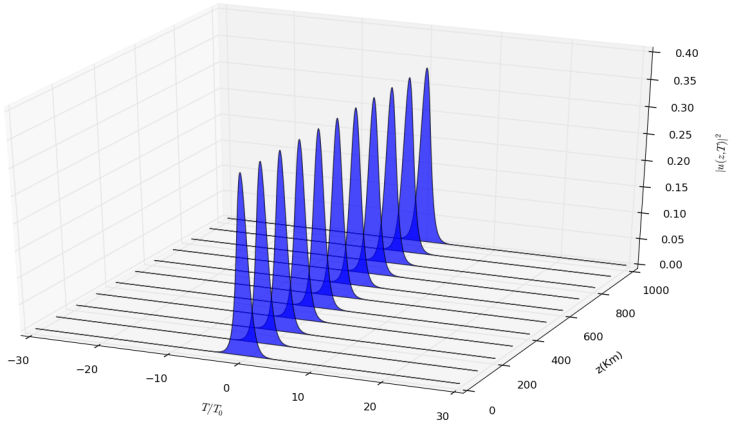


Figura 3.17: solitón para una $\beta_2 = -1$ y una P0=330mW.

largo de la longitud de la fibra sin sufrir variación de potencia o de forma. Esto es debido a que hemos supuesto un pulso de primer orden, donde la longitud de no linealidad es igual a la longitud de dispersión. .[En] este caso se dice que el solitón es fundamental pudiendose formar en fibras ópticas a niveles de potencia disponibles desde láseres semiconductores a una relativa alta tasa de bits de 20Gb/s"[1].

Cuando hablamos de solitones de orden 2, el campo de distribución asumiendo los dos autovalores como $\zeta_1 = j/2$ y $\zeta_2 = 3j/2$, es dado por la siguiente expresión:

$$u(\zeta,\tau) = \frac{4\left[cosh(3\tau) + 3exp(4j\zeta)cosh(\tau)\right]exp(j\zeta/2)}{\left[cosh(4\tau) + 4cosh(2\tau) + 3cos(4\zeta)\right]}$$

Bajo estas condiciones se puede observar cómo el cuadrado del módulo es periódico con un periodo $\zeta = \pi/2$, propiedad común a todos los solitones de orden superior. Esta perioricidad es debida a la interación desigual entre la no linealidad y la dispersión del pulso, contrario a lo que sucede en los solitones fundamentales donde ambas se compensan. Considerando la expresión de ζ en función del

desplazamiento y de la longitud de dispersión, se puede encontrar la longitud periódica que rige la propagación del pulso a lo largo de la fibra: $z = \frac{\pi}{2}\frac{T_0^2}{|\beta_2|}$. A continuación se mostrará el resultado de la simulación para la propagación de un solitón de orden dos sustituyendo los valores de $\tau = T/T_0$ y de $\zeta = \frac{z\gamma P_0}{4}$ y calculando la $P_0 = \frac{4|\beta_2|}{\gamma T_0^2}$ para dicho orden.

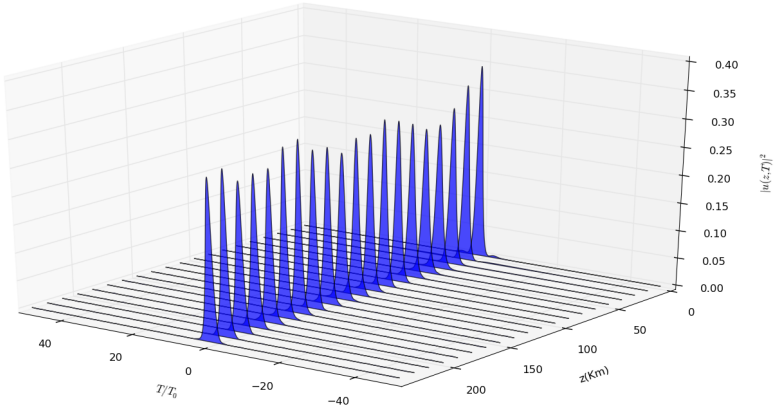


Figura 3.18: solitón de orden 2 para una $\beta_2 = -1$ y una P0=330mW.

En este solitón de orden 2 puede observarse una perioricidad en la amplitud del pulso, que no de la forma, puesto que no pierde la de la secante hiperbólica, de 90Km aproximadamente que se corresponde en grados con los $\pi/2$ de la expresión utlizada anteriormente, dado que tanto T_0 como $|\beta_2|$ son iguales a uno.

3.1.4. Análisis de tiempos y comparativa entre computación secuencial y en paralelo.

Aún cuando las llamadas que se deben hacer a los diferentes kernels y la realización de las ffts en la gpu conlleva un trasiego de datos constante entre cpu y dispositivo que merma considerablemente la velocidad de procesamiento, las altas capacidades de las gpus actuales y la paralelización de los datos hacen que merezca la pena su utilización cuando el tamaño de las muestras y la extensión del cómputo se hacen lo suficientemente importantes. Para calcular dicho factor de conveniencia del uso del paralelismo existen una serie de parámetros denominados métricas de rendimiento en arquitecturas paralelas. Estas métricas se basan principalmente en el número de núcleos que trabajan en paralelo y en el tiempo de ejecución tanto secuencial como en paralelo de los procedimientos cursados. (Computación avanzada, Tema1 de bloque1, pág 17).

Para que los análisis de tiempos sean coherentes, es necesario que el cálculo del proceso se aísle de los procedimientos en curso en la máquina. Existen tres tipos de tiempos atendiendo a este criterio:

- Tiempo Real, o tiempo de reloj. Es el tiempo total de ejecución de un programa desde que inicia su llamada hasta que se obtienen los resultados. En él juega un papel importante, a parte de la capacidad de la computadora, el número de procesos presentes en el momento de la ejecución del programa ya se encuentren en activo o bloqueados, por ejemplo, en espera de i/o.

- Tiempo de Usuario, es el tiempo de CPU empleado en la ejecución del proceso sin tener en cuenta el tiempo que pasa bloqueado y el resto de procesos ajenos al programa en cuestión.

- Tiempo del Sistema, es el tiempo de CPU en realizar llamadas a las distin-

tas funciones dentro del programa sin tener en cuenta los procedimientos internos de la máquina.

Por tanto, se puede decir que el tiempo de CPU es la suma del tiempo de usuario y el tiempo del sistema, además de que no coincide con el tiempo de reloj puesto que no tiene en cuenta los procesos ajenos al programa.

Para realizar una comparativa del tiempo empleado por las dos versiones del archivo ssf(), aquella que se realiza enteramente en secuencial y la que tiene partes ejecutadas en paralelo a través de los Kernels de CUDA y de la ejecución de la fft() de reikna, se ha utilizado una pequeña aplicación en la interfaz gráfica de usuario llamada `RunSnakeRun` con la representación de la estructura del programa en un mapa de bloques que implementa los distintos procedimientos ejecutados y que incorpora una tabla de datos con los tiempos parciales empleados en cada uno de ellos.

El archivo utilizado para tal fin es `agrawal_fig3_12_ssf.py`, una pequeña variación sobre el `agrawal_fig3_1_ssf.py` en la que se realiza un total de 10000 iteraciones del bucle for perteneciente al método split-step, con 4 aproximaciones o maxiter para el cálculo del operador no lineal. el tiempo de paso en cada iteración será de dz=0.002, lo que equivale a una longitud de 2m. Por tanto se realizará el cálculo de tiempos a 20km del origen. La constante de velocidad de dispersión de grupo, β_2, será de 1.0, el ancho a intensidad media FWHM de 2.0, la potencia pico de 1W y la constante de no linealidad, γ, de $3{,}0km^{-1}W^{-1}$. La linea de tiempos estará compuesta por nt=32768 muestras comprendidas en T=320 picosegundos y centradas en el origen, t0=0.0. El primero de los cálculos se realizó para la computación total en CPU a través de la función ssf() y arrojó los siguientes resultados:

En la imagen se puede observar un tiempo total de 244.37853 unidades de

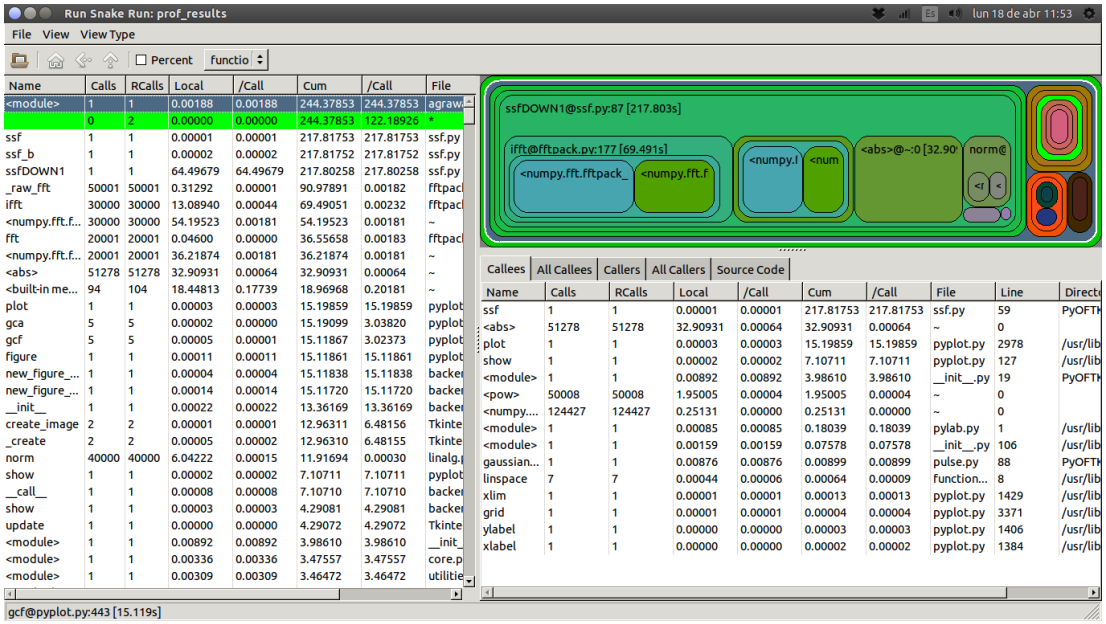


Figura 3.19: Ejecución de tiempos de cpu en programa RunSnakeRun().

reloj de las cuales, casi un 90 % pertenecen al ssf(). El resto del tiempo se lo reparte el cálculo de los módulos de las funciones resultantes para realizar los plot, los mismos plot y las representaciones gráficas de esos plot. La función de implementación del pulso gaussiano tiene una contribución casi anecdótica en el conjunto de la simulación.

La parte que utiliza paralelización será la que sufrirá algún cambio en la siguiente simulación, donde la función utilizada, también incluida en el archivo ssf.py, será ssfgpu() que hace una serie de llamadas a funciones intermedias por respetar la estructura presente en la librería PyOFTK original.

Se puede observar que la computación de ssfgpu() tarda en ejecutarse 48.909 tiempos de reloj, lo que supone un 61 % del total, frente a casi el 90 % que representa el ssf() en la ejecución secuencial. A medida que el número de iteraciones aumenta, la dependencia de la gpu se hace más palpable para la aceleración del cómputo. Para comprobar este dato, vamos a realizar unas nuevas simulaciones

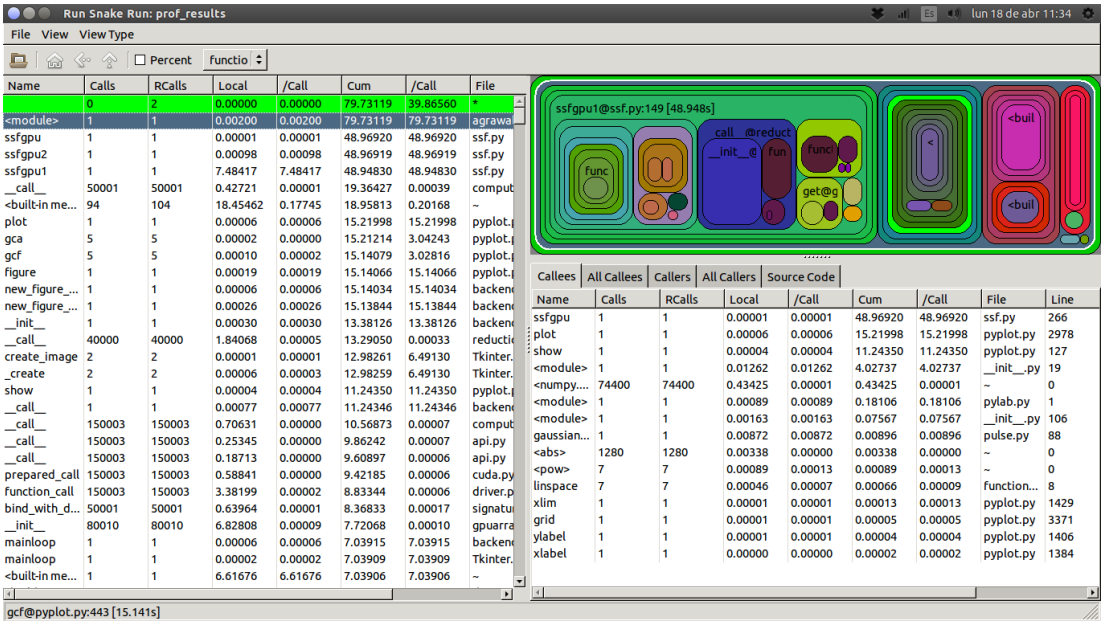


Figura 3.20: Ejecución de tiempos de cpu para agrawal_fig3_12_ssf.py con uso de ssfgpu(), resultados presentados en programa RunSnakeRun().

con 10 números de iteraciones distintos, tanto para la cpu como para la gpu, que veremos representados en una gráfica.

Nz(1000)	t_fft()	t_fftgpu()
10	217,817	48,91
20	448,288	97,034
30	663,029	143,466
40	912,384	190,075
50	1144,307	236,38
60	1349,247	285,877
70	1591,337	333,123
80	1796,31	378,193
90	2009,654	424,527
100	2259,842	479,354

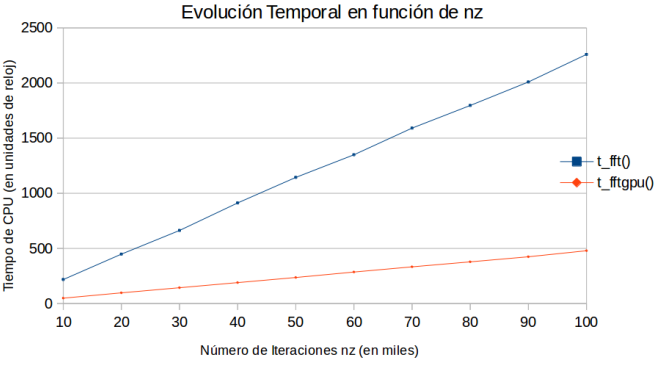


Figura 3.21: Comparativa de tiempos entre ejecución secuencial y en paralelo a través de CUDA.

La progresión entre ambos tipos de ejecución se mantiene bastante pareja a lo largo de la propagación del pulso, si bien a medida que aumentamos el número

de iteraciones la proporción de tiempo empleada por la programación secuencial es ligeramente mayor que la paralelizada. De este modo, con 10000 iteraciones la proporción entre ambas es de 4.45, mientras que para 100000 iteraciones la progresión es de 4.71.

Sin embargo, si realizamos esta misma comparación variando el número de muestras de las señales de entrada, es decir, cambiando el exponente de la variable nt de base dos de manera paulatina, podemos observar que el desempeño del método en la gpu es mucho mayor que en la cpu, de tal modo que, a partir de un número de muestras, aún cuando se produce un cambio brusco doblando los tiempos de ejecución, el crecimiento en la gpu es mucho más tenue y menor que en la cpu, donde sigue la misma progresión.

nt	T_cpu	T_gpu
256	46,357	119,987
512	54,446	120,84269
1024	69,59	125,169
2048	99,6	125,742
4096	167,148	227,523
8192	308,606	248,524
16384	581,763	249,031
32768	1138,504	257,549
65536	2410,234	258,972
131072	4966,09	310,5

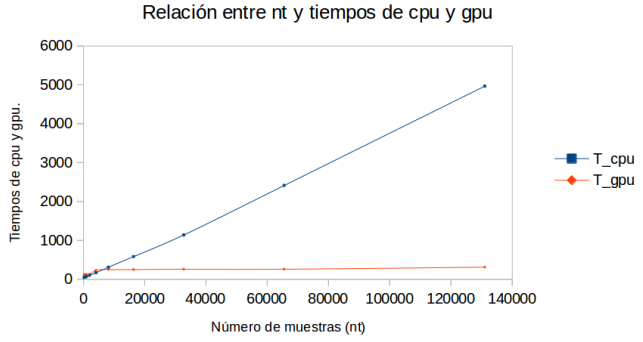


Figura 3.22: Comparativa de tiempos entre ejecución secuencial y en paralelo variando la nt.

Hasta 2^{12} priman las llamadas a los kernels y el cálculo de la fft() a través de reikna sobre el método secuencial en cpu, a la hora del cómputo de tiempos. Sin embargo, a partir de este número de muestras, donde la ejecución en la gpu sufre un incremento considerable del tiempo mientras que la cpu sigue aumentándolo con la misma pendiente, los tiempos de la gpu se estabilizan a lo largo del resto de valores de la nt, con una pendiente muy suave hasta los $nt = 2^{17}$ donde la relación

entre uno y otro viene a ser de 1:16. Por tanto, se puede concluir que a partir de un número de muestras se hace muy aconsejable el uso de la paralelización a través de llamadas a kernels en gpu y cálculo de las fft() en gpu con reikna. El trasiego de la información entre cpu y gpu se hace insignificante en relación al gran número de muestras que se han de computar secuencialmente en la cpu.

3.2. Funciones y ejemplos para el uso de las ecuaciones acopladas de Schrödinger

3.2.1. Implementación de un programa que realice el método split-step simetrizado para resolver las ecuaciones de Schrödinger acopladas.

Para desarrollar el programa que calcula la propagación de un pulso óptico en una fibra óptica dispersiva, no lineal y birrefrigente, donde se tienen en cuenta los dos ejes de polarización apareciendo dos ecuaciones diferenciales acopladas, se ha creado un nuevo archivo llamado ssfvec.py incluido en la carpeta PyOFTK. Este archivo se ha realizado a partir del programa SSPROP del mismo nombre creado para Matlab por el Laboratorio de Investigación Fotónica de la Universidad de Maryland, Estados Unidos. Este paquete se encuentra incluido en la carpeta ssprop del mismo PyOFTK, donde se ha realizado un wrapp que permite ejecutarlo desde python aún estando programado en C++. Lo curioso del caso es que nos encontramos con una archivo origen realizado en .m, mucho más sencillo de traducir a python que a C. Siguiendo este criterio, se ha llevado a cabo esta traducción de lenguaje de matlab a python del archivo en cuestión, salvo por pequeñas modificaciones que nos adelantamos a señalar. [9]

La estructura de todo el archivo origen se compone de una sola función con varios `if-else` sucesivos; sin embargo, nosotros hemos realizado funciones diferentes para cada apartado con el fin de estructurar de manera más organizada el conjunto de procesos que se deben realizar. Como se comentó en el apartado anterior, se ha omitido el vector sps de dos elementos, el primero el ángulo ψ referente a la elipticidad, y el segundo el χ, o ángulo de desviación de los ejes de simetría con respecto a los de abscisas y ordenadas. En su lugar, se introducirán

ambos por separado en los parámetros de la función.

Por otra parte, dado que el procedimiento para construir el operador de dispersión es el mismo tanto para el coeficiente de dispersión como para el de atenuación, dado que se ha considerado este último como dependiente de la frecuencia, a diferencia del caso escalar; se ha realizado una sóla función que implementa a ambos para las dos ecuaciones diferenciales acopladas. Las dos funciones que se encargan de realizar esta operación son las siguientes:

```
def aten_disp(param, w):
   halfstep=0.0
   for ii in arange(len(param)):
      if (ii% 2 == 0): m = -1
      else: m = 1
      halfstep = halfstep - 1.0j*m*param[ii]*pow(w,ii)/factorial(ii)
   return halfstep
def halfdisp(nt, dt, dz, nz, alpha, betap):
   w = wspace(dt*nt,nt)
   alfa = aten_disp(alpha,w)
   beta = aten_disp(betap,w)
   return (alfa - beta).astype(numpy.complex64)
```

La función halfdisp es llamada para cada una de las ecuaciones diferenciales acopladas, devolviendo el exponente complejo de la exponencial que sirve como parámetros de dispersión para el método de polarización elíptica o que son utilizadas para construir cada uno de los parámetros de la matriz de dispersión en el de polarización circular. En cualquier caso, ha y hb son invariantes para ambos métodos de modo que no es necesario realizar diferenciación entre ambos.

Para la realización del operador de no linealidad también se ha creado una nueva función en la que sí se hace necesario distinguir entre el operador de no linealidad del método de polarización elíptica del utilizado en la polarización circular. En ambos casos devuelve la función calculada en el dominio de la frecuencia; es decir, la transformada de Fourier del producto entre la función de entrada y el operador resultante en cada uno de los métodos.

```
def halfnl(half, gamma, dz, u01, u11, u02, u12, metodo, chi):
```

```
valor = (-1.0j*gamma*dz/3)
if metodo == 'circular':
   valor1 = pow(abs(u01),2) + pow(abs(u11),2) + 2*pow(abs(u02),2) + ...
     ... + 2*pow(abs(u12),2)
else:
   valor1 = (1 + pow(cos(2*chi),2)/2) * (pow(abs(u01),2) pow(abs(u11),2)) + ...
     ... + (1 + pow(sin(2*chi),2)) * (pow(abs(u02),2) + pow(abs(u12),2))
return fftpack.fft(half*exp(valor*valor1))
```

El cálculo de cada uno de los parámetros se lleva a cabo a partir de las matrices estudiadas en el apartado anterior y en el anexo [B] correpondiente a la descripción de las ecuaciones de Schrödinger acopladas que caracterizan la propagación de un pulso óptico en sus dos ejes de polarización a través de una fibra óptica no lineal y dispersiva.

La función principal ssfvec1() recibe quince parámetros, los mismos que en el caso escalar por duplicado, ya que hay dos pulsos de entrada y todos los componentes para cada uno de ellos, salvo el coeficiente de no linealidad γ que es el mismo en los dos, maxiter, tol y un parámetro llamado método con dos opciones: 'circular' y 'eliptico'.

```
def ssfvec1(u0x,u0y,dt,dz,nz,alphaa,alphab, + ...
  ... + betapa,betapb,gamma,psi,chi,metodo,maxiter,tol):
```

Lo primero que se realiza en la función es calcular el número de muestras de las señales entrantes y la construcción de los operadores disersivos ha y hb (llamados por analogía con el procedimiento escalar, halfstepa y halfstepb respectivamente), para lo que se realiza la exponencial al resultado de la llamada de la función halfdisp(), ya explicada.

```
nt = len(u0x)
halfstepa = exp(halfdisp(nt, dt, dz, nz, alphaa, betapa)*dz/2)
halfstepb = exp(halfdisp(nt, dt, dz, nz, alphab, betapb)*dz/2)
u0x = u0x.astype(numpy.complex64)
u0y = u0y.astype(numpy.complex64)
```

La primera distinción entre el método circular y elíptico se realiza acto seguido, donde se contruirán las nuevas funciones transformadas para los sistemas de referencia en ambos ejes. Si el métofdo empleado es el circular también se realizarán,

utilizando para ello los parámetros de no linealidad anteriormente calculados, de los cuatro elementos de la matriz de dispersión, h11, h12, h21 y h22.

```
if metodo == 'circular':
   u0a = (u0x + 1.0j*u0y)/sqrt(2)
   u0b = (1.0j*u0x + u0y)/sqrt(2)
   h11 = ((1.0+sin(2.0*chi))*halfstepa + (1.0-sin(2.0*chi))*halfstepb)/2.0
   h12 = -1.0j*(exp(1.0j*2.0*psi)*cos(2.0*chi)*(halfstepa-halfstepb))/2.0
   h21 = 1.0j*(exp(-1.0j*2.0*psi)*cos(2.0*chi)*(halfstepa-halfstepb))/2.0
   h22 = ((1.0-sin(2.0*chi))*halfstepa + (1.0+sin(2.0*chi))*halfstepb)/2.0
else:
   u0a = ( cos(psi)*cos(chi) - 1.0j*sin(psi)*sin(chi))*u0x +...
     ...+ (sin(psi)*cos(chi) + 1.0j*cos(psi)*sin(chi))*u0y
   u0b = (-sin(psi)*cos(chi) + 1.0j*cos(psi)*sin(chi))*u0x +...
     ...+ (cos(psi)*cos(chi) + 1.0j*sin(psi)*sin(chi))*u0y
```

Las funciones resultantes para los dos ejes de polarización se pasan al dominio de la frecuencia a través de una transformada de Fourier. De las funciones transformadas se realiza una copia tal y como se hacía en el procedimiento escalar, para modificar el valor de la función en el proceso de aproximación al valor de no linealidad realizado por el segundo bucle for, sin que se modifique la original hasta el final de cada uno de los saltos.

```
uafft = fftpack.fft(u0a)
ubfft = fftpack.fft(u0b)
u1a = u0a.astype(numpy.complex64)
u1b = u0b.astype(numpy.complex64)
```

A continuación se comienza a realizar el avance a lo largo de la fibra por medio del método split-step de Fourier simetrizado. La primera llamada al operador dispersivo se realiza para cada una de las ecuaciones según el método a emplear; de este modo, si se utiliza el método circular, la función resultante en el dominio de la frecuencia se construye por la suma de las dos funciones transformadas previamente multiplicadas por los operados h11 y h12 respectiva para la polarización circular a derechas, y por h21 y h22 para la polarización circular a izquierdas. Si el método es el elíptico, halfstepa se multiplica por la función del eje rápido, uafft, y halfstepb por la del eje lento, ubfft. Los resultados se pasan de nuevo al

dominio del tiempo por medio de la transformada inversa de Fourier, con el fin de realizar la mulitplicación por el operador no lineal:

```
for iz in arange(nz):
   if metodo == 'circular':
      uahalf = fftpack.ifft(h11*uafft+h12*ubfft)
      ubhalf = fftpack.ifft(h21*uafft+h22*ubfft)
   else:
      uahalf = fftpack.ifft(halfstepa*uafft)
      ubhalf = fftpack.ifft(halfstepb*ubfft)
   uahalf = uahalf.astype(numpy.complex64)
   ubhalf = ubhalf.astype(numpy.complex64)
```

Para aproximar el valor del pulso inicial al del siguiente salto, y poder así calcular el operador no lineal, tal y como se explicó en el caso escalar, se realiza un método de aproximación sucesiva utilizando otro bucle for donde se incrementa el valor de la función de manera paulatina un cierto número de veces. Hay que compensar cada salto para que tan pronto el error relativo de la función de avance y la inicial del paso alcance una tolerancia mínima que se introduce por parámetros , se realice un break para pasar a la siguiente iteración. Si se cumplieran todas las iteraciones del bucle interior sin que se produzca un break, sería prueba de que el pulso no ha alcanzado el avance que se había estipulado, en cuyo caso el programa termina con una excepción en la que se indica que no ha habido convergencia. Las operaciones dentro del bucle de aproximaciones son las mismas que en el caso escalar, con la salvedad de que hay que repetirlas para las dos ecuaciones y las peculiaridades ya explicadas de cada uno de los dos métodos empleados. Antes de comenzar el siguiente salto, se sobreescriben las funciones transformada con las de avance.

```
   for ii in arange(maxiter):
      uva = halfnl(uahalf, gamma, dz, u0a, u1a, u0b, u1b, metodo, chi)
      uvb = halfnl(ubhalf, gamma, dz, u0b, u1b, u0a, u1a, metodo, chi)
      if metodo == 'circular':
         uafft = (h11*uva + h12*uvb)
         ubfft = (h21*uva + h22*uvb)
      else:
         uafft = halfstepa*uva
```

```
        ubfft = halfstepb*uvb
      uva = fftpack.ifft(uafft)
      uvb = fftpack.ifft(ubfft)
      error = linalg.norm(abs(uva-u1a),2.0)
      e1a = linalg.norm(abs(u1a),2.0)
      error2 = linalg.norm(abs(uvb-u1b),2.0)
      e1b = linalg.norm(abs(u1b),2.0)
      error = sqrt(pow(abs(error),2.0) + pow(abs(error2),2.0))
      e1 = sqrt(pow(abs(e1a),2.0) + pow(abs(e1b),2.0))
      error = error/e1
      u1a = uva
      u1b = uvb
      if (error <tol):
         break
   if (ii == maxiter):
      raise Exception, "Failed to converge",
   u0a = u1a
   u0b = u1b
```

Cuando se han realizado todas las iteraciones del bucle for, es decir, cuando se ha alcanzado la longitud de la fibra que se pretendía, se realiza el cambio de referencia para comprobar los resultados en el extremo final de la fibra, en función siempre del método empleado. Para ello basta con multiplicar las funciones resultantes por las matrices transpuestas de la transformación del inicio:

```
if metodo ==’circular’:
   u1x = (u1a-1.0j*u1b)/sqrt(2.0)
   u1y = (u1b-1.0j*u1a)/sqrt(2.0)
else:
   u1x = (cos(psi)*cos(chi) + 1.0j*sin(psi)*sin(chi))*u1a +...
     ...+ (-sin(psi)*cos(chi) - 1.0j*cos(psi)*sin(chi))*u1b
   u1y = (sin(psi)*cos(chi) - 1.0j*cos(psi)*sin(chi))*u1a +...
     ...+ ( cos(psi)*cos(chi) - 1.0j*sin(psi)*sin(chi))*u1b
return [u1x, u1y]
```

Los dos métodos utlizados para el cálculo de las funciones en el programa original se realizan en un if-else, repitiéndose en cada uno de ellos todo lo que tienen en común, es decir, todo el método split-step. En su lugar, se han realizado if-else especificos en aquellos sitios donde era preciso, manteniendo el resto como una sola función. De este modo son muchas las líneas de código que nos ahorramos.

3.2.2. Paralelización con uso de GPU del programa implementado para resolver las ecuaciones vectoriales de Schrödinger acopladas.

Dentro del mismo archivo ssfvec.py se ha creado una segunda función ssfvecgpu() en la que se paraleliza la función ssfvec1() con el uso de kernels de PyCUDA y de la librería Reikna de manera análoga a como se hizo con la solución de la ecuación escalar.

Ya que el método es el mismo y que el procedimiento sigue exactamente los mismos pasos que en el formato secuancial, nos vamos a limitar a explicar las variables pasadas a la memoria de la GPU o creadas directamente en ella, y los kernels implementados para ejecutar en paralelo aquella parte de código paralelizable.

Para la creación de los operadores dispersivos ha y hb se han utilizado las mismas funciones que en el modelo secuencial. Por defecto se implementa el context a true, de modo que no es necesario inicializar el driver de pycuda ni indicar que el contexto utilizado por defecto es el que trae la librería, como se indicó en la paralelización del método escalar. Se guarda en una variable local la api() de la librería Reikna indicando que estamos trabajando en CUDA, y se crea el objeto Thread de dicha librería para utlización de hilos con la que se indica el contexto a utilizar, por defecto utilizará el dispositivo disponible, y la cola en la que se guardarán los kernels en la gpu. Se llama a la aplicación FFT() de Reikna que se compila por defecto para un array relleno de unos y de tamaño igual a las variables que se van a utilizar en el procedimiento. Cada vez que se utilice esta función thr_fft se compilará la FFT con las funciones guardadas en la gpu, que a su vez recibirán el mismo nombre que en la cpu precedido por un gpu_ en los espacios de memoria indicados por el contexto thr.

```
   def ssfvecgpu(u0x, u0y, dt, dz, nz, alphaa, alphab, betapa, betapb, gamma, psi, chi,
metodo, maxiter, tol, context):
       nt = len(u0x)
       halfstepa = exp(halfdisp(nt, dt, dz, nz, alphaa, betapa))
       halfstepb = exp(halfdisp(nt, dt, dz, nz, alphab, betapb))
       u0x = u0x.astype(numpy.complex64)
       u0y = u0y.astype(numpy.complex64)
       ue = numpy.zeros((1, nt))
       if context == False:
          cuda.init()
          context = make_default_context()
       u1x=u0x
       u1y=u0y
       api = cluda.cuda_api()
       thr = api.Thread.create()
       data = numpy.ones((1, nt), dtype=numpy.complex64)
       fft = FFT(data)
       thr_fft = fft.compile(thr)
```

Se crearán variables en la gpu para los parámetros de dispersión ha y hb, los cuatro elementos de la matriz de dispersión en el método de polarización circular h11, h12, h21 y h22, los dos pulsos de entrada, los dos pulsos de entrada transformados, los dos pulsos transformados con signos cambiados, una copia de los pulsos de entrada y los transformados, para los resultados del producto entre los pulsos y los operadores dispersivos, para los pulsos en cálculos intermedios de aproximación, para los pulsos en el dominio de la frecuencia, copia de estos y por último, para el cálculo del error relativo.

Para la realización del método paralelizado se han implementado una serie de kernels en aquellas secciones del procedimiento con cálculos de los pulsos de entrada, donde puedan ser divididos en sectores de un determinado número de muestras gobernadas por hilos que trabajan a la vez en la gpu. En primer lugar para la obtención de los pulsos transformados en los ejes de simetría y sistema de referencia de la fibra birrefrigente en ambos métodos. Se realiza una copia de los pulsos transformados en el dominio del tiempo para lo que se utiliza un nuevo kernel *`complexDeepCopy()`* ya creado para tal fin en el método escalar. A continuación se presenta el resto del código con las llamadas a los distintos

```
gpu_halfstepa = thr.to_device(halfstepa)
gpu_halfstepb = thr.to_device(halfstepb)
gpu_h11 = thr.empty_like(gpu_halfstepa)
gpu_h12 = thr.empty_like(gpu_halfstepa)
gpu_h21 = thr.empty_like(gpu_halfstepa)
gpu_h22 = thr.empty_like(gpu_halfstepa)
gpu_u0x = thr.to_device(u0x)
gpu_u0y = thr.to_device(u0y)
gpu_u0a = thr.empty_like(gpu_u0x)
gpu_u0b = thr.empty_like(gpu_u0x)
gpu_u0ainv = thr.empty_like(gpu_u0x)
gpu_u0binv = thr.empty_like(gpu_u0x)
```

```
gpu_u1x = thr.to_device(u1x)
gpu_u1a = thr.empty_like(gpu_u0x)
gpu_u1b = thr.empty_like(gpu_u0x)
gpu_uahalf = thr.empty_like(gpu_u0x)
gpu_ubhalf = thr.empty_like(gpu_u0x)
gpu_uva = thr.empty_like(gpu_u0x)
gpu_u1y = thr.to_device(u1y)
gpu_uvb = thr.empty_like(gpu_u0x)
gpu_uafft = thr.empty_like(gpu_u0x)
gpu_ubfft = thr.empty_like(gpu_u0x)
gpu_uafft1 = thr.empty_like(gpu_u0x)
gpu_ubfft1 = thr.empty_like(gpu_u0x)
gpu_ue = thr.to_device(ue))
```

kernels en color azul:

```
    if metodo == 'circular':
        transKernelCirc(gpu_u0x, gpu_u0y, gpu_u0a)
        transKernelCirc(gpu_u0y, gpu_u0x, gpu_u0b)
        paramhKernel(gpu_halfstepa, gpu_halfstepb, psi, chi, gpu_h11, gpu_h12, gpu_h21,
gpu_h22)
    else:
        transKernelElip(gpu_u0x, gpu_u0y, float(psi), float(chi), gpu_u0a)
        transKernelElip(gpu_u0y, gpu_u0x, float(-psi), float(chi), gpu_u0b)
    thr_fft(gpu_uafft,gpu_u0a)
    thr_fft(gpu_ubfft,gpu_u0b)
    complexDeepCopy(gpu_u1a,gpu_u0a)
    complexDeepCopy(gpu_u1b,gpu_u0b)
```

Para el cálculo de la parte dispersiva se crean dos kernels diferentes, uno para el método circular y uno para el método elíptico. Lo mismo se hace para el cálculo de la parte no dispersiva o no lineal. Se hace uso de un reductionkernel para el cálculo del error relativo que realiza la norma dos de los dos vectores de entrada.

También se ha creado un kernel que devuelve a la salida el valor con signo cambiado del vector de números complejos de la entrada y que será usado para el cambio del pulso de salida transformado al pulso de salida en el eje de coordenadas cartesianas. Para este fin se crea un nuevo kernel en el método elíptico, ya que en el circular se aprovecha el ya implementado en la transformación inicial.

```
      for iz in arange(nz):
         if metodo == 'circular':
            halfStepCircKernel(gpu_uafft, gpu_ubfft, gpu_h11, gpu_h12, gpu_uafft1)
            halfStepCircKernel(gpu_uafft, gpu_ubfft, gpu_h21, gpu_h22, gpu_ubfft1)
         else:
            halfStepElipKernel(gpu_uafft, gpu_halfstepa, gpu_uafft1)
            halfStepElipKernel(gpu_ubfft, gpu_halfstepb, gpu_ubfft1)
         thr_fft(gpu_uahalf, gpu_uafft1, inverse=True)
         thr_fft(gpu_ubhalf, gpu_ubfft1, inverse=True)
         for ii in arange(maxiter):
            if metodo == 'circular':
               nlKernelCircular(gpu_uahalf, gpu_u0a, gpu_u1a, gpu_u0b, gpu_u1b, gpu_uva,
float(gamma), float(dz))
               nlKernelCircular(gpu_ubhalf, gpu_u0b, gpu_u1b, gpu_u0a, gpu_u1a, gpu_uvb,
float(gamma), float(dz))
            else:
               nlKernelElipt(gpu_uahalf, gpu_u0a, gpu_u1a, gpu_u0b, gpu_u1b, gpu_uva, ...+
+ ... float(gamma), float(chi), float(dz))
               nlKernelElipt(gpu_ubhalf, gpu_u0b, gpu_u1b, gpu_u0a, gpu_u1a, gpu_uvb, ...+
+ ... float(gamma), float(chi), float(dz))
            thr_fft(gpu_uva, gpu_uva)
            thr_fft(gpu_uvb, gpu_uvb)
            if metodo == 'circular':
               halfStepCircKernel(gpu_uva, gpu_uvb, gpu_h11, gpu_h12, gpu_uafft)
               halfStepCircKernel(gpu_uva, gpu_uvb, gpu_h21, gpu_h22, gpu_ubfft)
            else:
               halfStepElipKernel(gpu_uva, gpu_halfstepa, gpu_uafft)
               halfStepElipKernel(gpu_uvb, gpu_halfstepb, gpu_ubfft)
            thr_fft(gpu_uva, gpu_uafft, inverse=True)
            thr_fft(gpu_uvb, gpu_ubfft, inverse=True)
            e_inia = computeError(gpu_u1a, gpu_ue).get()
            errora = computeError(gpu_u1a, gpu_uva).get()
            e_inib = computeError(gpu_u1b, gpu_ue).get()
            errorb =computeError(gpu_u1b, gpu_uvb).get()
            error = sqrt((errora + errorb)/(e_inia + e_inib))
            complexDeepCopy(gpu_u1a, gpu_uva)
            complexDeepCopy(gpu_u1b, gpu_uvb)
            if(error <tol):
               break
            if (ii == maxiter):
               if context:
                  context.pop()
                  raise Exception, "Failed to converge",
         complexDeepCopy(gpu_u0a, gpu_u1a)
         complexDeepCopy(gpu_u0b, gpu_u1b)
      complexDeepCopyInv(gpu_u0ainv, gpu_u0a)
      complexDeepCopyInv(gpu_u0binv, gpu_u0b)
      if metodo == 'circular':
         transKernelCirc(gpu_u0a, gpu_u0binv, gpu_u1x)
         transKernelCirc(gpu_u0b, gpu_u0ainv, gpu_u1y)
```

```
else:
   transKernelElipInv(gpu_u0a, gpu_u0b, float(psi), float(chi), gpu_u1x)
   transKernelElipInv(gpu_u0b, gpu_u0a, float(-psi), float(chi), gpu_u1y)
u0x = gpu_u1x.get()
u0y = gpu_u1y.get()
if context == False:
   context.pop()
gc.collect()
return [u0x, u0y]
```

Cabe señalar que todos los kernels son semejantes a los ya implementados en el método escalar y que se reducen a la realización de operaciones aritméticas con números complejos que deben descomponerse en sus partes reales e imaginarias, ya que el lenguaje derivado del C con que se trabaja en los kernels no permite el cálculo complejo aunque sí la descomposición. Todos ellos realizan las operaciones necesarias para llevar a cabo el conjunto del programa y que ya han sido vistas en la parte secuencial. Si acaso, se puede mencionar el uso de los valores cambiados de signo de los pulsos realizados en el *complexDeepCopyInv()* para ser usados en las operaciones de conversión a pulsos de salida en coordenadas cartesianas por medio del *transKernelCirc()* dentro del método de polarización circular. Para la polarización elíptica, el cambio debe realizarse en un nuevo kernel denominado *transKernelElipInv()*.

Respecto a la utlización del ReductionKernel *computeError()* de manera reiterada para el cálculo del error relativo del avance para la convergencia del parámetro de no linealidad, es más sencillo calcular cada una de las normas aún cuando no haya diferencia de elementos pertenecientes a vectores distintos, ya que en ese caso se utiliza como vector de resta un array del mismo tamaño que aquel del que se quiere obtener la norma y rellanado de ceros. Por tanto se produce la suma sucesiva del cuadrado del elemento no cero, que es lo que se pretende:

$$e_r = \sqrt{\frac{||u1a - uva||^2 + ||u1b - uvb||^2}{||u1a||^2 + ||u1b||^2}}$$

A continuación, para completar la presentación de la función realizada en la gpu, se muestra el conjunto de kernels, un total de diez, utilizados en la misma sin pararnos a explicar con detenimiento cada uno de ellos:

-transKernelCirc Para la conversión de los pulsos de entrada y de salida en el método circular:

```
      transKernelCirc = ElementwiseKernel("pycuda::complex<float>*u0x,
pycuda::complex<float>*u0y, pycuda::complex<float>*u0",
        "
        u0[i]._M_re = (u0x[i]._M_re - u0y[i]._M_im)/sqrt(2.0);
        u0[i]._M_im = (u0x[i]._M_im + u0y[i]._M_re)/sqrt(2.0)
        ",
        "halfstep_nonlinear",
        preamble="#include <pycuda-complex.hpp>",)
```

-transKernelElip Para la conversión de los pulsos de entrada en el método elíptico

```
      transKernelElip = ElementwiseKernel("pycuda::complex<float>*u0x,
pycuda::complex<float>*u0y, float psi, float chi, pycuda::complex<float>*u0",
        "
        u0[i]._M_re = cos(psi)*cos(chi)*u0x[i]._M_re + sin(psi)*sin(chi)*u0x[i]._M_im +...
          ...+ sin(psi)*cos(chi)*u0y[i]._M_re - cos(psi)*sin(chi)*u0y[i]._M_im;
        u0[i]._M_im = cos(psi)*cos(chi)*u0x[i]._M_im - sin(psi)*sin(chi)*u0x[i]._M_re +...
          ...+ sin(psi)*cos(chi)*u0y[i]._M_im + cos(psi)*sin(chi)*u0y[i]._M_re;
        ",
        "halfstep_nonlinear",
        preamble="#include <pycuda-complex.hpp>",)
```

-transKernelElipInv Para la conversión de los pulsos de salida en el método elíptico.

```
      transKernelElipInv = ElementwiseKernel("pycuda::complex<float>*u0x,
pycuda::complex<float>*u0y, float psi, float chi, pycuda::complex<float>*u0",
        "
        u0[i]._M_re = cos(psi)*cos(chi)*u0x[i]._M_re - sin(psi)*sin(chi)*u0x[i]._M_im -...
        ... - sin(psi)*cos(chi)*u0y[i]._M_re + cos(psi)*sin(chi)*u0y[i]._M_im;
        u0[i]._M_im = cos(psi)*cos(chi)*u0x[i]._M_im + sin(psi)*sin(chi)*u0x[i]._M_re - ...

        ...- sin(psi)*cos(chi)*u0y[i]._M_im - cos(psi)*sin(chi)*u0y[i]._M_re;
        ",
        "halfstep_nonlinear",
        preamble="#include <pycuda-complex.hpp>",)
```

-paramhKernel Para la obtención de los elementos h_{11}, h_{12}, h_{21} y h_{22} de la matriz de dispersión en el método circular.

```
      paramhKernel = ElementwiseKernel(
      "pycuda::complex<float>*halfstepa, pycuda::complex<float>*halfstepb, float psi, float
chi,
      pycuda::complex<float>*h11, pycuda::complex<float>*h12,pycuda::complex<float>*h21,
      pycuda::complex<float>*h22",
       ",
       h11[i]._M_re = ((1.0+sin(2.0*chi))*halfstepa[i]._M_re + (1.0-sin(2.0*chi)) *...
...* halfstepb[i]._M_re)/2.0;
       h11[i]._M_im = ((1.0+sin(2.0*chi))*halfstepa[i]._M_im + (1.0-sin(2.0*chi)) *...
...*halfstepb[i]._M_im)/2.0;
       h22[i]._M_re = ((1.0-sin(2.0*chi))*halfstepa[i]._M_re + (1.0+sin(2.0*chi)) *...
...*halfstepb[i]._M_re)/2.0;
       h22[i]._M_im = ((1.0-sin(2.0*chi))*halfstepa[i]._M_im + (1.0+sin(2.0*chi)) *...
...*halfstepb[i]._M_im)/2.0;
       h12[i]._M_re = (cos(2.0*psi)*cos(2.0*chi)*(halfstepa[i]._M_im - halfstepb[i]._M_im)
+...
       ...+ sin(2.0*psi)*cos(2.0*chi)*(halfstepa[i]._M_re - halfstepb[i]._M_re))/2.0;
       h12[i]._M_im = (-cos(2.0*psi)*cos(2.0*chi)*(halfstepa[i]._M_re - halfstepb[i]._M_re)
+...
       +... sin(2.0*psi)*cos(2.0*chi)*(halfstepa[i]._M_im - halfstepb[i]._M_im ))/2.0;
       h21[i]._M_re = (-cos(2.0*psi)*cos(2.0*chi)*(halfstepa[i]._M_im - halfstepb[i]._M_im)
+...
       sin(2.0*psi)*cos(2.0*chi)*(halfstepa[i]._M_re - halfstepb[i]._M_re ))/2.0;
       h21[i]._M_im =(sin(2.0*psi)*cos(2.0*chi)*(halfstepa[i]._M_im - halfstepb[i]._M_im)
+...
       cos(2.0*psi)*cos(2.0*chi)*(halfstepa[i]._M_re - halfstepb[i]._M_re ))/2.0;
       ",
       "halfstep_nonlinear",
       preamble="#include <pycuda-complex.hpp>",)
```

-halfStepCircKernel para la obtención del producto de dispersión en el método de polarización circular dentro del split-step de Fourier simetrizado. Hace uso de los elementos de la matriz de dispersión calculados en el kernel anterior y de los pulsos transformados y pasados al dominio de la frecuencia.

```
      halfStepCircKernel = ElementwiseKernel(
      "pycuda::complex<float>*ua, pycuda::complex<float>*ub, pycuda::complex<float>,
      *h1, pycuda::complex<float>*h2, pycuda::complex<float>*uhalf",
       ",
       uhalf[i]._M_re = ua[i]._M_re * h1[i]._M_re - ua[i]._M_im * h1[i]._M_im + ub[i]._M_re
* h2[i]._M_re - ub[i]._M_im * h2[i]._M_im;
       uhalf[i]._M_im = ua[i]._M_re * h1[i]._M_im + ua[i]._M_im * h1[i]._M_re + ub[i]._M_re
* h2[i]._M_im + ub[i]._M_im * h2[i]._M_re;
```

```
 ",
 "halfstep_linear",
 preamble= "#include <pycuda-complex.hpp>",)
```

-*halfStepElipKernel* para la obtención del producto de dispersión en el método de polarización elíptica dentro del split-step de Fourier simetrizado. Se hace uso de ha, hb y los dos pulsos transformados y pasados al dominio de la frecuencia.

```
halfStepElipKernel = ElementwiseKernel(
"pycuda::complex<float>*u, pycuda::complex<float>*halfstep, pycuda::complex<float>*uhalf",
 ",
 uhalf[i]._M_re = u[i]._M_re * halfstep[i]._M_re - u[i]._M_im * halfstep[i]._M_im;
 uhalf[i]._M_im = u[i]._M_re * halfstep[i]._M_im + u[i]._M_im * halfstep[i]._M_re;
 ",
 "halfstep_linear",
 preamble="#include <pycuda-complex.hpp>",)
```

-*nlKernelCircular* para la obtención en el método circular del producto de no linealidad en el dominio del tiempo entre el parámetro de no linealidad y el resultado del paso dispersivo anterior una vez pasado al dominio del tiempo a través de la transformada inversa de Fourier.

```
nlKernelCircular = ElementwiseKernel(
"pycuda::complex<float>*uhalf, pycuda::complex<float>*u0a, pycuda::complex<float>,
*u1a, pycuda::complex<float>*u0b, pycuda::complex<float>*u1b, pycuda::complex<float>*uv,

float gamma, float dz",
 ",
 float u0a_int = pow(u0a[i]._M_re,2) + pow(u0a[i]._M_im,2);
 float u1a_int = pow(u1a[i]._M_re,2) + pow(u1b[i]._M_im,2);
 float u0b_int = pow(u0b[i]._M_re,2) + pow(u0a[i]._M_im,2);
 float u1b_int = pow(u1b[i]._M_re,2) + pow(u1b[i]._M_im,2);
 float realArg = -gamma*(u0a_int + u1a_int + 2.0*u0b_int + 2.0*u1b_int)*dz/3.0;
 float euler1 = cos(realArg);
 float euler2 = sin(realArg);
 uv[i]._M_re = uhalf[i]._M_re * euler1 - uhalf[i]._M_im * euler2;
 uv[i]._M_im = uhalf[i]._M_im * euler1 + uhalf[i]._M_re * euler2;
 ",
 "halfstep_nonlinear",
 preamble="#include <pycuda-complex.hpp>",)
```

-*nlKernelElipt* misma utilidad que el anterior para polarización elíptica.

```
nlKernelElipt = ElementwiseKernel("pycuda::complex<float>*uhalf, pycuda::complex<float>*u0a,
pycuda::complex<float>*u1a, pycuda::complex<float>*u0b, pycuda::complex<float>
*u1b, pycuda::complex<float>*uv, float gamma, float chi, float dz",
 ",
 float u0a_int = pow(u0a[i]._M_re,2) + pow(u0a[i]._M_im,2);
 float u1a_int = pow(u1a[i]._M_re,2) + pow(u1b[i]._M_im,2);
 float u0b_int = pow(u0b[i]._M_re,2) + pow(u0a[i]._M_im,2);
 float u1b_int = pow(u1b[i]._M_re,2) + pow(u1b[i]._M_im,2);
 float realArg = -gamma*((1 + pow(cos(2.0*chi),2)/2)*(u0a_int + u1a_int) +...
 (1 + pow(sin(2*chi),2))*(u0b_int + u1b_int))*dz/3.0;
 float euler1 = cos(realArg);
 float euler2 = sin(realArg);
 uv[i]._M_re = uhalf[i]._M_re * euler1 - uhalf[i]._M_im * euler2;
 uv[i]._M_im = uhalf[i]._M_im * euler1 + uhalf[i]._M_re * euler2;
 ",
 "halfstep_nonlinear",
 preamble="#include <pycuda-complex.hpp>",)
```

-*computeError* obtención de la norma dos de la diferencia entre el pulso de entrada en cada paso del método y el pulso obtenido en cada avance de aproximación, para calcular el error entre ambos.

```
computeError = ReductionKernel(numpy.float32, neutral="0",
 reduce_expr=.a+b",, map_expr="pow(abs(a[i] - b[i]),2)",
 arguments="pycuda::complex<float>*a, pycuda::complex<float>*b",
 name=.error_reduction",
 preamble="#include <pycuda-complex.hpp>", )
```

-*complexDeepCopy* obtención en la gpu de un array de números complejos copia idéntica de otro.

```
complexDeepCopy = ElementwiseKernel("pycuda::complex<float>*u1, pycuda::complex<float>*u2",
 ü1[i]._M_re = u2[i]._M_re; u1[i]._M_im = u2[i]._M_im",
 "gpuarray_deepcopy",
 preamble="#include <pycuda-complex.hpp>",)
```

-*complexDeepCopyInv* obtención en la gpu de un array de números complejos cuyos elementos son los negativos del array de entrada.

```
 complexDeepCopyInv = ElementwiseKernel("pycuda::complex<float>*u1, pycuda::complex<float>*u2"
 ü1[i]._M_re = - u2[i]._M_re; u1[i]._M_im = - u2[i]._M_im",
 "gpuarray_deepcopy",
 preamble="#include <pycuda-complex.hpp>",)
```

3.2.3. Ejemplos de utilización de las funciones implementadas para la resolución de las ecuaciones de Schrödinger acopladas

Para la comprobación de los programas realizados de resolución de las ecuaciones de Schrödinger vectoriales se ha implementado una nueva función denominada `vectorial_2.py`. En ella se pretende hacer uso de la función `ssfvec1()` para la cpu, y de `ssfvecgpu()` para la gpu, ambas ubicadas en el archivo `ssfvec.py` añadido por nosotros a la librería PyOFTK.

En primer lugar presentamos una linea de tiempos con 2^{15} muestras que van desde -446ps a 446ps, idéntica a la realizada en los ejemplos pertenecientes a la solución escalar del método. El tamaño de salto que se utilizará en las iteraciones del método split-step será de $dz = 0{,}001km$ para un total de nz=20 iteraciones, es decir, se estudiará el pulso de salida a 20m del inicio. A su vez, las potencias de entrada del pulso serán las mismas para ambos ejes de polarización, e iguales a 1W. El coeficiente de no linealidad γ será igual a 3 1/w·km y los coeficientes de dispersión serán de GVD con $1{,}5ps^2/Km$ para ambos ejes de polarización, y un $\Delta\beta = \beta_{0x} - \beta_{0y} = 2\pi/L_B = 2\pi$ ya que consideramos una longitud de batido igual a un metro. Para fibras de alta birrefrigencia puede llegar a ser de incluso 1cm. Recordemos que el coeficiente de dispersión de velocidad de grupo β_2 y el coeficiente de no linealidad γ deben ser los mismos en ambos ejes de polarización para que el pulso sea emitido con una única longitud de onda λ y, sin embargo, los valores de β_1 suelen ser diferentes en ambos ejes para representar las distintas velocidades de grupo a las que viajan los componentes del pulso en cada eje de polarización. Empleamos método de componentes de polarización elíptica debido a que la longitud de batido es mucho menor a la no lineal, un número de iteraciones de aproximación del término no lineal para cada salto en el método split-step de

diez y una tolerancia de 10^{-5}. Vamos a suponer una desviación respecto a los ejes $\psi = \pi/4$, es decir, una inclinación de la elipse de 45° respecto al eje x. Por otra parte, consideraremos una elipticidad que vaya desde $\chi = 0$ o régimen de polarización lineal a derechas, a $\chi = \pi/4$ o en polarización circular. Si aumentamos el ángulo de elipticidad hasta llegar a los $\chi = \pi/2$ conseguiremos volver a la polarización lineal pero a izquierdas en lugar de a derechas. Las dos señales de entrada serán dos pulsos gaussianos, uno para el eje x y otro para el eje y, de misma potencia, mismo número de muestras y mismo ancho a intensidad mitad. Es decir, el pulso de entrada será idéntico para ambos ejes, las características de la fibra se encargarán de modificarlos a la salida.

Para observar los resultados de la transformación del pulso en ambos ejes, la diferencia de anchura y forma debida a la birrefrigencia de la fibra, se hará una representación gráfica tridimensional de la propagación a lo largo de la longitud de la fibra de los dos ejes de polarización. Para ello emplearemos la función `add_collection3d(poly, zs = z, zdir = 'y')` donde `poly` es el resultado de pasar los vectores bidimensionales de línea de tiempos y potencia de salida a lo largo de la propagación a través del comando `PolyCollection()`. De este modo se pueden acumular a lo largo del eje z todos los pulsos parciales como puede comprobarse en la siguiente representación gráfica en ambos ejes de polarización:

Se puede observar que, para los cinco ángulos de elipticidad entre la polarización lineal y circular, y de ésta de nuevo a la lineal, en las propagaciones centrales pertenecientes a 30°, 45° y 60°, aparecen pulsos secundarios; mientras que en las restantes, pertenecientes a 0° y 90°, dicho chirpeo no se produce. Una explicación factible para estos patrones intermedios puede ser la interferencia entre la componente del pulso en cada uno de los ejes. Si la polarización es lineal, el pulso se propaga en un solo eje de modo que no hay interferencia posible; si la polarización es elíptica, en uno de los ejes el pulso se desplaza a una velocidad mayor que el otro y a intensidades diferentes, lo que produce que entre ambos pulsos aparezcan zonas de máximos, interferencia constructiva, y zonas de mínimos, interferencia destructiva. Este tipo de efectos son denominados de inestabilidad de modulación (*Modulations Instability*) y son causados por el incremento arbitrario de un cierto margen a la señal original al inicio de la propagación. En la ecuación escalar son producidos por la SPM-inducida y sólo son visibles en régimen anómalo; sin embargo, para las ecuaciones acopladas en fibras birrefrigentes los efectos se dejan sentir en ambos regímenes de propagación y su causante es la XPM-inducida. Debido a esto, aparecen pequeñas variaciones de frecuencia entorno a la central en ambas componentes del pulso que originarán en principio un ensanchamiento espectral que deribe en trenes de pulsos temporales como los que aparecen en la figura adjunta. Es decir, se trata de efectos no lineales que van a depender de la potencia de entrada y del índice de no linealidad. Si elegimos valores suficientemente pequeños como para garantizar una longitud de no linealidad mayor a la de la fibra, estos efectos dejarán de producirse.

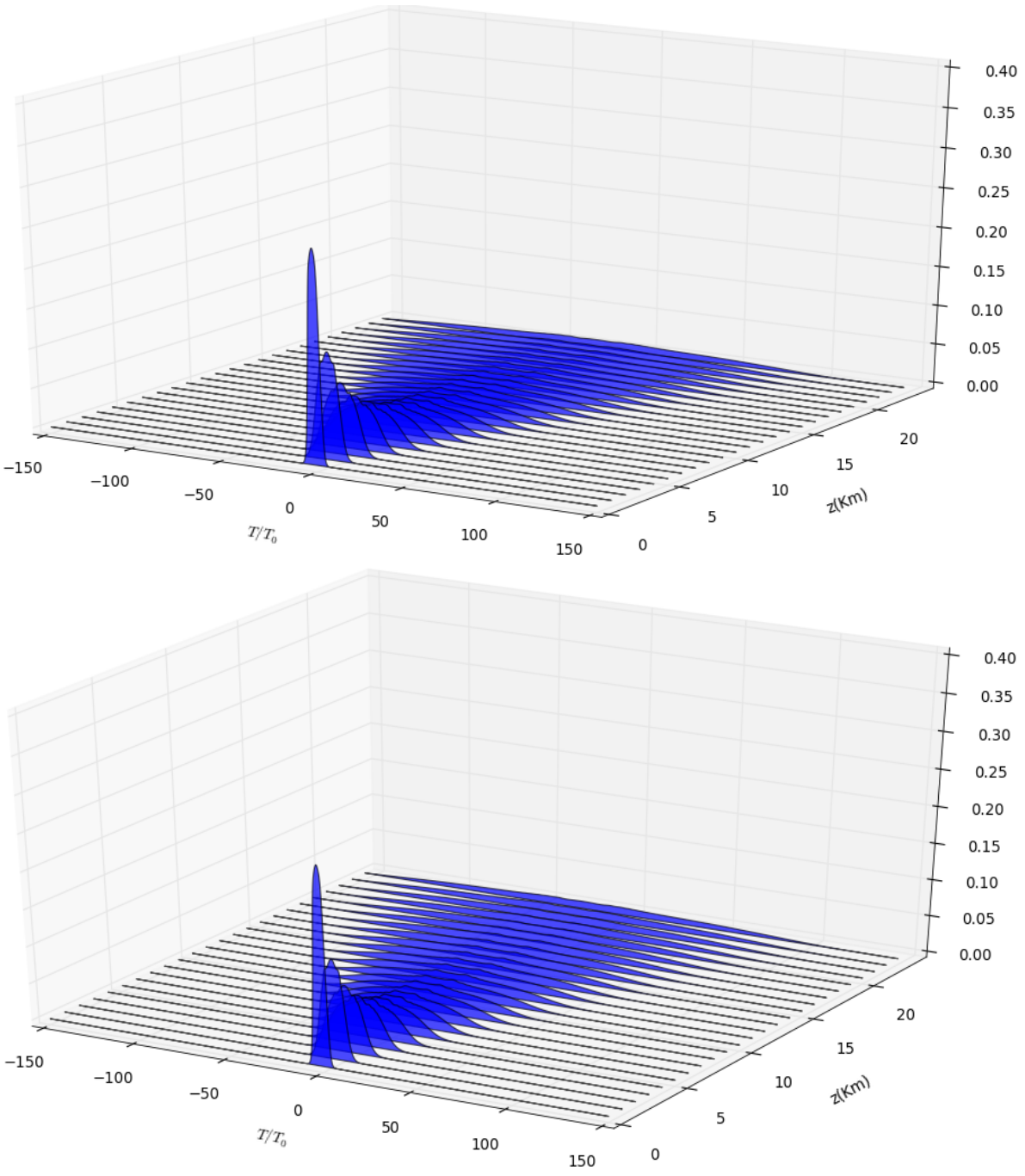


Figura 3.23: Propagación de un pulso de 1W a través de una fibra birrefrigente con dispersión de velocidad de grupo en ambos ejes de polarización igual a $\beta_2 = 1{,}5$.

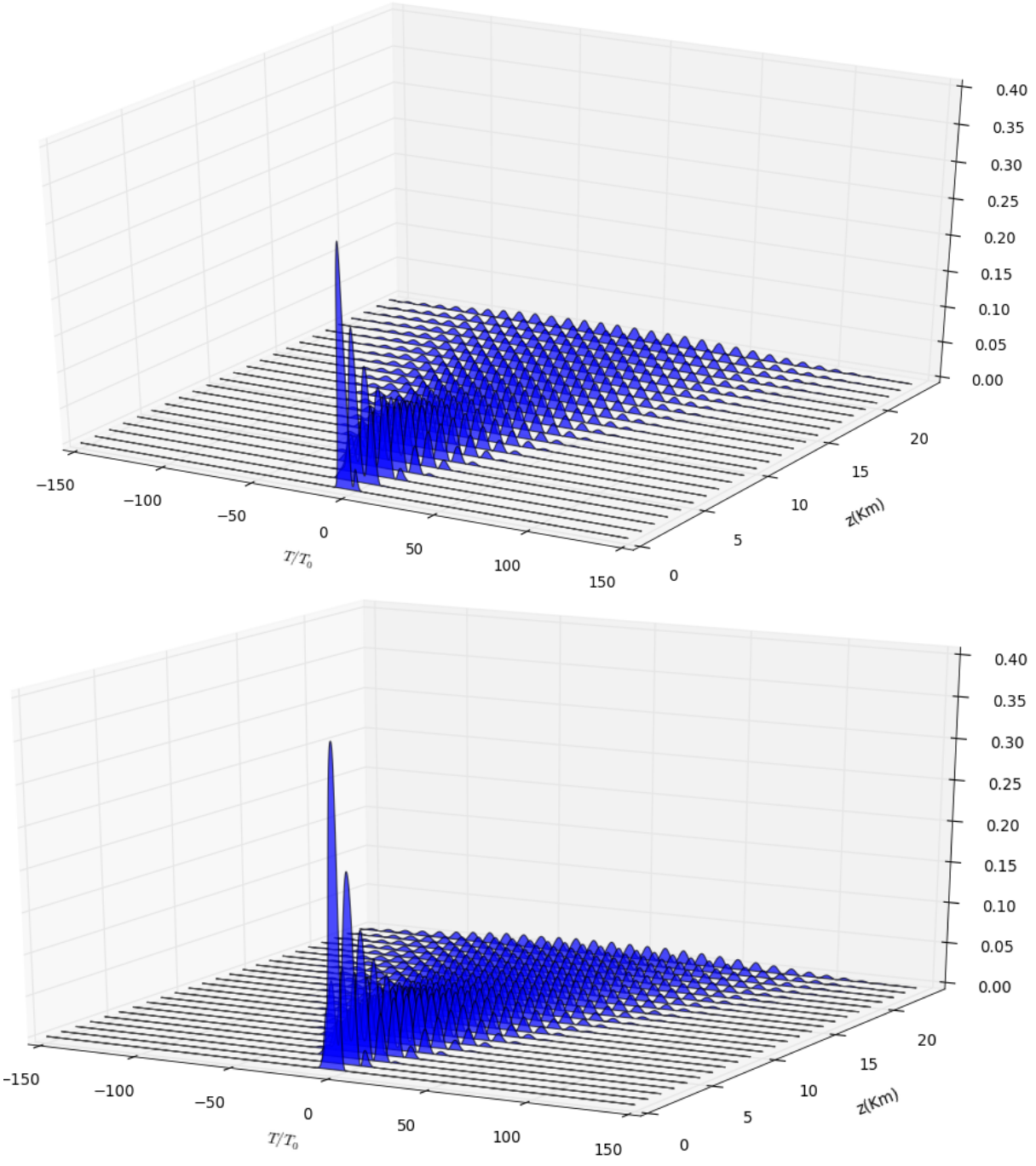


Figura 3.24: Propagación de un pulso de 1W a través de una fibra birrefrigente con dispersión de velocidad de grupo en ambos ejes de polarización igual a $\beta_2 = 1{,}5$.

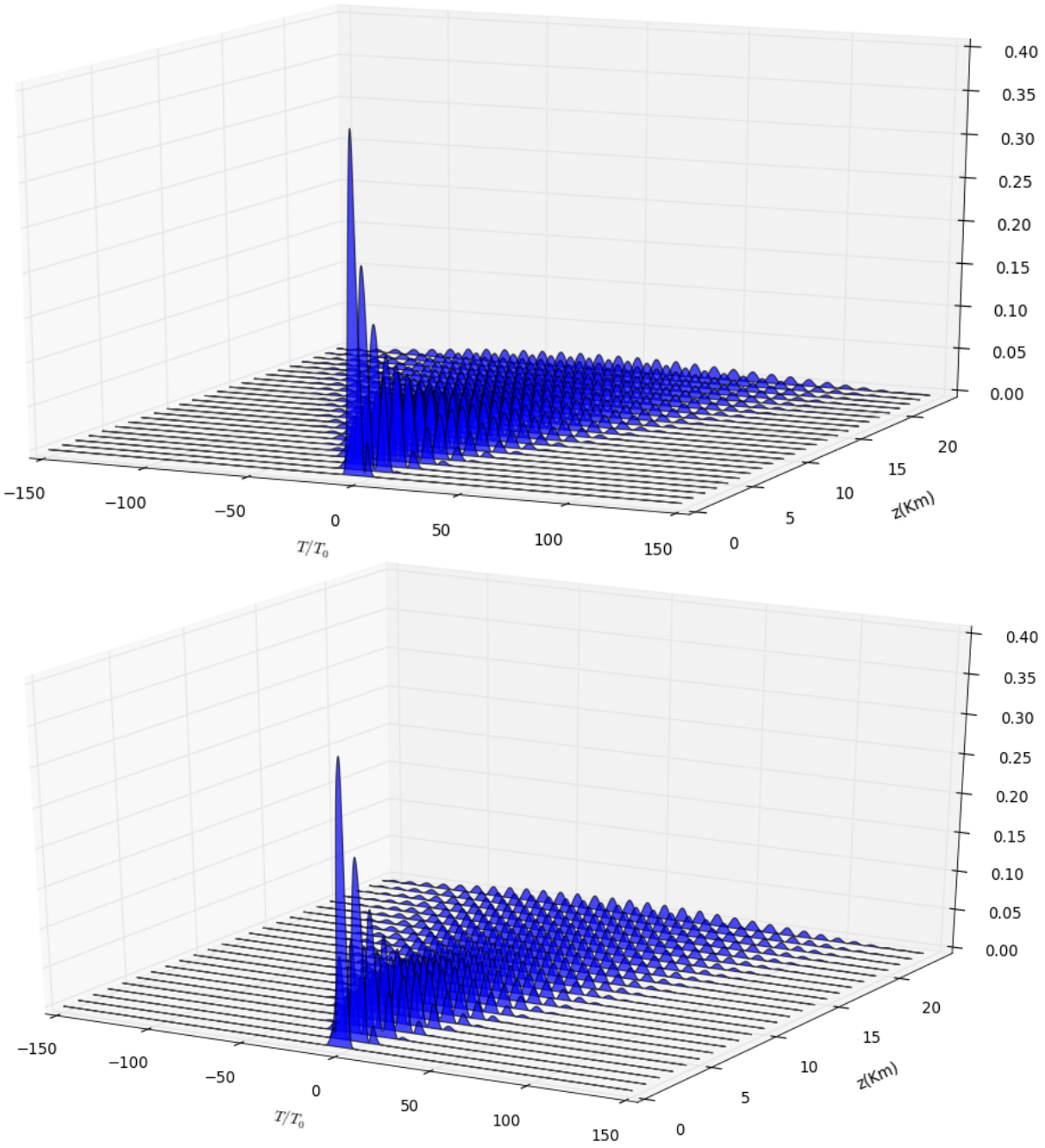


Figura 3.25: Propagación de un pulso de 1W a través de una fibra birrefrigente con dispersión de velocidad de grupo en ambos ejes de polarización igual a $\beta_2 = 1{,}5$.

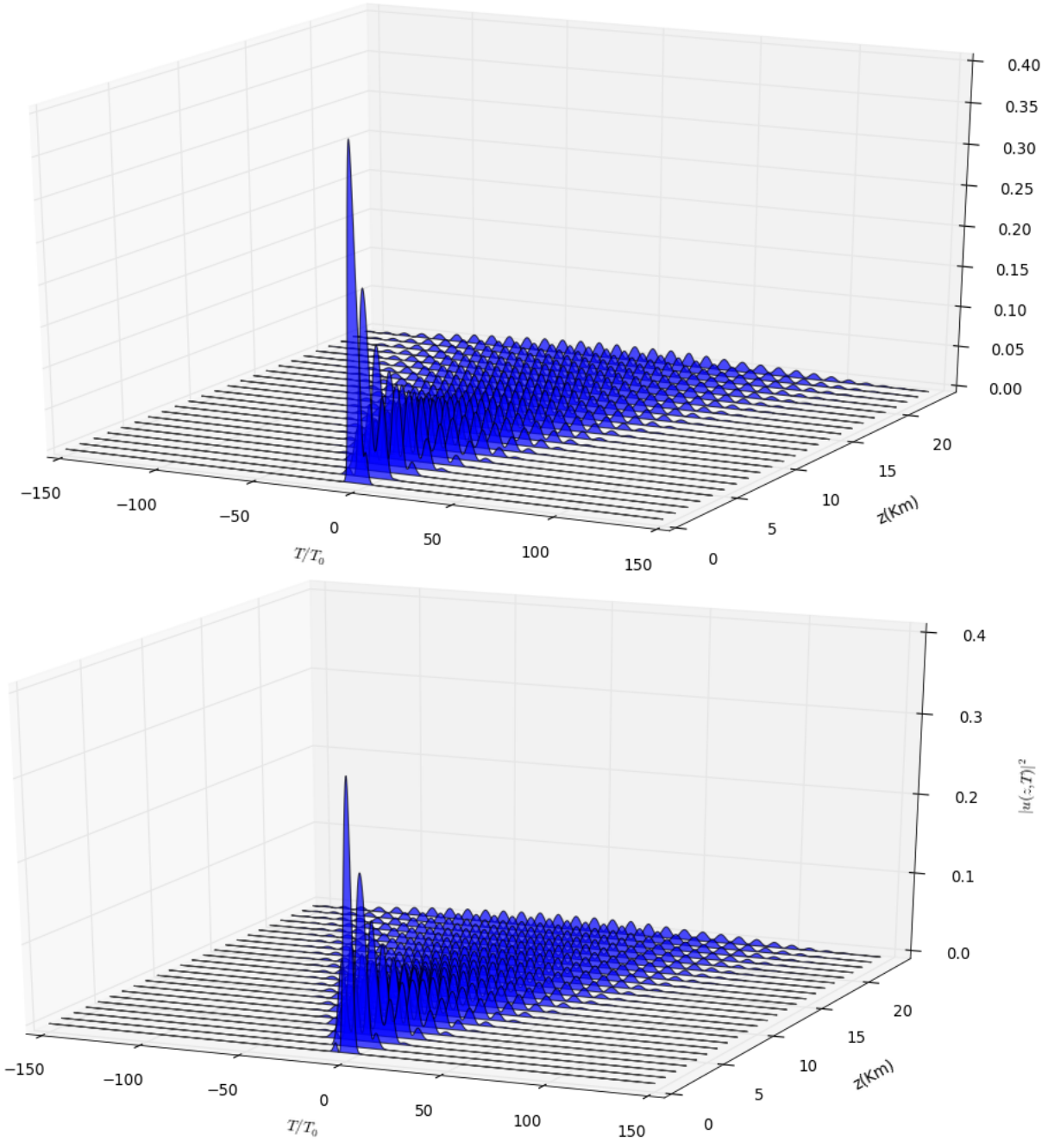


Figura 3.26: Propagación de un pulso de 1W a través de una fibra birrefrigente con dispersión de velocidad de grupo en ambos ejes de polarización igual a $\beta_2 = 1{,}5$.

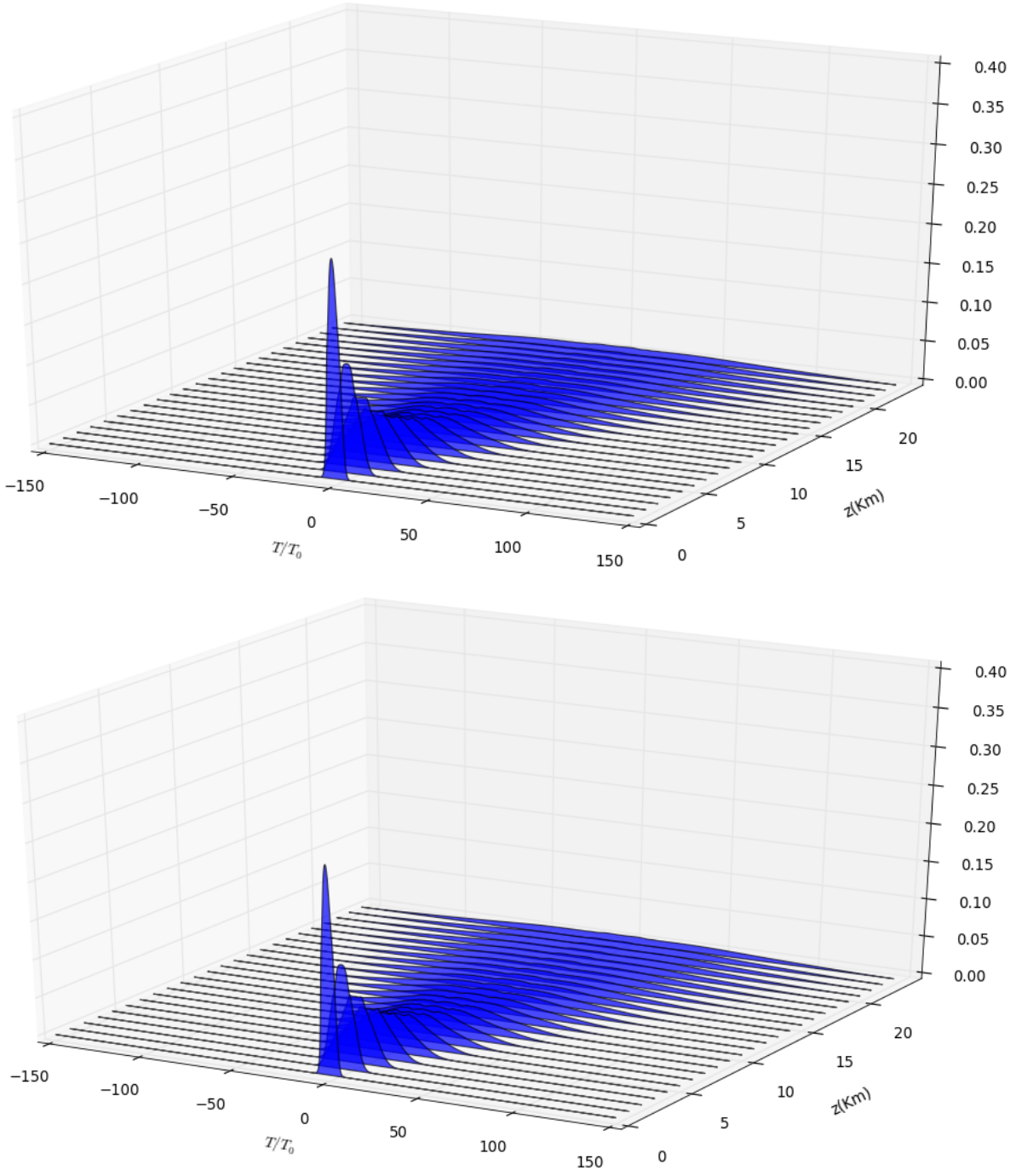


Figura 3.27: Propagación de un pulso de 1W a través de una fibra birrefrigente con dispersión de velocidad de grupo en ambos ejes de polarización igual a $\beta_2 = 1{,}5$.

Conclusions

Through PyOFTK library written in Python, using the SSPROP program at the University of Maryland, there have been a series of teaching examples on the propagation of an optical pulse through a nonlinear and dispersive fiber with the split-method Fourier symmetrized step that solves the Schrödinger equation in both its scale and its vector version. In the first, after describing the evolution of the pulse in its different regimes, depending on the distance of dispersion and nonlinearity, they have made several simulations of optical communication systems obtaining values for different fiber lengths very close to expected. To check the results has utlizado software NLSE Rochester University that performs graphical representations of different types of pulses at a given with the parameters that govern the propagation distance. This method may be of special interest to describe the behavior of a solitary pulse along a given means with which to implement media access techniques division multiplexing optical time OTDM. The soliton provides a number of advantages over techniques wavelength division but is not yet refined enough to isolate the effects of aliasing and inter-symbol interference. Testing in this regard, utlizando for the examples in this issue PyOFTK locked laser light pulse can be a good way forward for future projects.

Moreover, the procedure performed by the split-step method has been implemented both to operate sequentially in the CPU and parallel, sharing resources

cpu and gpu through the GPGPU library called CUDA, and to compare the efficiency of both methods was performed an analysis of time with Runsnake Python application in which it has been clearly prove that the parallelization computation speed increases progressively as does the number of iterations.

Finally, the solution of the coupled Schrödinger equations are useful for describing the propagation of a pulse in a medium birefringent where the refractive indices are slightly different according to the polarization axis so that each of the components travel through the guide at a different speed. The same SSPROP software uses two different methods depending on the type of birefringence of the fiber, whether small birefringence for which it uses the elliptical method, or large circular birefringence with method. The latter is used very rarely and for very small fiber lengths, since for large birrefrigencias can not be neglected increased dispersion which complicates the calculations; however, optical fibers normally do not possess differences between indexes or β_n greater than 10^{-6}, suitable for use elliptical method.

The archives and library PyOFTK procedures have been modified to suit the requirements of the simulations, as well as adding new ones and not only as examples of use, but as new methods for parallelization implemented with CUDA and Reikna. They will be incorporated as accompanying documentation to work.

Apéndice A

Propagación del Pulso en Fibra Óptica.

[1]

Para poder desarrollar la ecuación que describe la propagación de un pulso a lo largo de una fibra óptica no lineal, conviene comenzar explicando la ecuación de onda y su deducción en una guía óptica cilíndrica con pérdidas despreciables, para lo que trabajaremos en el dominio de la frecuencia.

La ecuación de onda de un campo electromagnético en ausencia de fuente externa viene definido por las ecuaciones de Maxwell del siguiente modo:

$$\nabla^2 \vec{E}(\vec{r},\omega) = -\frac{\omega^2}{c^2}\varepsilon(\omega)\vec{E}(\vec{r},\omega)$$

Donde la constante dieléctrica dependiente de la frecuencia para un medio lineal puede definirse como:

$$\varepsilon(\omega) = 1 + \hat{\chi}^{(1)}(\omega) = (n + j\alpha\frac{c}{2\omega})^2$$

El índice de refracción dependiente de la frecuencia se puede escribir en función del tensor de orden uno de la susceptibilidad eléctrica χ como $n(\omega) = 1 + \frac{1}{2}Re[\hat{\chi}^{(1)}(\omega)]$; mientras que la constante de pérdida α también depediente de la frecuencia y en función de la susceptibilidad eléctrica vendría definida como $\alpha(\omega) = \frac{\omega}{nc}Im[\hat{\chi}^{(1)}(\omega)]$. En ausencia de pérdidas, suposición que realizamos para fibras ópticas de alta velocidad, podemos despreciar dicha constante α, de modo que la ecuación de onda sólo aparezca en función del índice de refracción:

$$\nabla^2\vec{E}(\vec{r},\omega) + \frac{\omega^2}{c^2}n^2(\omega)\vec{E}(\vec{r},\omega) = 0$$

Para un medio propagante cilíndrico en una sola dirección espacial, por convenio se elige la coordena z como el medio de propagación, la ecuación de onda se puede escribir:

$$\frac{\partial^2\vec{E}_z}{\partial\rho^2} + \frac{1}{\rho}\frac{\partial\vec{E}_z}{\partial\rho} + \frac{1}{\rho^2}\frac{\partial^2\vec{E}_z}{\partial\phi^2} + \frac{\partial^2\vec{E}_z}{\partial z^2} + n^2(\omega)k_0^2\vec{E}_z = 0$$

Para resolver esta ecuación diferencial de varias variables se emplea comúnmente el método de separación de variables tras definir la intensidad de campo eléctrico propagado a lo largo del eje z del siguiente modo:

$$\vec{E}_z = F(\rho)\phi(\phi)z(z)$$

$$\frac{\partial^2\vec{\phi}(\phi)}{\partial\phi^2} = -\rho^2n^2k_0^2\phi(\phi); \quad \rho nk_0 = m; \quad \frac{\partial^2\vec{\phi}(\phi)}{\partial\phi^2} = -m^2\phi(\phi)$$

$$\vec{\phi}(\phi) = \phi_c(\omega)e^{\pm jm\phi}$$

$$\frac{\partial^2\vec{z}(z)}{\partial z^2} = -n^2k_0^2z(z); \quad nk_0 = \beta; \quad \frac{\partial^2\vec{z}(z)}{\partial z^2} = -\beta^2z(z)$$

$$\vec{z}(z) = z_c(\omega)e^{\pm j\beta z}$$

Para obtener la componente $F(\rho)$ sustituimos en la ecuación de onda en coorde-

nadas cilíndricas las derivadas parciales de segundo orden de las otras coordenadas por sus soluciones obtenidas en las ecuaciones anteriores. La ecuación resultante es la siguiente:

$$\frac{\partial^2 \vec{F}(\rho)}{\partial \rho^2} + \frac{1}{\rho}\frac{\partial \vec{F}(\rho)}{\partial \rho} + F(\rho)(n^2 k_0^2 - \frac{m^2}{\rho^2} - \beta^2) = 0$$

Esta ecuación se denomina ecuación diferencial de la función de Bessel y su solución compleja se compone de dos términos:

$$\vec{F}(\rho) = F_{c1}(\omega) J_m(\kappa\rho) + F_{c2}(\omega) N_m(\kappa\rho)$$

Donde J_m es la función de Bessel de primer orden o para valores enteros y N_m es la ecuación de Neumann o ecuación de Bessel de segundo orden, para valores no enteros. La ecuación de Neumann en valores enteros tiene singularidades, y por tanto nos interesa sólo la función de Bessel de primera especie para resolver la ecuación de onda en el modo fundamental de la fibra. Para un radio del núcleo de la fibra a, la solución será:

$$\vec{F}(\rho) = F_c(\omega)\vec{J}_m(\kappa\rho); \quad \rho \leq a$$

$$\vec{F}(\rho) = F_c(\omega)\vec{K}_m(\gamma\rho); \quad \rho \geq a$$

$$\kappa = \sqrt{n_1^2 k_0^2 - \beta^2}; \qquad \gamma = \sqrt{\beta^2 - n_2^2 k_0^2}$$

En la fibra óptica se dispone de dos medios diferentes: el núcleo, con un índice de refracción n_1, y la cubierta, con n_2, por tanto la propagación en ρ deberá dividirse en la solución para uno y para el otro. Para que la transmisión en la fibra sea efectiva nos interasan aquellos modos en que la propagación en la cubierta se anula, realizándose únicamente en el núcleo. Se denomina número de onda de

corte o longitud de onda de corte, a aquella longitud a partir de la cual la onda comienza a transmitirse a lo largo de la fibra para un modo determinado. Según las condiciones de contorno, en la interfaz de la fibra ($\rho = a$) existe una continuidad entre la onda transmitida en un medio y en el otro.

$$\kappa^2 + \gamma^2 = (n_1^2 - n_2^2)k_0^2$$

Y, por tanto, el número de onda de corte sera donde $\gamma = 0$:

$$\kappa_c = \sqrt{(n_1^2 - n_2^2)k_0^2}$$

$$\kappa_c = k_0\sqrt{n_1^2 - n_2^2}$$

$$\kappa_c = k_0 NA; \quad V = \kappa_c a; \quad V = ak_0 NA$$

La frecuencia normalizada V para una fibra monomodo calculada a partir de sus parámetros físicos debe dar un valor igual a $V_c = 2,405$, que será donde $J_0(V_c) = 0$. El único modo en fibras ópticas que cumple este requisito es el HE_{11} y es llamado el modo fundamental de la fibra. La solución para el campo eléctrico vendrá dada por composición de las componentes resueltas:

$$\vec{E}_z(\vec{r}, \omega) = A(\omega)J_m(\kappa\rho)e^{j\beta z}e^{jm\phi} \quad \rho \leq a$$

Si asumimos que la luz incidente es polarizada a lo largo del eje x, y por tanto con componente ϕ nula, el campo eléctrico para el modo fundamental HE_{11} se puede expresar como:

$$\vec{E}(\vec{r}, \omega) = \hat{x}[A(\omega)F(x, y)e^{j\beta(\omega)z}]$$

En el interior del núcleo, la función F(x,y) es la solución encontrada para el primer modo o fundamental de la fibra teniendo en cuenta que la distancia radial ρ es igual a $\sqrt{x^2+y^2}$.

$$F(x,y) = J_0(\kappa\rho), \qquad \rho \leq a$$

Mientras que en el cladding o cubierta, el campo se desvanece de manera exponencial desde el mismo valor en la interfaz core-cladding:

$$F(x,y) = \sqrt{\frac{a}{\rho}} J_0(\kappa a) e^{-\gamma(\rho-a)}, \qquad \rho \geq a$$

Debido a que trabajar con estos parámetros resulta incómodo en la práctica, suele aproximarse la solución del modo fundamental a una distribución gaussiana de la forma:

$$F(x,y) = e^{-\frac{x^2+y^2}{w^2}}$$

El parámetro w, o anchura modal, es determinado por adecuar la distribución exacta a una forma gaussiana o siguiendo un procedimiento de variación.

A.1. Ecuación de Propagación del Pulso

Para el estudio de los fenómenos dispersivos y no linealidades en una fibra óptica suelen utilizarse pulsos muy estrechos, del orden de 10ns a 10fs. De esta forma se puede observar más claramente los efectos sobre dicho pulso de los fenómenos que se quieren analizar tanto en forma como en espectro.

$$\nabla^2 \vec{E} = \frac{1}{c^2} \frac{\partial^2 \vec{E}}{\partial t^2} + \mu_0 \frac{\partial^2 P}{\partial t^2}$$

$$\vec{P}(\vec{r},t) = \vec{P}_L(\vec{r},t) + \vec{P}_{NL}(\vec{r},t)$$

De estas dos ecuaciones, donde P es el vector de polarización con su componente lineal y su componente no lineal, se deduce aquella que gobierna la propagación de un pulso corto a lo largo de una fibra óptica.

$$\nabla^2\vec{E} - \frac{1}{c^2}\frac{\partial^2\vec{E}}{\partial t^2} = \mu_0\frac{\partial^2\vec{P}_L}{\partial t^2} + \mu_0\frac{\partial^2\vec{P}_{NL}}{\partial t^2}$$

Se han de realizar varias simplificaciones para poder resolver esta ecuación diferencial. En primer lugar la P_{NL} es tratada como una pequeña perturbación de P_L. El campo óptico debe mantener la polarización a lo largo de toda la fibra, esto se consigue por medio de fibras de permanencia de polarización tales como la fibra panda. Y, por último, el campo óptico es asumido como cuasi-monocromático, donde el pulso espectral tiene un ancho mínimo ($\frac{\Delta\omega}{\omega_0}$)

La variación lenta de la envolvente del campo puede ser aproximada para separarla de la variación rápida, por la siguiente expresión:

$$\vec{E}_{slow}(\vec{r},t) = \frac{1}{2}\hat{x}[E(\vec{r},t)e^{-j\omega_0 t} + cc]$$

$$\vec{P}_L(\vec{r},t) = \frac{1}{2}\hat{x}[P_L(\vec{r},t)e^{-j\omega_0 t} + cc]$$

$$\vec{P}_{NL}(\vec{r},t) = \frac{1}{2}\hat{x}[P_{NL}(\vec{r},t)e^{-j\omega_0 t} + cc]$$

$$P_L(\vec{r},t) = \varepsilon_0\int_{-\infty}^{\infty}\chi^{(1)}_{\chi\chi}(t-t^l)E(\vec{r},t^l)e^{j\omega_0(t-t^l)}dt^l \approx \varepsilon_0\chi^{(1)}_{\chi\chi}E(\vec{r},t^l)$$

$$P_{NL}(\vec{r},t) = \varepsilon_0\int\int\int_{-\infty}^{\infty}\chi^{(3)}(t-t_1,t-t_2,t-t_3):\vec{E}(\vec{r},t_1)\vec{E}(\vec{r},t_2)\vec{E}(\vec{r},t_3)dt_1dt_2dt_3$$

$$P_{NL}(\vec{r},t) \approx \varepsilon_0\varepsilon_{NL}E(\vec{r},t), \qquad donde \;\; \varepsilon_{NL} = \frac{3}{4}\chi^{(3)}_{\chi\chi\chi\chi}|E(\vec{r},t)|^2$$

Si sustituimos estos resultados en la ecuación de onda para propagación de pulsos cortos en el dominio de la frecuencia, podemos observar cómo ésta satisface la

ecuación de Helmholtz:

$$\nabla^2\vec{E}(\vec{r},\omega-\omega_0)+\omega^2[\frac{\vec{E}(\vec{r},\omega-\omega_0)}{c^2}+\mu_0(P_L(\vec{r},\omega-\omega_0)+P_{NL}(\vec{r},\omega-\omega_0))]=0$$

$$\nabla^2E(\vec{r},\omega-\omega_0)+\varepsilon(\omega)k_0^2E(\vec{r},\omega-\omega_0)=0$$

$$\varepsilon(\omega)=1+\chi^{(1)}_{\chi\chi}(\omega)+\varepsilon_{NL}$$

Para resolver la ecuación volvemos a utilizar el método de la separación de variables, en esta ocasión en coordenadas cartesianas y fijando una sola variable F para las coordenada x e y.

$$\vec{E}=F(x,y)A(z)$$

$$\frac{\partial^2F_x}{\partial x^2}F_yA_z+\frac{\partial^2F_y}{\partial y^2}F_xA_z+\frac{\partial^2A_z}{\partial z^2}F_xF_y+\epsilon(\omega)k_0^2E=0$$

$$\frac{\partial^2F_x}{\partial x^2}F_yA_z+\frac{\partial^2F_y}{\partial y^2}F_xA_z+\frac{\partial^2A_z}{\partial z^2}F_xF_y+\epsilon(\omega)k_0^2F_xF_yA_z=0$$

Podemos tomar como constantes la distribución espacial del campo F(x,y) para obtener la ecuación de la ecuación de propagación de la envolvente lenta a lo largo del eje z, A(z):

$$\frac{\partial^2A_z}{\partial z^2}+\varepsilon(\omega)k_0^2A_z=0$$

Una posible solución a esta ecuación es $A_z=A_{zc}e^{j\beta_0z}$ que sustituimos en la original:

$$\frac{\partial A_z}{\partial z}=e^{j\beta_0z}\frac{\partial A_{zc}}{\partial z}+j\beta_0A_{zc}e^{j\beta_0z}$$

$$\frac{\partial^2A_z}{\partial z^2}=\frac{\partial^2A_{zc}}{\partial z^2}+2j\beta_0\frac{\partial A_{zc}}{\partial z}-\beta_0^2A_{zc}$$

$$\frac{\partial^2A_{zc}}{\partial z^2}+2j\beta_0\frac{\partial A_{zc}}{\partial z}-\beta_0^2A_{zc}+\varepsilon(\omega)k_0^2A_{zc}=0$$

La segunda derivada es despreciada debido a que la función Az varía lentamente

en la dirección z. La ecuación resultante queda expresada en función de la primera derivada:

$$2j\beta_0\frac{\partial A_{zc}}{\partial z} + (\hat{\beta}^2 - \beta_0^2)A_{zc} = 0$$

Para encontrar la ecuación que resuelva la componente F(x,y), en lugar de suponer la Az como constante para la resolución de la derivada segunda respecto a z, se sustituye por el valor obtenido en la ecuación anterior. De este modo se anularán las Az y nos quedará todo en función de las constantes de propagación y de la F(x,y):

$$A_z\frac{\partial^2 F(x,y)}{\partial x^2} + A_z\frac{\partial^2 F(x,y)}{\partial y^2} + F(x,y)\frac{\partial^2 A_z}{\partial z^2} + \varepsilon(\omega)k_0^2 A_z F(x,y) = 0$$

$$A_z\frac{\partial^2 F(x,y)}{\partial x^2} + A_z\frac{\partial^2 F(x,y)}{\partial y^2} - F(x,y)A_z\hat{\beta}^2 + \varepsilon(\omega)k_0^2 A_z F(x,y) = 0$$

$$\frac{\partial^2 F(x,y)}{\partial x^2} + \frac{\partial^2 F(x,y)}{\partial y^2} + (\varepsilon(\omega)k_0^2 - \hat{\beta}^2)F(x,y) = 0$$

La constante dieléctrica puede ser aproximada a partir de las siguientes igualdades:

$$\varepsilon(\omega) = 1 + \chi_{\chi\chi}^{(1)} + \varepsilon_{NL};\quad \varepsilon(\omega) = (\hat{n} + j\frac{\hat{\alpha}}{2k_0})^2;\quad \varepsilon(\omega) = (n + \Delta n)^2 \approx n^2 + 2n\Delta n$$

$$\varepsilon_{NL} = \frac{3}{4}\chi_{\chi\chi\chi\chi}^{(3)}|E|^2;\quad \hat{n} = n + n_{NL}|E|^2;\quad \hat{\alpha} = \alpha + \alpha_{NL}|E|^2$$

$$\Delta n = n_{NL}|E|^2 + j\frac{\hat{\alpha}}{2k_0}$$

Aunque el Δn no afecta a la distribución modal de F(x,y), los valores propios de $\beta(\omega)$ se transforman en:

$$\hat{\beta}(\omega) = \beta(\omega) + \Delta\beta;\quad \Delta\beta = \frac{k_0\int\int_{-\infty}^{\infty}\Delta n|F(x,y)|^2 dxdy}{\int\int_{-\infty}^{\infty}|F(x,y)|^2 dxdy}$$

En la ecuación de la envolvente del pulso de variación lenta A(z,t), se puede realizar una simplificación para las constantes de propagación:

$$\hat{\beta}^2 - \beta_0^2 = (\hat{\beta} + \beta_0)(\hat{\beta} - \beta_0) = 2\beta_0(\hat{\beta} - \beta_0)$$

De tal modo que la ecuación queda escrita de la siguiente manera:

$$\frac{\partial A(\vec{r},\omega)}{\partial z} = j[\beta(\omega) + \Delta\beta - \beta_0]A(\vec{r},\omega)$$

Cada componente espectral dentro de la envolvente del pulso adquiere, a medida que se desplaza por la fibra, un incremento de fase que es dependiente tanto de la frecuencia como de la intensidad.

Aún se puede simplificar más la expresión si se utiliza el desarrollo en serie de Taylor para definir la constante de propagación dependiente de la frecuencia:

$$\beta(\omega) - \beta_0 = (\omega - \omega_0)\beta_1 + \frac{1}{2}(\omega - \omega_0)^2\beta_2 + \frac{1}{6}(\omega - \omega_0)^3\beta_3 + ...,$$

$$\beta_m = (\frac{\partial^m \beta}{\partial \omega^m})_{\omega=\omega_0} \qquad m = 1, 2, 3, ...$$

De esta forma, aún en el dominio de la frecuencia y utilizando sólo hasta el segundo orden del desarrollo de Taylor, la ecuación de desplazamiento suave del pulso a través de la fibra queda:

$$\frac{\partial A(\vec{r},\omega)}{\partial z} = j[(\omega - \omega_0)\beta_1 + \frac{1}{2}(\omega - \omega_0)^2\beta_2 + \Delta\beta]A(\vec{r},\omega)$$

Se desprecian todos los términos a partir del tercer orden, éste incluido, para conservar la asunción de cuasimonocromaticidad de la ecuación, donde el ancho espectral del pulso es mucho menor a la frecuencia de portadora: $\Delta\omega \ll \omega_0$. Para

pasar al dominio del tiempo convertimos los términos $(\omega - \omega_0)$ del desarrollo en el operador $j(\frac{\partial}{\partial t})$ y realizamos la transformada inversa de Fourier de la envolvente del pulso $A(\vec{r}, \omega - \omega_0)$ de modo que nos quede como $A(\vec{r}, t)$:

$$\frac{\partial A(\vec{r},t)}{\partial z} = -\beta_1 \frac{\partial A(\vec{r},t)}{\partial t} - j\frac{\beta_2}{2}\frac{\partial^2 A(\vec{r},t)}{\partial t^2} + j\Delta\beta A(\vec{r},t)$$

Si sustituimos el valor del Δn en $\Delta\beta$, la ecuación quedará definida en función del módulo de la envolvente y del parámetro de no linealidad γ:

$$\Delta\beta = \frac{k_0 \int\int_{-\infty}^{\infty} \Delta n |F(x,y)|^2 dxdy}{\int\int_{-\infty}^{\infty} |F(x,y)|^2 dxdy}$$

$$\Delta n = n_{NL}|E|^2 + j\frac{\hat{\alpha}}{2k_0} = (n_{NL} + j\frac{\alpha_{NL}}{2k_0})|E|^2 + j\frac{\alpha}{2k_0}$$

$$\Delta\beta = j\frac{\alpha}{2} + \frac{k_0 \int\int_{-\infty}^{\infty} (n_{NL} + j\frac{\alpha_{NL}}{2k_0})|E|^2 |F(x,y)|^2 dxdy}{\int\int_{-\infty}^{\infty} |F(x,y)|^2 dxdy}$$

$$\Delta\beta = j\frac{\alpha}{2} + (k_0 n_{NL} + j\frac{\alpha_{NL}}{2})|A|^2 \frac{\int\int_{-\infty}^{\infty} |F(x,y)|^4 dxdy}{\int\int_{-\infty}^{\infty} |F(x,y)|^2 dxdy}$$

Se denomina área efectiva del núcleo de la fibra a un parámetro dependiente del radio del núcleo y de la relación entre el índice de refracción del núcleo y de la cubierta. Para una distribución gaussiana de F(x,y), $A_{eff} = \pi w^2$, donde w es el parámetro de anchura. la A_{eff} se incrementa intencionalmente para reducir el impacto de las no linealidades.

$$A_{eff} = \frac{(\int\int_{-\infty}^{\infty} |F(x,y)|^2 dxdy)^2}{\int\int_{-\infty}^{\infty} |F(x,y)|^4 dxdy}; \quad \gamma = \frac{n_{NL}\omega_0}{cA_{eff}}$$

Esta expresión puede ser sustituida en $\Delta\beta$ de modo que obtengamos, tras realizar los cambios pertinentes para ajustarla al factor γ anteriormente indicado, el valor final. Si normalizamos $|A|^2$ para que sea la potencia, dividiremos por la integral

superficial de todo el campo, de modo que compense el cuadrado del numerador en la fórmula del área efectiva. Por otra parte, podemos considerar despreciable las pérdidas no lineales en la fibra si consideramos la parte imaginaria del tensor de orden cuatro de la susceptibilidad mucho más pequeña que su parte real:

$$\Delta\beta = j\frac{\alpha}{2} + (\gamma + j\frac{\alpha_{NL}}{2A_{eff}})|A|^2 \int\int_{-\infty}^{\infty} |F(x,y)|^2 dxdy$$

$$\Delta\beta = j\frac{\alpha}{2} + (1 + j\frac{\alpha_{NL}}{n_{NL}}\frac{1}{2k_0})\gamma|A|^2 \int\int_{-\infty}^{\infty} |F(x,y)|^2 dxdy$$

$$\alpha_{NL} = \frac{3\omega_0}{4nc} Im[\chi^{(3)}]; \quad n_{NL} = \frac{3}{8n} Re[\chi^{(3)}]$$

$$\Delta\beta = j\frac{\alpha}{2} + (1 + j\frac{Im[\chi^{(3)}]}{Re[\chi^{(3)}]})\gamma|A|^2 \int\int_{-\infty}^{\infty} |F(x,y)|^2 dxdy$$

$$\Delta\beta = j\frac{\alpha}{2} + \gamma|A|^2$$

Este ya sólo ha de ser sustituido en la ecuación diferencial que rige la variación del pulso de manera suave a lo largo de la fibra:

$$\frac{\partial A}{\partial z} = -\beta_1 \frac{\partial A}{\partial t} - j\frac{\beta_2}{2}\frac{\partial^2 A}{\partial t^2} - \frac{\alpha}{2}A + j\gamma|A|^2 A$$

Esta ecuación suele denominarse ecuación diferencial no lineal de Schrödinger(NLS), puesto que puede ser reducida a su forma bajo ciertas condiciones relacionadas con las pérdidas en la fibra, caracterizadas por α, con su dispersión cromática gobernada por los parámetros β_1 *y* β_2, y con las no linealidades bajo el parámetro γ. El parámetro β_1 da una idea de la velocidad a la que viaja la envolvente del pulso óptico o velocidad de grupo:

$$\beta_1 = \frac{1}{v_g} = \frac{n_g}{c} = \frac{1}{c}(n + \omega\frac{dn}{d\omega})$$

Para una frecuencia de trabajo, o longitud de onda, se puede encontrar el índice de refracción de una determinada fibra con una serie de caracterísitcas materiales, a partir de la ecuación de Sellmeier:

$$n^2(\lambda) = 1 + \frac{B_1\lambda^2}{\lambda^2 - C_1} + \frac{B_2\lambda^2}{\lambda^2 - C_2} + \frac{B_3\lambda^2}{\lambda^2 - C_3}$$

Donde B_i *y* C_i son los coeficientes de Sellmeier, obtenidos experimentalmente y que para un cristal de silicio puro vienen a ser igual a:

$$B_1 = 0{,}696166300; \qquad C_1 = 4{,}67914826 \times 10^{-3}\mu m^2;$$

$$B_2 = 0{,}407942600; \qquad C_2 = 1{,}35120631 \times 10^{-2}\mu m^2;$$

$$B_3 = 0{,}897479400; \qquad C_3 = 97{,}9340025\mu m^2;$$

Si tenemos en cuenta estos valores, tras derivar el índice de refracción en función de la longitud de onda y sustituir en la β_1, llegamos a la siguiente expresión:

$$\beta_1 = \frac{1}{C}(n + \omega\frac{\partial n}{\partial \omega}) = \frac{1}{C}(n + \lambda\frac{\partial n}{\partial \lambda})$$

$$\beta_1 = \frac{1}{C}(n - \frac{\lambda}{n}(\frac{B_1C_1}{(\lambda^2 - C_1)^2} + \frac{B_2C_2}{(\lambda^2 - C_2)^2} + \frac{B_3C_3}{(\lambda^2 - C_3)^2}))$$

En un principio, conocer la β_1 sería suficiente para caracterizar la fibra óptica empleada en función de la longitud de onda de trabajo. Sin embargo, este parámetro suele ser tan pequeño que no se incluye en las características, incluso se omite en el desarrollo de la ecuación de Schrödinger con motivo de agilizar los cálculos.

En contrapartida, cuando tratamos con fibras birrefrigentes, donde cada eje de polarización tiene su propio índice de refracción y, por añadidura, su β_1 propia, este cambio en función de la frecuencia de trabajo da idea de la dispersión del modo de polarización (PMD), efecto que influirá significativamente en el ensan-

chamiento del pulso óptico según la siguiente relación:

$$\Delta T = \left| \frac{L}{v_{gx}} - \frac{L}{v_{gy}} \right| = L|\beta_{1x} - \beta_{1y}| = Lk_0 \frac{\partial \beta_m}{\partial \omega}$$

En esta relación, de especial interés es el índice de birrefrigencia modal β_m, que indica la diferencia de índices de refracción en ambos ejes de polarización:

$$\beta_m = \frac{|\beta_x - \beta_y|}{k_0} = |n_x - n_y|$$

Los dos modos intercambian sus potencias a lo largo de la fibra de manera periódica produciéndose un batido con un cierto periodo $L_\beta = \lambda/\beta_m$.

Estas variaciones que, en principio, deberían mantenerse constantes para las diferentes coordenadas de polarización, son irregulares debido a las fluctuaciones en la forma del núcleo de la fibra y al estrés anisotrópico.

Aunque el grado de ensanchamiento del pulso puede ser contabilizado por la variación temporal de llegada de sus dos coordenadas principales, ya que cada una tendrá una constante de propagación de primer orden diferentes, no podemos aplicar esta expresión para la estimación de la PMD debido a los cambios aleatorios en la birrefrigencia de la fibra que tienden a equalizar los tiempos de propagación de las componentes de polarización. PMD es caracterizado por el valor RMS de ΔT, obtenido tras realizar un promedio de las perturbaciones aleatorias:

$$\sigma_T^2 = (\Delta T)^2 = 2(\delta' l_c)^2 [e^{-\frac{L}{l_c}} + \frac{L}{l_c} - 1]$$

Donde Δ' es la dispersión modal intrínseca, y l_c la longitud de correlación bajo la que los dos componentes de polarización permanecen correlados. Si suponemos una longitud de la fibra mayor a cien metros, podemos considerar la longitud de correlación mucho más pequeña, ya que suele ser del orden de 10cm. En ese caso,

la expresión anterior queda reducida considerablemente:

$$\sigma_T \approx \Delta' \sqrt{2 l_c L} = D_p \sqrt{L}$$

Donde D_p es el parámetro PMD y suele ser del orden $0,1 - 1ps/\sqrt{km}$. Para algunas aplicaciones es conveniente mantener la polarización a lo largo de la fibra. En este caso, gran cantidad de birrefrigencia es introducida a través de determinadas modificaciones que enmascaren pequeñas variaciones de birrefrigencia para que no afecten al estado de polarización. Las fibras que rompen la simetría cilíndrica del núcleo consiguen grados de birrefrigencia del orden de $\beta_m \approx 10^{-6}$; sin embargo, esquemas modificados a través de estrés inducido permiten β_m del orden de 10^{-4}. Este tipo de fibra óptica que consigue preservar la polarización a lo largo de su recorrido se denomina de polarización mantenida. Para su uso se requiere la identificación del eje lento y del eje rápido antes de que una señal óptica sea introducida. Si la dirección de propagación de la luz polarizada linealmente coincide con el eje lento o rápido de la fibra, la polarización permanece constante; sin embargo, si describe un cierto ángulo con dichos ejes, la polarización variará con un periodo igual al periodo de batido $L_\beta = \frac{\lambda}{\beta_m}$. Si introducimos una nueva variable temporal que viaje a la velocidad de grupo y despreciamos las pérdidas, la ecuación diferencial no lineal de Schrödinger se reducirá a la siguiente expresión:

$$\frac{\partial A}{\partial z} = -j \frac{\beta_2}{2} \frac{\partial^2 A}{\partial t^2} + j\gamma |A|^2 A$$

Donde el parámetro β_2 representa la dispersión de la velocidad de grupo (GVD) y es responsable del ensanchamiento del pulso debido a la dispersión cromática.

$$\beta_2 = \frac{\partial \beta_1}{\partial \omega} = \frac{1}{c}(2\frac{dn}{d\omega} + \omega \frac{d^2 n}{d\omega^2})$$

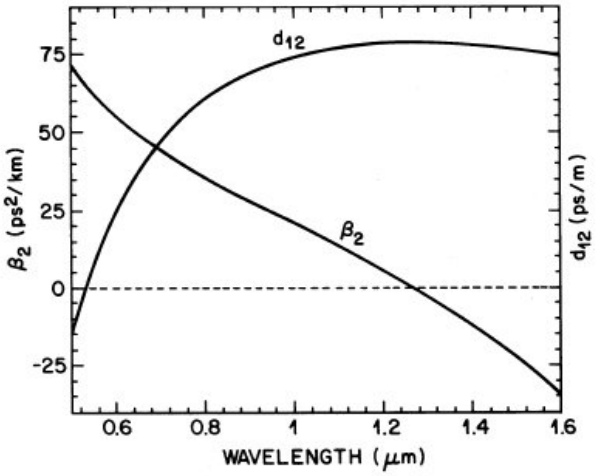


Figura A.1: Parámetros β_2 y d_{12} en función de λ. [1].

Si volvemos a relacionar este parámetro de dispersión de velocidad de grupo con la ecuación de Sellmeier para caracterizar la fibra en función de la longitud de onda, tras realizar las derivadas sucesivas del índice de refracción, obtenemos una nueva expresión:

$$\beta_2 = \frac{\lambda^3}{n(2\pi c)^2}\left[4\lambda^2\left(\frac{B_1C_1}{(\lambda^2-C_1)^3}+\frac{B_2C_2}{(\lambda^2-C_2)^3}+\frac{B_3C_3}{(\lambda^2-C_3)^3}\right)\right]\ldots$$
$$\ldots-\frac{\lambda^3}{n(2\pi c)^2}\left[\left(\frac{B_1C_1}{(\lambda^2-C_1)^2}+\frac{B_2C_2}{(\lambda^2-C_2)^2}+\frac{B_3C_3}{(\lambda^2-C_3)^2}\right)+\frac{\lambda^2}{n}\left(\frac{B_1C_1}{(\lambda^2-C_1)^2}+\frac{B_2C_2}{(\lambda^2-C_2)^2}+\frac{B_3C_3}{(\lambda^2-C_3)^2}\right)^2\right]$$

Esta expresión caracteriza el tipo de fibra a emplear con los coeficientes de Sellmeier para una determinada longitud de onda. El índice de refracción puede ser sustituido por el resultado de la ecuación de Sellmeier de modo que β_2 sólo quede relacionado por los parámetros antes indicados. De este modo, para una fibra de un cierto material, la GVD indica la longitud de onda de trabajo; y viceversa, para una determinada frecuencia de trabajo, la β_2 nos está indicando el tipo de fibra que se está utilizando.

Para conseguir un pulso con dispersión cero, debemos trabajar a longitudes de onda donde ésta se desvanezca. Esta longitud de onda es referida como de dispersión cero o λ_D y suele ser de 1,27um como puede comprobarse en la figura

1. Para longitudes de onda menores de λ_D, la β_2 será positiva o de dispersión normal. En altas frecuencias las componentes del pulso se suavizan. Cuando la longitud de onda es mayor a λ_D, la β_2 será negativa o de dispersión anómala y el ensanchamiento del pulso se producirá a bajas frecuencias. Este último régimen anómalo es de considerable interés debido a que en él la fibra óptica soporta solitones en un balance entre efectos lineales o dispersivos y no lineales.

Por el contrario, si queremos que desaparezcan los efectos no lineales producidos por la interacción de dos pulsos de diferente velocidad de grupo, conviene que la diferencia de velocidades entre ambos sea la máxima posible. El parámetro que contabiliza dicha velocidad se denomina factor de walk-off, d_{12}, y viene expresado del siguiente modo:

$$d_{12} = \beta_1(\delta_1) - \beta_1(\delta_2)$$

Si consideramos pulsos de ancho T_0, se define la longitud de walk-off a la distancia en la que se superponen ambos pulsos de un mismo periodo:

$$L_w = \frac{T_0}{|d_{12}|}$$

Para reducir esta longitud lo máximo posible se debería aumentar en la misma proporción la GVD, lo que puede ser útil para transmitir canales mucho más estrechos en sistemas DWDM.

Para contabilizar los efectos dispersivos en fibras no lineales suele utilizarse el parámetro de dispersión D:

$$D = \frac{\partial \beta_1}{\partial \lambda} = -\frac{2\pi c}{\lambda^2}\beta_2 \approx -\frac{\lambda}{c}\frac{d^2 n}{d\lambda^2}$$

En el momento en que se consigue que desaparezca β_2, a un rango de cercanía a la λ_D del orden del nanómetro, entrarían a jugar un papel importante las

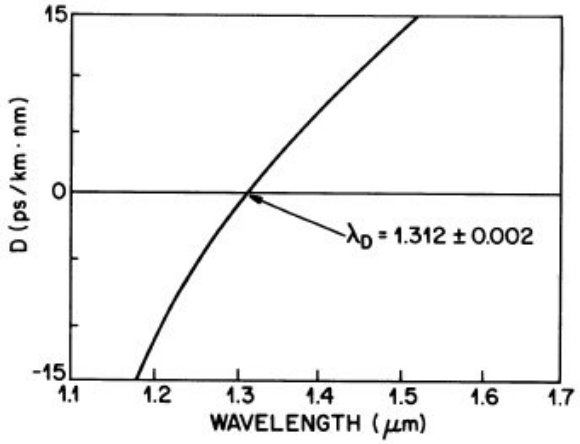


Figura A.2: Parámetro D en función de λ con TOD. [1].

dispersiones de tercer orden (TOD), β_3, haciendo que la longitud de onda de dispersión cero se desplace hasta $1,312\mu m$ en fibras monomodo, como se puede observar en la figura 2.

Sin embargo, a esta longitud de onda las pérdidas en la fibra aún son considerables y en muchas ocasiones conviene desplazarla a valores próximos a $1,55\mu m$ donde éstas se vuelven insignificantes. Las fibras que consiguen este efecto se denominan fibras de dispersión desplazada, entre las que se encuentran las fibras de dispersión cero con D=0 a $1,55\mu m$; y fibras de dispersión no zero modificada,NZD con $D \neq 0$ a $1,55\mu m$. En este último caso, cuando la dispersión cero se desplaza a $1,6\mu m$, las fibras se denominan de dispersión compensada, DCFs. Estas fibras exhiben un alto valor de GVD lo que es de gran utilidad para compensar los efectos de no linealidad descritos por el factor de walk-off d_{12}. Esto se consigue variando los parámetros de diseño de la fibra, el radio del núcleo o la diferencia de índices núcleo-cubierta.

Los efectos producidos por el GVD son mucho más acusados que el PMD inducido del pulso ensanchado. El PMD se convierte en un límite para sistemas

de alta velocidad que operen a longitudes de onda muy próximas a λ_D.

Las no linealidades dependen en gran medida de la relación entre el índice de refracción y la intensidad eléctrica $|E|^2$. El vector de polarización relacina la susceptibilidad eléctrica, la constante dieléctrica y la intensidad de campo. En función de si nos encontramos con un tensor de susceptibilidad de primero, tercero o posteriores órdenes, los efectos no lineales serán diferentes. La susceptibilidad de segundo orden no influye en los efectos no lineales. En concreto, $\chi^{(3)}_{\chi,\chi,\chi,\chi}$ son responsables de fenómenos como la generación del tercer armónico, la mezca de cuatro ondas FWM y la refracción no lineal.

Uno de los términos que influyen más directamente en la variación del parámetro no lineal γ, es la intensidad de campo dependiente del índice de refracción no lineal:

$$\hat{n}(\omega, |E|^2) = n(\omega) + n_{NL}|E|^2; n_{NL} = \frac{3}{8n} Re[\chi^{(3)}_{\chi,\chi,\chi,\chi}]$$

Los efectos no lineales dependientes del índice de radiación más extendidos son:

SPM, modulación de fase inducida, $\phi = \hat{n}k_0L = (n(\omega) + n_{NL}|E|^2)k_0L$. Es responsable del ensanchamiento espectral de pulsos ultracortos y de la formación de solitones ópticos en régimen de dispersión anómala.

XPM, modulación de fase cruzada, referida al cambio de fase no lineal de un campo óptico inducido por otro campo con una longitud de onda, una dirección o un estado de polarización diferentes. El cambio de fase no lineal es dado por:

$$\phi_{NL} = n_{NL}k_0L[|E_1|^2 + 2|E_2|^2]$$

La contribución de XPM al cambio de fase no lineal es dos veces la de SPM. Es responsable del ensanchamiento espectral asimétrico.

Dentro de los efectos no lineales, aquellos no referidos al índice de refracción no lineal, dependerán directamente de la constante de pérdidas o del traslado de

energía de la onda a la fibra. Los efectos no lineales elásticos se relacionan con el intercambio de energía entre el campo electromagnético y el medio dieléctrico; los inelásticos, por el contrario, están relacionados con el traslado de energía del campo óptico al medio no lineal, sílice. A estos últimos se les denomina Dispersión Inelástica Estimulada y son responsables de dos fenómenos:

-SRS o Dispersión de Raman Estimulada, cuyos responsables son los fonones ópticos y que se producen en ambos sentidos de propagación de la fibra. $I_p^{th} = 16(\frac{\alpha}{g_r})$. El crecimiento incial de la onda de Stokes viene definida por: $\frac{dI_s}{dz} = g_r I_p I_s$

Donde I_s es la intensidad de Stokes, I_p es la intensidad de bombeo y g_r es el coeficiente de Raman.

-SBS o Dispersión de Brillouni Estimulada, causada por los phonones acústicos y que se producen sólo en la dirección hacia atrás. $I_p^{th} = 21(\frac{alpha}{g_b})$.

Estos parámetros son como mínimo dos órdenes de magnitud menores a las otras medidas no lineales vistas anteriormente. Por este motivo suelen ser despreciados cuando se escribe la ecuación no lineal de Schrödinger. Esto se debe principalmente a dos causas, un pequeño tamaño del pulso del orden de $10\mu m$ y unas pérdidas extremadamente reducidas ($\ll 1$ dB/km) en el rango de longitudes de onda 1-1.6μm.

Apéndice B

Análisis Vectorial de la Ecuación de Schrödinger. Polarización

[1]

Si bien hasta ahora se ha supuesto que el modo de polarización se conserva a lo largo de la propagación del pulso a lo largo de la fibra, la realidad es justo la contraria. Exite una variación del modo de polarización que afecta significativamente a la forma y características del pulso en función del acoplamiento entre los dos componentes ortogonales de polarización, el rápido y el lento. El efecto responsable de dicho acople es el de modulación de fase cruzada, XPM, siempre acompañado del de autofase, SPM. Estos efectos pueden ocurrir tanto para ondas ópticas de una misma longitud de onda, componentes ortogonales de un mismo pulso, como para longitudes de onda diferentes como ocurre en los casos no degenerados.

Como se mencionó en secciones anteriores, las modificaciones en la forma y estrés anisotrópico en la fibra a lo largo de su trayectoria, provocan diferencias en el índice de refracción según el eje de polarización, $n_x \neq n_y$, lo que se viene

a llamar Birrefrigencia Lineal. El grado de birrefrigencia viene indicado por su índice $B_m = |n_x - n_y|$. Cuando los efectos no lineales se hacen importantes, un campo óptico lo suficientemente intenso puede inducir un tipo de birrefrigencia no lineal cuya magnitud es directamente dependiente de la intensidad. Si asumimos una constante modal de birrefrigencia constante con dos ejes de polarización mantenidos a lo largo de la fibra, denominados eje rápido y eje lento, $n_x > n_y$, para altas potencias, la luz de onda continua es lanzada con una dirección de polarización orientada en un ángulo con respecto a dichos ejes, el estado de polarización varía a lo largo de la propagación con un periodo mantenido llamado longitud de batido.

$$L_B = \frac{2pi}{|\beta_x - \beta_y|} = \frac{\lambda}{\beta_m}$$

Esta longitud de batido puede ser del orden de 1cm para fibras de alta birrefrigencia, $B_m \approx 10^{-4}$, hasta de 1m para fibras de baja birrefrigencia, $B_m \approx 10^{-6}$. La expresión que describe el campo óptico con polarización arbitraria para ondas cuasiplanas donde la componente en el eje de propagación, E_z, es considerado despreciable:

$$\vec{E}(\vec{r}, t) = \frac{1}{2}(\hat{x}E_x + \hat{y}E_y)e^{-j\omega_0 t} + c.c.,$$

viene descrita para dos componentes de polarización de amplitudes complejas en ambos ejes, rápido y lento, y una frecuencia de portadora ω_0. La parte no lineal del vector de polarización P_{NL} es expresada del siguiente modo:

$$\hat{P}_{NL}(\hat{r}, t) = \frac{1}{2}(\hat{x}P_x + \hat{y}P_y)e^{-j\omega_0 t} + c.c.,$$

donde las componentes en los ejes vienen dados por la siguiente expresión:

$$P_i = \frac{3\varepsilon_0}{4} \sum j(\chi^{(3)}_{xxyy} E_i E_j E_j * + \chi^{(3)}_{xyxy} E_j E_i E_j * + \chi^{(3)}_{xyyx} E_j E_j E_i *)$$

con i,j como x e y, y la susceptibilidad magnética de tercer orden con tres componentes independientes entre sí para medios isotrópicos de los ochenta y uno de los que dispone:

$$\chi^{(3)}_{ijkl} = \chi^{(3)}_{xxyy}\delta_{ij}\delta_{kl} + \chi^{(3)}_{xyxy}\delta_{ik}\delta_{jl} + \chi^{(3)}_{xyyx}\delta_{il}\delta_{jk}$$

en la que las δ_{ij} se corresponden con la función delta de Kronecker, igual a 1 cuando i=j y 0 en otro caso. De modo, que si consideramos los cuatro componentes iguales a x, como en el caso de definición del índice de refracción no lineal del caso escalar, podemos reducir esta expresión a:

$$\chi^{(3)}_{xxxx} = \chi^{(3)}_{xxyy} + \chi^{(3)}_{xyxy} + \chi^{(3)}_{xyyx}$$

En el caso de la fibra de silicio, la contribución dominante es de origen electrónico y los tres componentes toman magnitudes muy similares. Si las asumimos como idénticas, los componentes de polarización quedan reducidos a estas dos ecuaciones:

$$P_x = \frac{3\varepsilon_0}{4}\psi^{(3)}_{xxxx}\left[\left(|E_x|^2 + \frac{2}{3}|E_y|^2\right)E_x + \frac{1}{3}(E_x * E_y)E_y\right]$$

$$P_y = \frac{3\varepsilon_0}{4}\psi^{(3)}_{xxxx}\left[\left(|E_y|^2 + \frac{2}{3}|E_x|^2\right)E_y + \frac{1}{3}(E_y * E_x)E_x\right]$$

Si escribimos $P_j = \varepsilon_0\varepsilon^{NL}_j E_j$ con $\varepsilon_j = \varepsilon^L_j + \varepsilon^{NL}_j = (n^2_j + \Delta n_j)^2$. La contribución no lineal al índice de refracción tanto para la Ex como para la Ey se pueden expresar como:

$$\Delta n_x = n_{NL}\left(|E_x|^2 + \frac{2}{3}|E_y|^2\right)$$

$$\Delta n_y = n_{NL}\left(|E_y|^2 + \frac{2}{3}|E_x|^2\right)$$

El primer término de la expresión es responsable de la modulación de autofase, el segundo término es responsable de la modulación de fase cruzada, XPM, porque la fase no lineal adquirida por un componente de polarización depende de la intensidad de campo del otro componente, lo que induce un acoplamiento no lineal entre ambos. En general la contribución no lineal al índice de refracción varía en ambos términos, lo que crea una birrefrigencia no lineal que se manifiesta con una rotación de la elipse de polarización.

El mismo método de separación de variables utilizado para obtener la ecuación escalar de propagación del pulso puede ser utlizada para describir la evolución de las dos componentes de polarización a lo largo de la fibra. Para ello se ha de asumir que los efectos no lineales no afectan a la fibra significativamente:

$$E_j(\hat{r},t) = F(x,y)A_j(z,t)e^{j\beta_{0j}z}$$

Se buscan las A_j que satisfagan el siguiente conjunto de ecuaciones de modos acoplados:

$$\frac{\partial A_x}{\partial z} + \beta_{1x}\frac{\partial A_x}{\partial t} + j\frac{\beta_2}{2}\frac{\partial^2 A_x}{\partial t^2} + \frac{\alpha}{2}A_x = j\gamma\left(|A_x|^2 + \frac{2}{3}|A_y|^2\right)A_x + j\frac{\gamma}{3}A_x^* A_y^2 e^{-j\Delta\beta z}$$

$$\frac{\partial A_y}{\partial z} + \beta_{1y}\frac{\partial A_y}{\partial t} + j\frac{\beta_2}{2}\frac{\partial^2 A_y}{\partial t^2} + \frac{\alpha}{2}A_y = j\gamma\left(|A_y|^2 + \frac{2}{3}|A_x|^2\right)A_y + j\frac{\gamma}{3}A_y^* A_x^2 e^{-j\Delta\beta z}$$

donde $\Delta\beta = \beta_{0x} - \beta_{0y} = (2pi/\lambda)B_m = 2pi/L_B$.

Dado que las constantes de propagación de primer orden son diferentes para ambos componentes, las velocidades de grupo serán también diferentes en esta birrefrigencia modal. Por el contrario, las de segundo orden, las de dispersión por velocidad de grupo, son las mismas en ambos casos, al igual que el coeficiente de no linealidad, de modo que se tiene una misma longitud de onda al tratarse del mismo pulso. Si se tratara de pulsos diferentes a longitudes de ondas distintas,

los parámetros de dispersión de velocidad de grupo y de no linealidad también serían diferentes.

El último término del margen derecho de las ecuaciones es debido al acoplamiento coherente de los dos componentes de polarización, lo que conduce a la denominada mezcla de cuatro ondas. En fibras de alta birrefrigencia, donde $L_B \approx 1cm$, este término puede ser despreciado, sin embargo, en las de baja birrefrigencia ha de ser tenido en cuenta, sobre todo en las de corta longitud.

Para fibras de baja birrefrigencia, es conveniente reescribir las ecuaciones de propagación en función de las componentes de polarización circular hacia derecha, A_+ y hacia izquierda, A_-, considerando $\Gamma = \Delta\beta z$ [10]:

$$\begin{pmatrix} A_+ \\ A_- \end{pmatrix} = \begin{pmatrix} cos\psi & -sen\psi \\ sen\psi & cos\psi \end{pmatrix} \begin{pmatrix} e^{\frac{-j\Gamma}{2}} & 0 \\ 0 & e^{\frac{j\Gamma}{2}} \end{pmatrix} \begin{pmatrix} cos\psi & sen\psi \\ -sen\psi & cos\psi \end{pmatrix} \begin{pmatrix} A_x \\ A_y \end{pmatrix}$$

$$\begin{pmatrix} A_+ \\ A_- \end{pmatrix} = \begin{pmatrix} cos\frac{\Gamma}{2} + jsen\frac{\Gamma}{2}cos2\psi & -jsen\frac{\Gamma}{2}sen2\psi \\ -jsen\frac{\Gamma}{2}sen2\psi & cos\frac{\Gamma}{2} - jsen\frac{\Gamma}{2}cos2\psi \end{pmatrix} \begin{pmatrix} A_x \\ A_y \end{pmatrix}$$

Que podemos hacer coincidir con ángulos $\Gamma = \frac{\pi}{2}$ y $\psi = \frac{\pi}{4}$ para obtener la siguiente matriz de cambio [9]:

$$\begin{pmatrix} A_+ \\ A_- \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -j \\ -j & 1 \end{pmatrix} \begin{pmatrix} A_x \\ A_y \end{pmatrix}$$

Resultando las siguientes expresiones:

$$\frac{\partial A_+}{\partial z} + \beta_1 \frac{\partial A_+}{\partial t} + j\frac{\beta_2}{2}\frac{\partial^2 A_+}{\partial t^2} + \frac{\alpha}{2}A_+ = \frac{j}{2}(\Delta\beta)A_- + j\frac{2\gamma}{3}\left(|A_+|^2 + 2|A_-|^2\right)A_+$$

$$\frac{\partial A_-}{\partial z} + \beta_1 \frac{\partial A_-}{\partial t} + j\frac{\beta_2}{2}\frac{\partial^2 A_-}{\partial t^2} + \frac{\alpha}{2}A_- = \frac{j}{2}(\Delta\beta)A_+ + j\frac{2\gamma}{3}\left(|A_-|^2 + 2|A_+|^2\right)A_-$$

Para fibras elípticamente birrefrigentes las ecuaciones de modos acoplados son modificadas considerablemente, para lo que se han de considerar los vectores unitarios de la elipse que sustituyan en la ecuación de intensidad de campo electríco a los de las coordenadas cartesinas $\hat{x}, \hat{y}$:

$$\bar{E}(\bar{r},t) = \frac{1}{2}(\hat{e}_x E_x + \hat{e}_y E_y)e^{-j\omega_0 t} + c.c.$$

$$\hat{e}_x = \frac{\hat{x} - jr\hat{y}}{\sqrt{1+r^2}}, \quad \hat{e}_y = \frac{-jr\hat{x} + \hat{y}}{\sqrt{1+r^2}}$$

Los parámetros r representan la elipticidad introducida por la torsión del diseño y viene representado por el ángulo de elipticidad $r = tan\ \Gamma/2$. Cuando el ángulo es igual a 0 ó $\pi/2$ se corresponde respectivamente con la polarización lineal y circular.

Teniendo en cuenta este cambio de coordenadas, para realizar el cambio de base que tenga en cuenta el giro y la elipticidad de la polarización, multiplicamos la matriz del cambio por la matriz de giro:

$$\begin{pmatrix} A_x \\ A_y \end{pmatrix} = \begin{pmatrix} cos\psi & -sen\psi \\ sen\psi & cos\psi \end{pmatrix} \begin{pmatrix} cos\frac{\Gamma}{2} & -jsen\frac{\Gamma}{2} \\ -jsen\frac{\Gamma}{2} & cos\frac{\Gamma}{2} \end{pmatrix} \begin{pmatrix} E_x \\ E_y \end{pmatrix}$$

$$\begin{pmatrix} A_x \\ A_y \end{pmatrix} = \begin{pmatrix} cos\frac{\Gamma}{2}cos\psi + jsen\frac{\Gamma}{2}sen\psi & -cos\frac{\Gamma}{2}sen\psi - jsen\frac{\Gamma}{2}cos\psi \\ -jsen\frac{\Gamma}{2}cos\psi + cos\frac{\Gamma}{2}sen\psi & cos\frac{\Gamma}{2}cos\psi - jsen\frac{\Gamma}{2}sen\psi \end{pmatrix} \begin{pmatrix} E_x \\ E_y \end{pmatrix}$$

Si consideramos fibras ópticas de gran birrefrigencia donde la longitud de batido L_B es mucho más pequeña que la longitud de la fibra, la propagación de los pulsos ópticos son gobernados por el siguiente conjunto de ecuaciones acopladas [9]:

$$\frac{\partial A_x}{\partial z} + \beta_{1x}\frac{\partial A_x}{\partial t} + j\frac{\beta_2}{2}\frac{\partial^2 A_x}{\partial t^2} + \frac{\alpha}{2}A_x = j\gamma(|A_x|^2 + B|A_y|^2)A_x$$

$$\frac{\partial A_y}{\partial z} + \beta_{1y}\frac{\partial A_y}{\partial t} + j\frac{\beta_2}{2}\frac{\partial^2 A_y}{\partial t^2} + \frac{\alpha}{2}A_y = j\gamma(|A_y|^2 + B|A_x|^2)A_y$$

Que representan una extensión de la ecuación no lineal de Schrödinger considerando los efectos de polarización y son referidas como las ecuaciones acopladas NLS. La B es el parámetro de acoplamiento y depende del ángulo de elipticidad Γ. Puede variar desde 2/3 a 2 cuando el ángulo varía de 0 a pi/2, es decir, de birrefrigencia lineal a birrefrigencia circular.

$$B = \frac{2 + 2sin^2\Gamma}{2 + cos^2\Gamma}$$

Apéndice C

Archivos y funciones de PyOFTK

core.py Este archivo implementa quince clases diferentes en las que se definen un grupo de funciones relacionadas con la emisión con láser y propagación en fibra óptica, necesarias para el desarrollo de los ejemplos realizados en la librería. Para su elaboración se importan varias clases y funciones de librerías previas:

from numpy import *

import matplotlib.pyplot as pl

from PyOFTK.utilities import *

import scipy.interpolate as itrp

from pygraph.classes.graph import graph

from pygraph.classes.digraph import digraph

from pygraph.algorithms.searching import breadth_first_search

from pygraph.readwrite.dot import write

from pygraph.mixins.basegraph import basegraph

Las clases definidas en este archivo son las siguientes:

class OFTKDevice -¿representa un dispositivo genérico de PyOFTK indicando su tipo y realizando una descripción del mismo.

class OFTKExperiment(digraph, basegraph) -¿representa un experimento OFTK, que no es más que un grupo de dispositivos OFTK. las funciones que contiene se encargan de inicializar el experimento, informar sobre los dispositvos que contiene, añadir nuevos dispositivos al experimento, conectar entre sí dos dispositivos distintos, unir dos experimentos en una nueva instancia, generar una imagen del expermiento.

class eFieldSVEA -¿representa la envoltura SVEA (slowly varying envelope aproximation) del campo.

class cwLaser(OFTKDevice) -¿representa un laser cw, de onda continua. Se define la longitud de onda, la potencia y el ancho de la línea del haz laser.

class pumpLaser(cwLaser) -¿representa un laser pump, o de bombeo. Se definen la longitud de onda, la potencia y el ancho de línea.

class pulsedSource(OFTKdevice) -¿representa una fuente de luz continua. Define la longitud de onda, la potencia, la razón de repetición de la fuente y la forma temporal de los pulsos.

class FibreStepIndex(OFTKDevice) -¿representa una fibra de salto de índice. Define el radio del core, el radio de la cubierta, la concentración de Germanio del core y de la cubierta y la longitud de la fibra.

class Fiber() -¿representa una fibra óptica de ejes simétricos con implementación de índice arbitrario. Se define el índice radial de la fibra y longitud de la fibra.

class couple1x2(FibreStepIndex) -¿acoplador de 1x2, no implementado.

class YbDopedFiber(FibreStepIndex) -¿representa un fibra de salto de índice dopada de Ytterbium. Se implementa desde la clase FibreStepIndex.

class YbDopedDCOF(YbDopedFiber) -¿representa una fibra de salto de índice con doble capa y dopada de Ytterbium. Hereda desde la clase YbDopedFiber.

class ErDopedFiber(FibreStepIndex) -¿representa una fibra de salto de índice dopada de Erbio. Hereda de FibreStepIndex.

class fbg(FibreStepIndex) -¿representa una fibra bragg de rejilla en una fibra de salto de índice. Hereda de FibreStepIndex.

class mirror(FibreStepIndex) -¿representa un espejo con cierta reflectividad, no implementado.

class simpleFBG(FibreStepIndex) -¿representa una fibra de bragg de rejilla uniforme en una fibra de salto de indice. Hereda de FibreStepIndex.

class apodizeddFBG(FibreStepIdewx) -¿representa una fibra de bragg de rejilla apodizada en una fibra de salto de índice. Hereda de FibreStepIndex.

class apodizedFBGv(FibreStepIndex) -¿primeras tres funciones de la anterior.

utilities.py En este archivo se desarrollan cincuenta y ocho funciones para la generación de pulsos, utilización de filtros, manipulación y creación de arrays y vectores, etc. necesarias para el desarrollo de los ejemplos propuestos en la librería. Las funciones son muy variadas y no tienen una temática específica. No hay ninguna clase implementada.

ssf.py función que realiza el método split-step de Fourier tanto para campos vectoriales como escalares. Para ello utiliza una serie de clases y funciones pertenecientes a varias de las librerías detalladas previamente.

```
from numpy import *
import numexpr as ne
from scipy import *
from scipy.misc import factorial
from PyOFTK.utilities import *
import scipy.fftpack as fftpack
import ssprop
from pyfft.cuda import Plan
```

from pycuda.tools import make_default_context

import pycuda.gpuarray as gpuarray

import pycuda.driver as cuda

from pycuda.compiler import SourceModule

from pycuda.elementwise import ElementwiseKernel

from pycuda.reduction import ReductionKernel

Este es el único archivo que hace uso, de partida, de la librería pycuda, es decir, que realiza operaciones en paralelo sirviéndose de la gpu. Para ello implementa una serie de funciones semejantes a las realizadas previamente para programación lineal. Utiliza la librería obsoleta pyfft que ha sido sustituida por reikna en la elaboración de nuestro trabajo. Las funciones que se describen en este archivo son las siguientes:

def vector(u0x, u0y, dt, dz, nz, alphaa, alphab, betaa, betap, gamma, psp, maxiter, tol, psi, chi, allStep = False): en esta función se lleva a cabo el salto de banda o split del método split step para la resolución de la ecuación diferencial no lineal de Shrodinguer en dos coordenadas espaciales. Para ello llama en cada salto a la función ssprop.vector(), donde se realiza el método en cuestión para cada una de estas bandas.

def scalar(u0, dt, dz, nz, alpha, betap, gamma, tr = 0, to = 0, maxiter = 4, tol = 1e-5): realiza el método split step para la resolución de la ecuación diferencial no lineal de Shrodinger en una coordenada espacial, llamando a la función ssprop.scalar(). Esta función, como incremento espacial nz se le introduce por parámetro un número distinto a 1, y por tanto realiza tanto el split como el step del método, al contrario de lo que sucedía en la anterior.

def ssf(u0, dt, dz, nz, alpha, betap, gamma, maxiter = 4, tol = 1e-5, phiNLOut = False): realiza la implementación del método split-step de Fourier simetrizado para cualquier ecuación diferencial no lineal del tipo de la de Shrodinger.

def ssf_b(u0, dt, dz, nz, alpha, betap, gamma, maxiter = 4, tol = 1e-5, phiNLOut = False): mismo caso que el anterior pero sin fase adicional.

def ssfu(up, dt, dz, nz, alpha, betap, gamma, maxiter = 4, tol = 1e-5): implementa el método split-step de Fourier no simetrizado para cualquier ecuación diferencial no lineal del tipo de la de Shrodinger. La diferencia a grandes rasgos con respecto a las anteriores, es que no realiza avances de medias unidades espaciales entre step y step, por lo que el error será mayor y la precisión menor.

def ssfgpuStd(u0, dt, dz, nz, alpha, betap, gamma, maxiter = 4, tol = 1e-5, phiNLOut = False): realiza la misma operación que el ssf() pero utilizando para ello la programación en paralelo sirviéndose de la gpu de la tarjeta gráfica por medio de la librería PyCUDA.

def ssfgpu(u0, dt, dz, nz, alpha, betap, gamma, maxiter = 4, tol = 1e-5, context = False, phiNLOut = False, fullOutput=False): selecciona entre las tres funciones ssfgpu a partir de varios parámetros booleanos de entrada, de tal modo de que si el context es false, realiza ssfgpuStd que no retorna el array fase. Si context es true compara el parámetro fullOutput para ver si saca la salida completa, con fase y argumento, en cuyo caso llamará a ssfgpuFull(), o sólo saca la fase a través de la función ssfgpu2().

def ssfgpu2(u0, dt, dz, nz, alpha, betap, gamma, context, maxiter = 4, tol = 1e-5, phiNLOut = False): junto con el vector solución de la ecuación diferencial, también saca la fase. Según viene implementada la función en la librería, no hay distinción alguna entre esta función y la ssfgpuStd().

def ssfgpuFull(u0, dt, dz, nz, alpha, betap, gamma, context, maxiter = 4, tol = 1e-5, phiNLOut = False): junto con el vector solución de la ecuación diferencial, también saca la fase y un vector de argumentos como cuadrado de las soluciones parciales.

pulse En este archivo se implementan varias funciones que simulan distintos tipos de pulsos, en concreto son once pulsos:

def pulseResampling(u_in, newSize): pulso de remuestreo.

def parabolicPulseFit(t, FWHM, T0, P0): hace un ajuste parabólico en una curva con forma de pulso.

def parabolicPulse(t, P0): realiza una pulso parabólico de una determinada potencia en una escala de tiempos.

def gaussianPulse(t, FWHM, t0, P0=1.0, m=1, C=0): retorna un pulso gaussiano a partir de la función de Euler de números complejos. Para ello se introduce un vector de línea de tiempos, el ancho del pulso a intensidad media, el centro del pulso, la potencia central que se convertirá en intensidad con tal de hallar su raíz cuadrada, el orden gaussiano que por defecto será 1 y un parámetro de chirp para indicar el nivel de ruido del pulso y que, por defecto, para obtener sólo la envolvente, será de cero.

def sechPulse(t, FWHM, t0, P0, C): computa un pulso secante hiperbólica con parámetros especificados.

def solitonPulse(t,t0,epsilon,N): función generadora de un solitone.

def sechPulseCmpt(t, p): misma función que la del pulso secante hiperbólico compactada.

def gaussianPulseCmpt(t, p): misma función que la generadora de un pulso gaussiano en versión compactada.

def parabolicPulseCmpt(t, t0, P0): genera la envolvente de un pulso parabólico.

def parabolicPulseFitCmpt(t, p): misma función compacta de la que genera un ajuste parabólico de una curva como un pulso.

def parabolicPulseFitCmpt2(t, p):misma función a la anterior.

A parte de los archivos escritos en Python directamente, hay dos carpetas

adicionales, ssprop y freespace, con archivos escritos en Python, C y C++. En estas carpetas se utilizan archivos originales programados en C y, a través de wraps o envolturas, se interpretan en Python sin necesidad de traducirlos de un idioma a otro. Además de todo lo anterior, hay una carpeta con diferentes ejemplos de simulaciones realizadas en Python para los que se utilizan todas las funciones y clases pertenecientes a la librería de Python.

Apéndice D

Archivos para las simulaciones.

- Archivo agrawal_fig3_1_ssf.py para primer régimen de propagación:

```
import numpy as np
import numexpr as ne
from scipy import *
from pylab import *
from PyOFTK.utilities import *
import PyOFTK
T = 3200.0
nt = 614400
aa = 1
betap = array([0.0,0.0,20.0])
while aa <= 500:
   dt = T/nt
   t = linspace(-T/2.0, T/2.0, nt)
   u_ini_x = PyOFTK.gaussianPulse(t,166.5,0.0,0.667.0,1.0,0.0)
   u_out_x = PyOFTK.ssf(u_ini_x, dt, dz=2.0, nz, alpha=0.0, betap, 0.003, maxiter=10,
1e-5, phinLOut = False, down1 = True)
   plot(t, pow(abs(u_out_x),2), 'r', t, pow(abs(u_ini_x),2), 'b')
   ylabel("|u(z,T)|^2")
   xlabel("T/T_0")
   xlim([-16,16])
   grid(True)
   savefig('figura_' + str(aa) + '.png')
   clf()
   aa += 1
```

- Archivo agrawal_sellmeier.py para calcular la β_2 en función del índice de refracción hallado a través de la ec. de Sellmeier:

```
from numpy import *
import numpy as np
import numexpr as ne
from scipy import *
from pylab import *
from time import *
import PyOFTK
from PyOFTK.utilities import *
B1 = 0.69616630
B2 = 0.40794260
B3 = 0.89747940
C1 = pow(0.068404300,2)
C2 = pow(0.11624140,2)
C3 = pow(9.8961610,2)
T = 4000.0
nt = pow(2,15)
nz = 400
aa = 1
dz = 2.0
dt = T/nt
maxiter = 10
t = linspace(-T/2, T/2, nt)
lamb = 0.4 #um
c=3e14 #um/s
beta22=[]
lamb22=[]
disper22=[]
beta11=0.0
while lamb <= 1.6 :
   n = sellmeier(5,lamb)
   #print n
   bet = (pow(lamb,3) / (n*pow(c*2*pi,2)))
   bet2 = 4*pow(lamb,2)*((B1*C1)/pow((pow(lamb,2)-C1),3) + (B2*C2)/pow((pow(lamb,2)-C2),3)
+ (B3*C3)/pow((pow(lamb,2)-C3),3))
   bet3 = ((B1*C1)/pow((pow(lamb,2)-C1),2) + (B2*C2)/pow((pow(lamb,2)-C2),2) + (B3*C3)/
   pow((pow(lamb,2)-C3),2))
   bet4 = (pow(lamb,2)/n) * pow(bet3,2)
   beta2 = bet * (bet2 - bet3 - bet4)
```

```
            disper = - (2*pi*c/pow(lamb,2))*beta2 #s/um·nm
            beta2 = beta2*pow(10,33) #ps$\hat{2}$/km
            disper = disper*pow(10,19) #ps/km nm
            beta22 = np.append(beta22, beta2)
            lamb22 = np.append(lamb22,lamb)
            disper22 = np.append(disper22,disper)
            print beta2
            print disper
            lamb += 0.02
        plot(lamb22,beta22)
        ylabel("$beta_2(ps^2/km)$")
        xlabel("$\lambda(\mu m)$")
        grid(True)
        show()
        plot(lamb22[40:60],disper22[40:60])
        ylabel("$D(ps/kmnm)$")
        xlabel("$\lambda(\mu m)$")
        grid(True)
        show()
        betap = array([0.0,0.0,beta22[43]])
        alpha = 0.0
        T0 = 100.0
        FWHM = T0*2*sqrt(log(2))
        P0 = 0.667
        gamma = 0.003
        u_ini_x = PyOFTK.gaussianPulse(t, FWHM, 0.0, P0, 1.0, 0.0)
        u_out_x = PyOFTK.ssf(u_ini_x, dt, dz, nz, alpha, betap, gamma, maxiter, 1e-5, phiNLOut
= False, down1 = True)
        plot(t, pow(abs(u_ini_x),2), t, pow(abs(u_out_x),2))
        show()
```

- Archivo agrawal_fig3_1_ssf.py para segundo régimen de propagación:

```
import numpy as np
import numexpr as ne
from scipy import *
from pylab import *
from PyOFTK.utilities import *
import PyOFTK
T = 32.0
nt = pow(2,10)
aa = 200
dz = 0.002
nz = 2
maxiter = 10
betap = array([0.0,0.0,20.0])
alpha = 0.0
while aa <= 500:
   dt = T/nt
   t = linspace(-T/2.0, T/2.0, nt)
   u_ini_x = PyOFTK.gaussianPulse(t, 1.665, 0.0, 0.667, 1.0, 0.0)
   u_out_x = PyOFTK.ssf(u_ini_x, dt, dz, nz, alpha, betap, 0.003, maxiter, 1e-5, phinLOut
= False, down1 = True)
   plot(t, pow(abs(u_out_x),2), 'r', t, pow(abs(u_ini_x),2), 'b')
   ylabel("|u(z,T)|^2")
   xlabel("T/T_0")
   xlim([-16,16])
   grid(True)
   savefig('figura_' + str(aa) + '.png')
   clf()
   aa += 1
```

- Archivo agrawal_fig3_1_ssf.py para tercer régimen de propagación:

```
import numpy as np
import numexpr as ne
from scipy import *
from pylab import *
from PyOFTK.utilities import *
import PyOFTK
T = 320.0*2
nt = 61440*2
aa = 1
dz = 0.002
maxiter = 10
tiemp = linspace(1200. 1200. 2)
t_menos = linspace(-1600, -T/2.0, nt*(1600-T/2)/T)
t_zeros = np.zeros((len(t_menos)))
betap = array([0.0,0.0,20.0])
alpha = 0.0
while aa <= 500:
   dt = T/nt
   t = linspace(-T/2.0, T/2.0, nt)
   u_ini_x = PyOFTK.gaussianPulse(t, 166.5, 0.0, 667, 1.0, 0.0)
   u_out_x = PyOFTK.ssf(u_ini_x, dt, dz, nz, alpha, betap, 0.003, maxiter, 1e-5, phinLOut
= False, down1 = True)
   u_ini_x1 = np.concatenate([t_zeros, u_ini_x])
   u_ini_x1 = np.concatenate([t_ini_x1, t_zeros])
   u_out_x1 = np.concatenate([t_zeros, t_out_x])
   u_out_x1 = np.concatenate([t_out_x1, t_zeros])
   t1 = linspace(-1600,1600,int(3200/dt))
   plot(t1, pow(abs(u_out_x),2), 'r', t1, pow(abs(u_ini_x),2), 'b')
   ylabel("|u(z,T)|^2")
   xlabel("T/T_0")
   xlim([-16,16])
   grid(True)
   savefig('figura_' + str(aa) + '.png')
   clf()
   aa += 1
```

-Archivo agrawal_factor_element.py para calcular los factores de ensanche en la gpu para tercer régimen de propagación en fibra óptica:

```
from numpy import *
import numpy as np
import numexpr as ne
from scipy import *
from pylab import *
from time import *
import reikna.cluda as cluda
from pycuda.reduction import ReductionKernel
import PyOFTK
T = 340.0*2
aa= 0
dz = np.float32(0.002)
nt = np.int32(pow(2,17))
t = linspace(-T/2.0, T/2.0, nt)
dt = T/nt
t1=np.ones((nt), dtype= np.complex64)
for i in arange(nt):
   if t[i] <0: t1[i] = 1.0j*sqrt(abs(t[i]))
   else: t1[i] = sqrt(t[i])
maxiter = 10
tiemp = linspace(1200, 1200, 2)
sigma0_tot = []
factor01_tot = []
factor_tot = []
factor1_tot = []
factor2_tot = []
t_menos =linspace(-1600, -T/2.0, nt*(1600-T/2)/T)
t_zeros = np.zeros((len(t_menos)))
betap = array([0.0,0.0,20.0,0.0])
beta2 = np.float32(betap[2])
alpha = 0.0
gamma = 0.003
P0= 667.0
C=1.0
u_ini_x = PyOFTK.gaussianPulse(t,166.,0.0,667.0,1,1.0)
T00 = (166.5/1.665) #FWHM/1.665
LD = pow(T00,2)/abs(beta2)
sigma0 = np.float32(0.0)
```

```
        sigma = np.float32(0.0)
        sigma01 = np.float32(0.0)
        api = cluda.cuda_api()
        thr = api.Thread.create()
        t_gpu = thr.to_device(t.astype(np.complex64))
        tsqrt_gpu = thr.to_device(t1.astype(np.complex64))
        u_inix_gpu = thr.to_device(u_ini_x.astype(np.complex64))
        sigma0_gpu = thr.to_device(sigma0)
        sigma01_gpu = thr.to_device(sigma01)
        sigma_gpu = thr.to_device(sigma)
        sumas_parciales = ReductionKernel(np.float32, neutral="0",
           reduce_expr=.a+b", map_expr="pow(abs(a[i]*b[i]),2)",
           arguments="pycuda::complex<float>*a, pycuda::complex<float>*b",
           name=.error_reduction",
           preamble="#include <pycuda-complex.hpp>")
        unos = np.ones((1,nt), dtype=np.complex64)
        uno = thr.to_device(unos.astype(np.complex64))
        rms = sumas_parciales(u_inix_gpu, t_gpu).get()
        Nc = sumas_parciales(uno, u_inix_gpu).get()
        cuad = sumas_parciales(u_inix_gpu, tsqrt_gpu).get()
        sigma0 = sqrt(abs(rmsNc - pow((cuadNc),2)))
        seg = 2*pow(sigma0,2)
        orden3 = (pow(1 + pow(C,2),2)) * 1/2*pow(betap[3]*dz/(4*pow(sigma0,3)),2)
        while aa <= 20000:
           aa = np.int32(aa)
           #u_out_x = PyOFTK.ssf(u_ini_x, dt, dz, aa, alpha, betap, gamma, maxiter, 1e-5,
phiNLOut = False, down1 = True)
           u_out_x = PyOFTK.ssfgpu(u_ini_x, dt, dz, aa, alpha, betap, gamma, maxiter, 1e-5,
context = False, phiNLOut = False, fullOutput = False)
           u_outx_gpu = thr.to_device(u_out_x.astype(np.complex64))
           rms1 = sumas_parciales(t_gpu, u_outx_gpu).get()
           Nc1 = sumas_parciales(uno, u_outx_gpu).get()
           cuad1 = sumas_parciales(u_outx_gpu, tsqrt_gpu).get()
           sigma01 = abs(rms1/Nc1-pow((cuad1/Nc1),2))
           factor01 = sqrt(sigma01)/sigma0
           factor01_tot = np.append(factor01_tot, [factor01])
           Sp1 = sumas_parciales(u_inix_gpu, u_inix_gpu).get()
           Sp2 = sumas_parciales(u_outx_gpu, uno).get()
           Sp = Sp1 / Sp2
           sigma = sqrt(pow(sigma0,2) + gamma*betap[2]*Sp*pow(dz*aa,2)/2)
```

```
          factor = sigma/sigma0
          factor_tot = np.append(factor_tot, factor)
          phimax = gamma*P0*dz*aa
          valor = sqrt(2)*phimax*dz*aa/LD
          valor1 = pow((dz*aa/LD),2)*(1+(4*pow(phimax,2))/(3*sqrt(3)))
          factor1 = sqrt(1 + valor + valor1)
          factor1_tot = np.append(factor1_tot, [factor1]) #pag 114 agrawal
          sig = beta2*aa*dz
          sug = sig/seg
          factor2 = sqrt(pow(1+C*sug,2) + pow(sug,2) + orden3*pow(aa,2))
          factor2_tot = np.append(factor2_tot, [factor2])
          aa += 1000
       fl = 20000/LD
       tl = linspace(0,fl,21)
       plot(tl, factor01_tot, '--', tl, factor_tot, ':', tl, factor1_tot, ',', tl, factor2_tot,
'*', color='black')
       ylabel("$\sigma/\sigma_0$")
       xlabel("$Z/L_D$")
       legend((r"$\sigma_{VCM}$", "$sigma_{=shape}$","$\sigma_{phimax}$", "$\sigma_{conC}$"))
       grid(True)
       savefig('grafica_sigmas.png')
       clf()
       u_ini_x1 = np.concatenate ([t_zeros,u_ini_x])
       u_ini_x1 = np.concatenate ([u_ini_x1,t_zeros])
       u_out_x1 = np.concatenate ([t_zeros,u_out_x])
       u_out_x1 = np.concatenate ([u_out_x1,t_zeros])
       t1 = linspace(-1700,1700,int(3400/dt))
       plot(t1, pow(abs(u_out_x1),2), 'r', t1, pow(abs(u_ini_x1),2), 'b')
       ylabel("$|u(z,T)|^2$")
       xlabel("$T/T_0$")
       grid(True)
       savefig('figura_' + str(aa) + '.png')
       clf()
```

- Archivo agrawal_fig3_1_ssf.py para cuarto régimen de propagación:

```
import numpy as np
import numexpr as ne
from scipy import *
from pylab import *
from PyOFTK.utilities import *
import PyOFTK
T = 3200.0
nt = 614400
aa = 1
dz =2.0
maxiter = 10
betap = array([0.0,0.0,20.0])
alpha = 0.0
while aa <= 500:
   dt = T/nt
   t = linspace(-T/2.0, T/2.0, nt)
   u_ini_x = PyOFTK.gaussianPulse(t,166.5,0.0,0.667.0,1.0,0.0)
   u_out_x = PyOFTK.ssf(u_ini_x, dt, dz, nz, alpha, betap, 0.003, maxiter, 1e-5, phinLOut
= False, down1 = True)
   plot(t, pow(abs(u_out_x),2), 'r', t, pow(abs(u_ini_x),2), 'b')
   ylabel("|u(z,T)|^2")
   xlabel("T/T_0")
   xlim([-16,16])
   grid(True)
   savefig('figura_' + str(aa) + '.png')
   clf()
   aa += 1
```

- Archivo agrawal_fig3_1_ssf.py para cuarto régimen de propagación:

```
from numpy import *
import numpy as np
import numexpr as ne
from scipy import *
from pylab import *
from time import *
from PyOFTK.utilities import *
import PyOFTK
T = 170.0
aa= 10
dz = 0.002
nt = pow(2,10)
maxiter = 10
betap = array([0.0,0.0,20])
alpha = 0.0
t = linspace(-T/2.0, T/2.0, nt)
dt = T/nt
os.mkdir('Imagenes/prueba_tiempos')
while aa <= 1000:
    u_ini_x = PyOFTK.gaussianPulse(t,1.665,0.0,20,1,0.0)
    u_out_x = PyOFTK.ssf(u_ini_x, dt, dz, aa, alpha, betap, 3.0, maxiter, 1e-5, phiNLOut
= False, down1 = True)
    #u_out_x = PyOFTK.ssfgpu(u_ini_x, dt, dz, aa, alpha, betap, 0.003, maxiter, 1e-5,
context = False, phiNLOut = False, fullOutput = False)
    figure(figsize=(10,5))
    plot(t, pow(abs(u_out_x),2), 'r', t, pow(abs(u_ini_x),2), 'b')
    axis([-50,50,0,22])
    ylabel("|u(z,T)|^2")
    xlabel("T/T_0")
    grid(True)
    savefig('figura_' + str(aa) + '.png')
    clf()
    aa += 10
```

- Archivo para la simulación de la dispersión de tercer orden, TOD, en cuarto régimen de propagación (Archivo *agrawal_fig3_6_ssf.py*):

```
import PyOFTK
from pylab import *
T = 48.0
nt = pow(2,10)
dt = T/nt
t = linspace(-T/2, T/2, nt)
dz = 5.0
u_ini_x = exp(-pow(t,2)/2)
u_ini_y = zeros(nt)
betap = array([0,0,0,1])
alpha = array([0])
[u_out_x, outputParam] = PyOFTK.scalar(u_ini_x, nt, dt, dz, 1, alpha, betap, 1.0,
0, 0, 1, 1e-5, 0)
betap = array([0,0,1,1])
[u_out2_x, outputParam] = PyOFTK.scalar(u_ini_x, nt, dt, dz, 1, alpha, betap, 1.0,
0, 0, 1, 1e-5, 0)
plot(t, pow(abs(u_out_x),2), t, u_ini_x, ':', t, pow(abs(u_out2_x),2), '--',color='black')
ylabel("|u(z,T)|^2")
xlabel("T/T_0")
legend((r"beta_2 = 0", "z = 0","L_D = L'_D"))
xlim([-12,12])
show()
```

-Archivo soliton_ssf.py para la simulación de la propagación de un solitón de primer orden en una fibra óptica no lineal y dispersiva.

```
from numpy import *
import numpy as np
import numexpr as ne
from scipy import *
from pylab import *
from time import *
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
from matplotlib.colors import colorConverter
import os
import PyOFTK
T = 43.0
nt = pow(2,15)
maxiter = 10
frec=192
dz = 0.002
alpha = 0.0
gamma = 3.0
T0 = 0.0
FWHM = 1*2*arccosh(sqrt(2))
FWHM = np.float64(FWHM)
dt = T/nt
dt = np.float64(dt)
t = linspace(-T/2.0, T/2.0, nt)
u_out_x_total = []
betap=([0.0,0.0,-1])
P0 = pow(2*arccosh(sqrt(2)),2)/(gamma*pow(FWHM,2))
N=1
P0=np.float64(P0)
aa = 1/(P0*gamma*dz)
valor = []
z=[]
while aa <= 3*bb:
   u_ini_x = PyOFTK.sechPulse(t, FWHM, T0, P0, gamma, dz, aa, N)
   u_out_x = PyOFTK.ssf(u_ini_x, dt, dz, aa, alpha, betap, gamma, maxiter, 1e-5, phiNLOut
= False, down1 = True)
   valor.append(list(zip(t,pow(abs(u_out_x),2))))
   z = np.append(z,aa*dz)
```

```
    plot(t, pow(abs(u_out_x),2), 'r', t, pow(abs(u_ini_x),2), 'b')
    ylabel("|u(z,T)|^2")
    xlabel("T/T_0")
    grid(True)
    clf()
    aa += 1*bb
fig = plt.figure()
ax = fig.gca(projection='3d')
poly = PolyCollection(valor)
poly.set_alpha(0.7)
ax.add_collection3d(poly, zs=z, zdir='y')
ax.set_xlabel('T/T_0')
ax.set_xlim3d(-T/2, T/2)
ax.set_ylabel('z(Km)')
ax.set_ylim3d(0, z[len(z)-1])
ax.set_zlabel('|u(z,T)|^2')
ax.set_zlim3d(0, 0.4)
plt.show()
```

- Archivo vectorial2.py para la propagación de un pulso óptico en sus dos componentes de polarización a lo largo de una fibra birrefrigente:

```
from numpy import *
from pylab import *
from scipy import *
from scipy import interpolate
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
from matplotlib import cm
from PyOFTK.utilities import *
import PyOFTK
T = 992.0
nt = pow(2,15)
dt = T/nt
dt = np.float64(dt)
t = linspace(-T/2.0, T/2.0, nt)
dz = 0.002
T0x=1.0
T0y=1.0
FWHMx = T0x*2*(sqrt(log(2)))
FWHMx = np.float64(FWHMx)
FWHMy = T0y*2*(sqrt(log(2)))
FWHMy = np.float64(FWHMy)
dt = T/nt
dt = np.float64(dt)
gamma = 3.0
POWx = 1.0
POWy = 1.0
betapa = array([0.0, 1.0, 1.5])
betapb = array([0.0, 0.0, 1.5])
alphaa = array([0.0, 0.0, 0.0])
alphab = array([0.0, 0.0, 0.0])
psi = pi/4
chi = pi/2
metodo = 'eliptico'
maxiter = 10.0
tol = 1e-5
mm = 0
nz = 1
```

```
       N = 1
       valor = []
       valor1 = []
       z = 0.0
       os.mkdir('Imagenes/vectorial/vectorial3/pulso_gaussiano')
       esc = 100
       while nz <= 120: mm = int(round(nz*dz))
          u_ini_x = PyOFTK.gaussianPulse(t, FWHMx, 0.0, POWx, 1, 0.0)
          u_ini_y = PyOFTK.gaussianPulse(t, FWHMy, 0.0, POWy, 1, 0.0)
          [u_out_x, u_out_y] = PyOFTK.ssfvec1(u_ini_x, u_ini_y, dt, dz, nz, alphaa, alphab,
betapa, betapb, gamma, psi, chi, metodo, maxiter, tol)
          t1 = linspace(-esc,esc,2*esc*nt/T+1)
          u_out_x = u_out_x[len(u_out_x)/2-esc*nt/T:len(u_out_x)/2+esc*nt/T]
          u_out_y = u_out_y[len(u_out_y)/2-esc*nt/T:len(u_out_y)/2+esc*nt/T]
          u_ini_x = u_ini_x[len(u_ini_x)/2-esc*nt/T:len(u_ini_x)/2+esc*nt/T]
          u_ini_y = u_ini_y[len(u_ini_y)/2-esc*nt/T:len(u_ini_y)/2+esc*nt/T]
          plot(t1, pow(abs(u_ini_x),2), t1, pow(abs(u_ini_y),2), t1, pow(abs(u_out_x),2),
t1, pow(abs(u_out_y),2))
          valor.append(list(zip(t1, pow(abs(u_out_x),2))))
          valor1.append(list(zip(t1, pow(abs(u_out_y),2))))
          z = np.append(z,nz*dz)
          grid(True)
          savefig( 'Imagenes/vectorial/vectorial3/pulso_gaussiano/kmss_' + str(mm) + '.png')
          nz += 4
       show()
       fig = plt.figure()
       ax = fig.gca(projection='3d')
       poly = PolyCollection(valor)
       poly.set_alpha(0.7)
       ax.add_collection3d(poly, zs=z, zdir='y')
       ax.set_xlabel('$T/T_0$')
       ax.set_xlim3d(-esc, esc)
       ax.set_ylabel('z(Km)')
       ax.set_ylim3d(0, z[len(z)-1])
       ax.set_zlabel('$|u(z,T)|^2$')
       ax.set_zlim3d(0, 1.0)
       plt.show()
       fig1 = plt.figure()
       ax = fig1.gca(projection='3d')
       poly1 = PolyCollection(valor1)
```

```
poly1.set_alpha(0.7)
ax.add_collection3d(poly1, zs=z, zdir='y')
ax.set_xlabel('$T/T_0$')
ax.set_xlim3d(-esc, esc)
ax.set_ylabel('z(Km)')
ax.set_ylim3d(0, z[len(z)-1])
ax.set_zlabel('$|u(z,T)|^2$')
ax.set_zlim3d(0, 1.0)
plt.show()
```

Bibliografía

[1] Govind P. Agrawal, *"Nonlinear Fiber Optics", Third Edition*, Optics and Photonics, Academic Press, San Diego, California, 2001.

[2] Francesc Guim e Ivan Rodero, *.Arquitecturas basadas en computación gráfica (GPU)"*, PID_00184818, UOC, Universitat Oberta de Catalunya.

[3] Sonia Martínez López, *Tesis doctoral: "Generación de supercontinuo en fibras ópticas monomodo con fuentes de bombeo continuo"*, Universidad Complutense de Madrid, Facultad de Ciencias Físicas, Madrid, 2006.

[4] "NVIDIA Fermi Architecture",

http://www.orangeowlsolutions.com/archives/388

[5] Peter N. Glaskowsky, *"NVIDIA's Fermi: The First Complete GPU Computing Architecture", September 2009*

[6] "http://ht4u.net/reviews/2011/ nvidia_geforce_gtx_560_ti_msi_n560ti_twin_frozr_2/index2.php", *"NVIDIA GeForce GTX 560 TI im Test"*

[7] B. E. A. Saleh, M. C. Teich, *"Fundamentals of Photonics", Second Edition*, Wiley Series in Pure and Applied Optics, 2007.

[8] Rajiv Ramaswami, Kumar N. Sivarajan, Galan H. Sasaki, *.Optical Networks. A Practical Perspective", Third Edition* Morgan Kaufmann Publishers, Elsevier, EEUU, 2010.

[9] Prof. Thomas E. Murphy, Kathryn Tracey, *http://www.photonics.umd.edu/software/ssprop/* Universidad de Maryland.

[10] Prof. Rafael Gómez Alcalá, *Apuntes Dispositivos de Radiofrecuencia y Comunicaciones Ópticas* Escuela Politécnica, Cáceres. Universidad de Extremadura.

[11] Prof. Pedro Núñez Trujillo, *Apuntes Implementación de Sistemas de Comunicación por Línea y Vía Satélite* Escuela Politécnica, Cáceres. Universidad de Extremadura.

[12] Fernández de Jáuregui Ruiz, Iván, *Tesis:.Estudio de Sistemas Ópticos WDM para su Implementación en Redes de Alta Velocidad"* Facultad de Ingeniería, México DF. Universidad Nacional Autónoma de México,2010.

[13] A. Consoli, I. Esquivias y F. J. López-Hernández "Generación de Pulsos Ópticos a $1{,}5\mu m$ Mediante Conmutación de Ganancia en Láseres de Cavidad Vertical", Dpto. de Tecnología Fotónica, E.T.S.I. Telecomunicación. Universidad Politécnica de Madrid, Ciudad Universitaria, 28040, Madrid.

[14] "Transmission Systems and Media, Digital Systems and Networks", ITU-T-REC-G. Sup42 (04/2014).

[15] mlaprise/PyOFTK https://github.com/mlaprise/PyOFTK 27-febrero-2011.

[16] FFTW User's Manual "http://www.cs.berkeley.edu/ fateman/papers/fftw2.pdf" For version 2.1.5, 16 March 2003.

[17] "9 Upgrading from FFTW version 2" "http://www.fftw.org/doc/Upgrading-from-FFTW-version-2.html"

[18] "Python: reikna 6.0.7" "https://pypi.python.org/pypi/reikna"

[19] .Andreas Klöckner's web page" "http://mathema.tician.de/software/pycuda/"