



ugr | Universidad
de **Granada**

TRABAJO FIN DE GRADO
INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Laboratorio de Señales en Python

Autor

Luis Serra García

Directores

Sonia Mota Fernández
Pablo Padilla de la Torre



*ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN*

Granada, noviembre de 2017

Índice general

1	<i>Introducción</i>	6
1.1	Abstract	7
2	<i>Propuesta</i>	9
2.1	Python	9
2.2	Matlab	12
2.3	¿Por qué Python?	12
2.4	Python 2.x vs 3.x	13
2.5	Opciones	16
3	<i>Instalación</i>	18
3.1	Python	18
3.2	LabUGR	21
3.3	Dependencias externas	22
4	<i>Uso</i>	24
5	<i>Control de versiones</i>	25
6	<i>Estructura</i>	26
7	<i>Submódulos</i>	30
7.1	Funciones importadas de NumPy	30
7.1.1	Funciones para la creación y manipulación de vectores y matrices	30
7.1.2	Funciones matemáticas	31
7.2	Funciones importadas de Matplotlib	32
7.3	Integración	33
7.4	Transformada de Fourier	34
7.4.1	Transformadas reales	38
7.4.2	Transformada de Fourier de tiempo reducido	39
7.4.3	Transformada de Hilbert	42

7.5 Señales	43
7.5.1 Ventanas	43
7.5.2 Formas de Onda	45
7.5.3 Análisis espectral	47
7.5.4 Convolución - correlación	49
7.6 Sistemas	51
7.6.1 Conversión de sistemas LTI	52
7.6.2 Trazado en el plano z	53
7.6.3 Sistemas LTI continuos	53
7.6.4 Sistemas LTI discretos	57
7.7 Filtros	58
7.7.1 Diseño de filtros	58
7.7.1.1 Filtros de Respuesta Finita al Impulso (FIR)	58
7.7.1.2 Filtros de Respuesta infinita al Impulso (IIR)	60
7.7.2 Simulación de filtros	63
7.7.3 Análisis espectral	64
7.7.4 Representación de filtros	64
7.8 Audio	65
7.9 Doc	67
7.10 Testing	68
7.10.1 Unit testing	68
8 Compilación	70
8.1 Numpy y Cython	70
8.2 Compiladores C++ y Fortran	70
8.3 LAPACK y BLAS	72
8.4 Modo desarrollador	75
9 Distribución	76
9.1 Wheel	77
9.1.1 Etiquetas para las Wheels	78
9.2 Mac	80
9.2.1 Pyenv	80
9.2.2 Delocate	80
9.2.3 Script	81
9.3 Windows	82
9.4 Linux	82
9.4.1 Docker	83
9.4.1.1 Auditwheel	84

10 Aplicación práctica: laboratorio de señales analógicas y digitales	86
10.1 Práctica 0: Introducción a Python	86
10.2 Sistemas Lineales	102
10.2.1 Práctica 1: Representación de señales	102
10.2.2 Práctica 2: Números complejos	110
10.2.3 Práctica 3: Desarrollo en serie de Fourier	114
10.2.4 Práctica 4: Sistemas Lineales	121
10.3 Señales digitales	126
10.3.1 Práctica 1: Señales básicas	126
10.3.2 Práctica 2: Transformada discreta de Fourier	129
10.3.3 Práctica 3: Diseño de filtros por el método de las ventanas	132
11 Conclusiones y líneas futuras	136
11.1 Conclusiones	136
11.2 Líneas futuras	136
12 Bibliografía	138
Apéndice A: Planificación y costes del proyecto	140
A.1 Planificación	140
A.2 Costes	142

1 Introducción

En la actualidad, el software más utilizado para el procesado y representación de señal es Matlab, un programa muy refinado, con un gran repertorio de funcionalidades y una gran comunidad científica que avala su uso. El principal problema de Matlab es su coste, con licencias que llegan a superar las decenas de miles de euros (incluyendo herramientas adicionales) lo que lo convierte en una herramienta inaccesible para muchas personas.

Hoy en día, hay varias alternativas gratuitas y de código abierto que, aunque pueden no estar tan refinadas y tener tantas funcionalidades como Matlab, proporcionan las herramientas esenciales requeridas para lanzarse al campo del procesamiento, análisis y representación de señales. De todas estas opciones disponibles, la que se alza por encima de las demás es Python. Python es un lenguaje de programación de alto nivel especialmente diseñado para ser similar al lenguaje humano y por lo tanto es amigable y fácil de aprender. Se puede aplicar no solo a la computación matemática, sino a muchas otras disciplinas profesionales, como la ciencia de datos, el aprendizaje automático o el diseño de páginas web. Además, Python cuenta con una extensa comunidad de colaboradores que contribuyen a la mejora de Python, así como la creación de módulos que aportan nuevas funcionalidades. Los módulos principales que se pueden utilizar para el procesado de señales son los siguientes; NumPy, para cálculos matemáticos y matriciales; Matplotlib, para visualización y representación de señales; SciPy, para el análisis de señales.

Aunque Python tiene los módulos necesarios para el análisis de señales, la instalación de estas bibliotecas individualmente puede ser realmente complicada, especialmente en el caso de SciPy en Windows. Algunos paquetes, como Anaconda o WinPython, intentan abordar esto proporcionando compilaciones de varios módulos en una sola instalación. Sin embargo, esto hace que la instalación de Python sea muy pesada y consume muchos recursos, y no resuelve el hecho de que el usuario final todavía tiene que saber dónde se encuentra cada función dentro de los diferentes módulos para poder usarlas.

El objetivo de proyecto es la construcción de una librería libre en el lenguaje de programación Python para el análisis, procesado y representación de señales. Es un trabajo de recopilación y empaquetado de todas las funciones necesarias para facilitar una introducción al mundo de Python y el procesado de señales a través de las asignaturas de Señales Digitales y Sistemas Lineales de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada.

Este documento empieza con un análisis de Python, sus ventajas/inconvenientes con respecto a Matlab y las razones que hacen de él un lenguaje de programación ideal para un uso introductorio en el mundo del procesado de señales. A continuación, se describen los pasos a seguir para la instalación de Python, así como de la librería construida en este proyecto, de aquí en adelante referida como labUGR. Seguido de un apartado donde se describe la estructura de esta librería, así como una pequeña documentación y ejemplos de todas las funciones incluidas en los diferentes submódulos. A continuación, se describen los pasos seguidos para la compilación y distribución de labUGR, así como las herramientas utilizadas. Al final del documento se incluyen una serie de prácticas propuestas para las dos asignaturas de la UGR.

Palabras clave: procesado de señales, Matlab, Python, NumPy, Matplotlib, SciPy, compilación y distribución de paquetes Python, Docker, Linux, Mac OS, Windows.

1.1 Abstract

In the field of signal processing, there is a piece of software that sticks out from the rest; Matlab. Matlab is a professionally developed proprietary mathematical environment established by MathWorks, and has been the main tool used for signal analysis and essentially most numerical computations in University and Company settings. Though, with such a complex and specialized software, it has one main drawback; its price. Ranging from hundreds to the tens of thousands of Euros for commercial licenses inclusive of additional toolboxes, these prohibitive prices transform this great software into an inaccessible resource for many individuals.

Nowadays, there are several free and open source alternatives that, although they might not be as finely tuned and with as many functionalities as Matlab, provide the essential tools required to launch into the field of the processing, analysis and representation of signals. Of all of these available options, the best suited and highly recommended language is Python. Python is a high-level programming language specially designed to be similar to the human language and thus making it friendly and easy to learn. It can be applied not only to math computation but to many diverse professional disciplines such as data science, machine learning and web-design. Furthermore, Python has an extensive community of contributors which constantly create, update and upgrade new modules which provide new and improved functionalities. The main modules which can be used for signal processing are the following; NumPy for mathematical computations and matrix calculations; Matplotlib for signal visualization and representation; SciPy for signal analysis.

Although Python has the necessary modules for signal analysis, installing these libraries individually can become really complicated, especially in the case of SciPy in Windows. Some packages, like Anaconda or WinPython, try to tackle this by providing compilations of several modules in a single installation. However, this makes the Python installation really heavy and resource consuming and doesn't solve the fact that the end user still has to know where each function is located within the libraries in order to use them.

The aim of this project is to build an open source Python library using Matplotlib and NumPy as a base and compiling functions from various other modules to create a compact environment which includes all the necessary functions for signal processing and visualization.

Firstly, this document describes both Python and Matlab, stating the reasons why the former is a good alternative to Matlab. The following chapters describe the steps required to install both Python and the aforementioned library itself, named labUGR; followed by a characterization of the library's structure as well as a small description of every function included in the various submodules, including some practical usage examples and links to the full official English documentations.

After the structure, there are various chapters describing the steps followed for both the compilation and the distribution of the library along with all the tools employed. Finally, at the end of this project, a series of labs are proposed for the “Señales digitales” and “Sistemas Lineales” courses from the University of Granada, serving as an introduction to the usage of Python and this library.

Keywords: signal processing, Matlab, Python, NumPy, Matplotlib, SciPy, compilation and distribution of Python modules, Docker, Linux, Mac OS, Windows.

2 Propuesta

En la actualidad, existen diferentes opciones a la hora de escoger software para el procesado de señales. Entre ellas se encuentran Matlab y Python, que destacan por encima de otras alternativas como R y Octave ya que son las más usadas por las principales instituciones académicas y científicas.

2.1 Python

Python es un lenguaje de programación de alto nivel de propósito general diseñado para ser utilizado en una gran variedad de tipos de aplicaciones, desde el desarrollo de páginas webs hasta aplicaciones en el mundo de la ciencia de datos.



El desarrollo de Python se inició en 1989 por Guido van Rossum, programador holandés de la universidad de Ámsterdam, como un sucesor al lenguaje de programación ABC. Ha tenido tres grandes versiones a lo largo de su historia, Python 1.0 en el año 1994, Python 2.0 en 2000 y Python 3.0 en 2008. En la actualidad solo las versiones 2.7, 3.3, 3.4, 3.5 y 3.6 son estables y están siendo mantenidas con actualizaciones periódicas. A partir del año 2000, Python pasó a ser un proyecto de código abierto, bajo la fundación PSF (Python Software Foundation) con un repositorio público en GitHub (<https://github.com/python/cpython>) y más de 400 contribuidores.

Las principales características de Python son las siguientes:

- Simple: Python es un lenguaje simple y minimalista que pretende aproximar el lenguaje de programación a lenguaje humano, facilitando el desarrollo de código que es fácil de comprender a simple vista.
- De código abierto: el uso de Python es completamente gratuito y permite la distribución de software basado en él, tanto de pago como libre. El código fuente forma parte del dominio público y tiene una gran comunidad de desarrolladores que contribuyen a la mejora este de manera colaborativa.
- De alto nivel: Python es un lenguaje de alto nivel, es decir, en lugar de tratar con direcciones de memoria y registros, este utiliza conceptos como variables, objetos, funciones, bucles, ... Esto facilita enormemente su uso a costa de una disminución en su rendimiento.
- Interpretado: a diferencias de lenguajes de programación compilados como C o Java, que necesitan de una compilación del código fuente antes de ser ejecutados, el código en Python se ejecuta directamente. Python forma parte de los lenguajes de programación conocidos como interpretados, estos utilizan un intérprete para interpretar el código en tiempo real.

- Portable: gracias a ser un lenguaje interpretado, cualquier código programado en Python se puede ejecutar sin realizar ningún cambio en cualquier sistema con un intérprete de Python con un mismo resultado.
 - Orientado a objetos: Python es un lenguaje que utiliza el paradigma de programación orientada a objetos (POO).
 - Modular: la librería de Python contiene un gran número de funcionalidades incluidas en diferentes módulos internos; por ejemplo, funcionalidades para la encriptación (hashlib, md5, sha), funcionalidades para utilizar protocolos de internet (httplib para HTTP, ftplib para FTP, smtplib, para un cliente SMTP), ... Esto permite que Python sea utilizado en muchos ámbitos diferentes.
- Gracias a ser un lenguaje de código abierto y tener una gran comunidad, existen un gran número de módulos de terceros que implementan nuevas funcionalidades para Python y la mayoría son de acceso libre.
- Capaz de interactuar con otros lenguajes: Python contiene funcionalidades para interactuar con código escrito en otros lenguajes.

El uso de Python como lenguaje de programación ha crecido enormemente en los últimos años, especialmente en los campos de la ciencia de datos, machine learning (aprendizaje automático e inteligencia artificial) y la investigación académica. En 2017, Python se ha convertido en el lenguaje con una mayor tasa de crecimiento (fuente: [stackoverflow](#)):

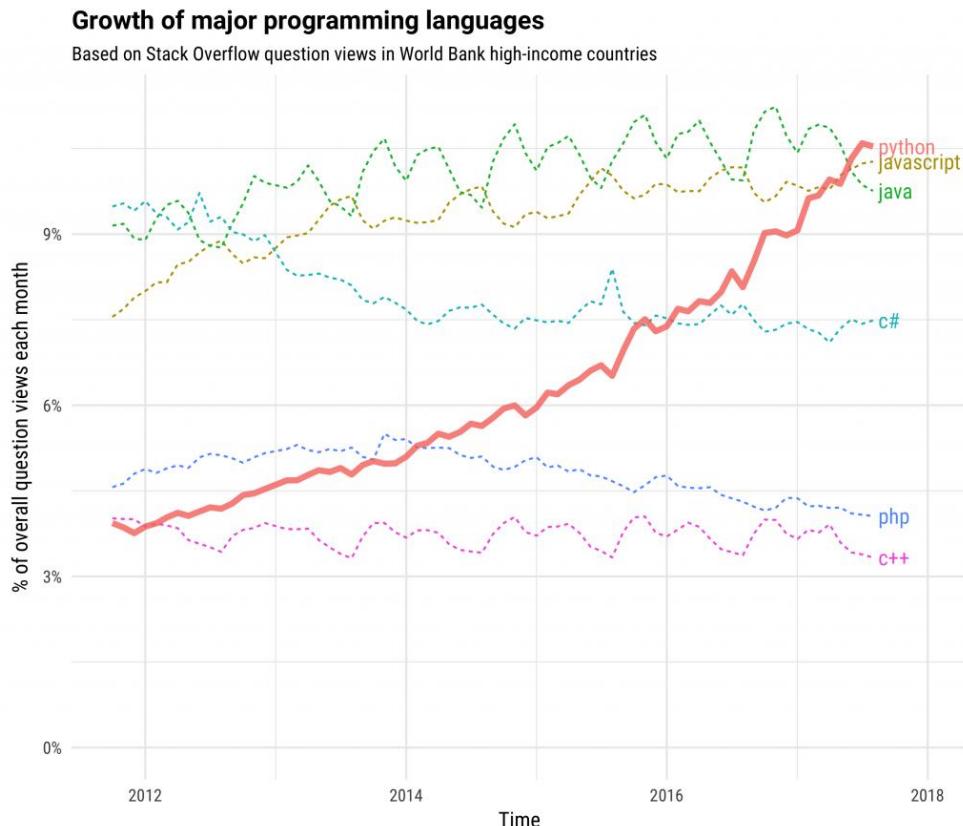


Figura 1: Crecimiento del uso de diversos lenguajes de programación según el número de preguntas en Stack Overflow (fuente)

También ha sido escogido como el lenguaje de programación del año 2017 por la plataforma [IEEE Spectrum](#) y es el segundo lenguaje más popular en GitHub (fuente: [Business Insider](#)) según el número de ‘pull requests’ (solicitud para modificar código), por detrás de JavaScript:

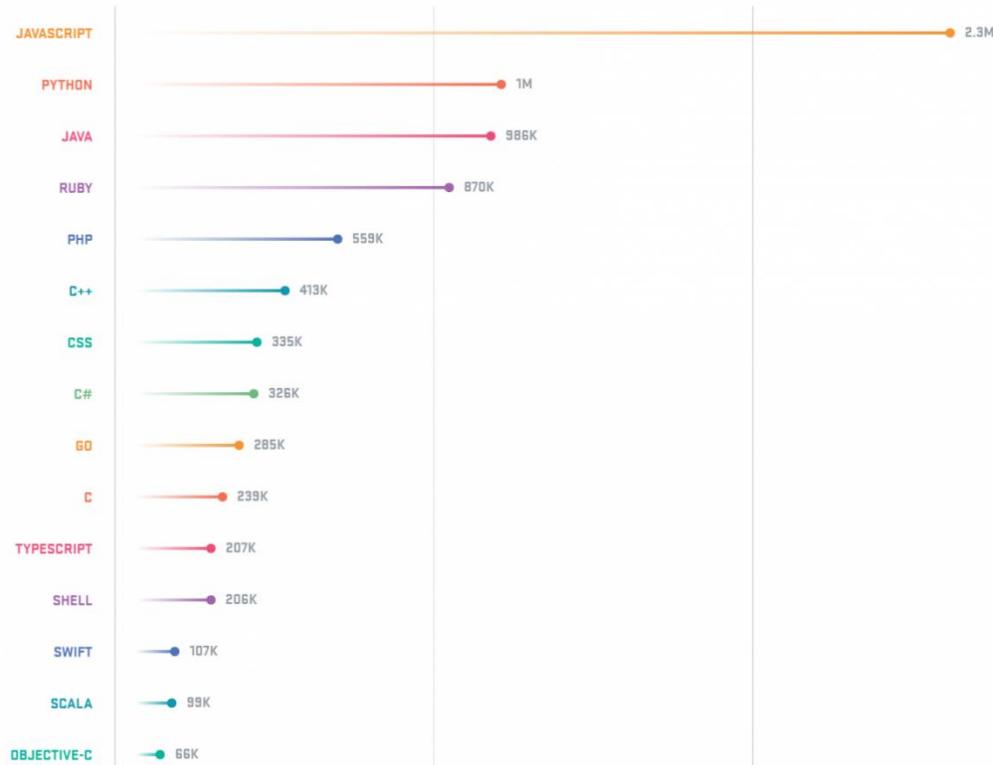
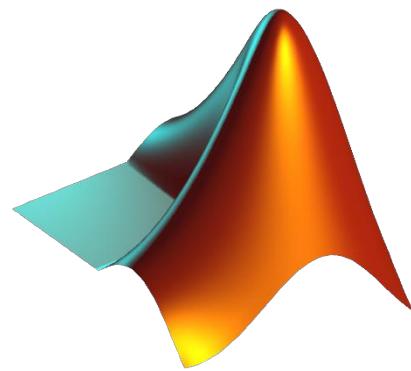


Figura 2: Número de interacciones en GitHub (‘pull requests’) de los principales lenguajes de programación ([fuente](#))

2.2 Matlab

Matlab o laboratorio de matrices (**MAT**rix **L**ABoratory) es una herramienta para el cálculo matemático y visualización gráfica. Tiene 2 elementos principales, un lenguaje de programación propio y un entorno de desarrollo integrado (IDE) que proporciona un entorno interactivo. A diferencia de otros lenguajes de programación, Matlab está dirigido únicamente al cálculo matemático y a la representación gráfica, lo que permite utilizar una notación matemática a la hora de programar. Esto facilita mucho la resolución de problemas matemáticos en comparación con lenguajes de programación tradicionales.



Matlab es muy popular en el ámbito académico y de investigación gracias a su facilidad de uso y suele ser la herramienta estándar para los cursos introductorios y avanzados de matemáticas e ingenierías. Su edición básica tiene un gran número de funcionalidades, principalmente: la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos y la creación de interfaces de usuario (GUI).

Como muchos otros lenguajes de programación, MATLAB tiene una amplia variedad de extensiones que se pueden añadir a la edición básica para su uso en diversos campos. Estas bibliotecas, llamadas ‘toolboxes’, añaden nuevas funcionalidades y herramientas directamente en la interfaz principal de Matlab (IDE). Existen un total de 47 ‘toolboxes’ disponibles para Matlab, distribuidas en 8 categorías: computación paralela; matemáticas, estadísticas y optimización; sistemas de control; procesamiento de señales y comunicaciones; procesamiento de imágenes y visión artificial; pruebas y mediciones; computación financiera; computación biológica.

Matlab también contiene una herramienta separada de la interfaz principal, llamada Simulink. Este es un entorno de programación visual que incluye herramientas para el diseño y simulación de modelos o sistemas. Utiliza un nivel de abstracción por encima del módulo principal de Matlab, utilizando piezas y sistemas como cajas negras y centrándose en el análisis de sucesos.

2.3 ¿Por qué Python?

Matlab tiene un gran número de ventajas que han hecho que se convierta en una herramienta clave en cursos técnicos de la mayoría de universidades del mundo:

- Tiene una sólida implementación de un gran número de funciones y librerías, todas testeadas y optimizadas por desarrolladores profesionales.
- Tiene una gran comunidad científica que lo utiliza a diario.
- Tiene un lenguaje de programación simple y especialmente diseñado para el cálculo matemático lo que hace que sea fácil de aprender.
- Incluye todas las herramientas en una interfaz única.
- No hay muchas buenas alternativas a Simulink en la actualidad.

A pesar de todas estas ventajas, Matlab tiene un gran inconveniente, el precio. Matlab es un software propietario de la empresa MathWorks, con un precio muy elevado tanto para un uso personal como institucional o comercial. Cada uno de los módulos complementarios, o toolboxes, se debe comprar por separado, elevando el precio final de producto. Algunas de las toolboxes básicas para el análisis de señales son las siguientes: ‘Signal Processing Toolbox’, para el procesado de señales; DSP Toolbox, para el procesado de señales digitales; Control System Toolbox, para el análisis de sistemas lineales; Image Processing Toolbox, para el procesado de imágenes.

Con esta librería de Python queremos resolver los inconvenientes del uso de Matlab, poniendo a disposición de los estudiantes un software gratuito, de fácil acceso e instalación con el que puedan realizar prácticas en análisis de señales.

Siendo Python uno de los lenguajes de programación con una mayor tasa de crecimiento en los últimos años; con aplicaciones en ámbitos tan diversos como la ciencia de datos, el desarrollo de páginas webs, inteligencia artificial, ...; con una sintaxis simple y legible; lo que lo convierte en un lenguaje ideal para la iniciación en el mundo de la programación.

2.4 Python 2.x vs 3.x

Una vez establecidas las razones por las cuales es interesante entrar en el mundo del análisis de señales en Python, el primer paso es seleccionar la versión de Python a utilizar. Actualmente, hay dos versiones principales que tienen soporte oficial, con actualizaciones regulares para solucionar errores y mejorar el rendimiento. Estas son Python 2.7.x y Python 3.x, las cuales son incompatibles entre sí y las principales diferencias entre ellas son las siguientes:

- División de números enteros: uno de los mayores cambios entre la versión 2.7.x y las 3.x y causa principal de los problemas de compatibilidad entre ellas, es la manera de manipular divisiones de números enteros. En el caso de Python 2.7.x, el resultado siempre será un número entero. Por ejemplo, dividendo 5 entre 4 daría como resultado **1 ($5/4 = 1$)** a no ser que uno de los números sea definido como flotante, añadiendo ‘.0’ **($5.0/4 = 1.25$)**. En cambio, en la versión 3.x el operador ‘/’ siempre realizará una división exacta de los números y para realizar una división entera hay que utilizar el operador ‘//’.
- La función print(): este es un cambio más trivial en la sintaxis de la función print (para mostrar valores en pantalla) pero que también introduce incompatibilidad entre las 2 versiones. En Python 2.7.x se puede utilizar tanto **print 'Hello World!'** como **print ('Hello World!')** y puede ser tratada como sentencia o como función. En Python 3.x solo se puede utilizar print como una función (**print('Hello World!')**), y si se usa de la otra manera provoca un error de sintaxis.

- Caracteres UNICODE: en la versión 2.x las cadenas de caracteres están codificadas en ASCII que define un total de 127 caracteres diferentes. Con 3.x las cadenas pasaron a ser Unicode (utf-8) que incluye de por si caracteres ASCII, lo que las hacen compatibles entre sí. Con Unicode se pueden representar un total de 136.690 caracteres utilizando de 1 a 4 bytes (en el caso de los caracteres ASCII, utiliza 1 Byte, empezando con un 0). Esto nos permite utilizar caracteres como por ejemplo $\mu(\text{\u03bc})$ o $\pi(\text{\u03c0})$ en nuestras cadenas de caracteres utilizando '\u' más el código Unicode del carácter.
- Entrada de usuario (input()): en Python, para obtener valores introducidos por el usuario se utiliza la función input(). En Python 3.x, cualquier variable creada de esta forma será almacenada como una cadena de caracteres (**1234 -> '1234'**). En el caso de 2.7.x, Python primero intenta convertir el valor introducido a un número, por lo que si el usuario introduce un número este será guardado como un número y no una cadena de caracteres. Por ello, utilizando Python 3.x, es necesario convertir el valor de entrada a un número utilizando la funciones ***int()*** para números enteros o ***float()*** para números reales.

Estas diferencias sintácticas no son muy grandes, y eligiendo cualquiera de las dos versiones, no sería muy difícil adaptarse a la otra. Aparte de estas diferencias básicas, hay un gran número de nuevas funcionalidades y mejoras de rendimientos solo disponibles en las versiones 3.x. Citando la página oficial de Python, “*Python 2.x is legacy, Python 3.x is the present and future of the language*”

El único factor que te puede llevar a utilizar Python 2.x en vez de Python 3.x es el hecho de que no todas las librerías están aún adaptadas a las nuevas versiones y puedes encontrar problemas de incompatibilidad y el hecho de que por defecto, las versiones de Python que vienen incluidas en Mac OS y Linux son 2.7. Esto me llevó a pensar en escoger Python 2.7.x (más tarde cambie a 3.x) para realizar mi proyecto, con miedo a más adelante encontrarme con librerías incompatibles. Incluso, uno de los libros más recomendados para aprender a utilizar Python y que yo mismo utilicé, “***Learn Python The Hard Way***” (<https://learnpythonthehardway.org/book/>) contiene el siguiente párrafo en su introducción:

“A programmer may try to get you to install Python 3 and learn that. Say, “When all of the Python code on your computer is Python 3, then I’ll try to learn it.” That should keep them busy for about 10 years. I repeat, do not use Python 3. Python 3 is not used very much, and if you learn Python 2 you can easily learn Python 3 when you need it. If you learn Python 3 then you’ll still have to learn Python 2 to get anything done. Just learn Python 2 and ignore people saying Python 3 is the future.”



Básicamente, el autor recomienda no utilizar Python 3 ya que esta versión no es el estándar. La última edición de este libro se publicó a finales de 2013 y no ha sido hasta el verano de 2017 cuando se ha publicado la versión este libro para Python 3. El mundo de Python ha sufrido grandes cambios en los últimos años y, como hemos visto anteriormente, se ha convertido en uno de los lenguajes que crece más rápidamente, por encima de JavaScript y Java.

Esto ha provocado que la adaptación completa a las versiones 3.x en el mundo de Python haya sido mucho más rápida de lo esperado. La mayoría de librerías en la actualidad son compatibles entre las versiones 2.7.x y 3.x, pero esto provoca que estas no puedan utilizar las mejoras que Python 3.x introduce en su totalidad. En 2016, un grupo de proyectos de Python declaró que iban a dejar de dar soporte a Python 2.7.x (<http://www.python3statement.org/>) a partir de 2020, entre ellas Matplotlib, unas de las librerías esenciales para este proyecto. No solo eso, sino que la versión entera de Python 2.x va a dejar de ser mantenida a partir de 2020.



A medida que progresé en la investigación y recopilación de las librerías que iba a utilizar en el proyecto, he podido comprobar que la mayoría de librerías están ya adaptadas a Python 3.x y son solo algunos módulos pequeños y no muy populares los que no están adaptados. Esto, junto con el hecho de que en mi opinión los cambios sintácticos en 3.x detallados anteriormente lo hacen más comprensible, me ha llevado a finalmente **escoger Python 3.x como la versión a utilizar en mi proyecto**.

A fecha de octubre de 2017, la versión estable más reciente de Python es la 3.6.3, estando la siguiente versión 3.7.0 en fase Alpha 2 con pronósticos de ser lanzada oficialmente a mediados de 2018. Dentro de la versión 3.x, este proyecto dará soporte a las versiones 3.4.x, 3.5.x y 3.6.x debido a que algunas de las librerías utilizadas en el proyecto, como NumPy y Matplotlib, solo soportan Python 2.7.x o mayor que 3.4.x.

El hecho de no solo incluir una única versión es debido a que normalmente las distribuciones de Linux incluyen una versión de Python 3.x, normalmente 3.4.x o 3.5.x y puede ser bastante complicado cambiar la versión de Python ya que algunas herramientas del sistema y otras aplicaciones en Linux utilizan Python3 y no es posible simplemente desinstalar e instalar una nueva versión.

2.5 Opciones

A la hora de utilizar Python para el procesado y representación de señales nos encontramos con una serie de problemas: existen un gran número de librerías externas en Python y es necesario determinar cuáles de ellas son necesarias; la instalación de algunas de estas librerías puede llegar a ser complicado, especialmente en Windows; es necesario también tener un extenso conocimiento de las librerías en sí, para saber dónde exactamente está cada función dentro de la propia librería.

Existen varias alternativas a la hora de utilizar Python para el procesador y representación de señales:

- Opción I: la primera opción sería instalar una por una todas las librerías necesarias para el procesado de señales. Algunas de estas librerías como SciPy provocan muchos problemas durante la instalación.

```
>>>
PS C:\Users\LuikS> pip3 install scipy
Collecting scipy
  Downloading scipy-0.19.1.tar.gz (14.1MB)
    100% |██████████| 14.1MB 82kB/s
Building wheels for collected packages: scipy
  Running setup.py bdist_wheel for scipy ... error
    Complete output from command c:\users\luiks\appdata\local\programs\python\python36-32\python.exe -u -c "import setuptools, tokenize; __file__='C:\\Users\\Luiks\\AppData\\Local\\Temp\\pip-build-q9gk62j3\\scipy\\setup.py';f=getattr(tokenize, 'open', open)(__file__);code=f.read();f.close();exec(compile(code, __file__, 'exec'))" bdist_wheel -d C:\\Users\\Luiks\\AppData\\Local\\Temp\\tmpxiz6x77epip-wheel- --python-tag cp36:
    lapack_opt_info:
      libraries mkl_rt not found in ['c:\\users\\luiks\\appdata\\local\\programs\\python\\python36-32\\lib', 'c:\\', 'c:\\users\\luiks\\appdata\\local\\programs\\python\\python36-32\\\\libs']
      NOT AVAILABLE

      openblas_lapack_info:
        libraries openblas not found in ['c:\\users\\luiks\\appdata\\local\\programs\\python\\python36-32\\lib', 'c:\\', 'c:\\users\\luiks\\appdata\\local\\programs\\python\\python36-32\\\\libs']
        NOT AVAILABLE

      atlas_3_10_threads_info:
        Setting PTATLAS=ATLAS
        c:\users\luiks\appdata\local\programs\python\python36-32\lib\site-packages\numpy\distutils\system_info.py:1051: UserWarning: Specified path C:\\projects\\numpy-wheels\\windows-wheel-builder\\atlas-builds\\atlas-3.10.1-sse2-32\\lib is invalid.
          pre_dirs = system_info.get_paths(self, section, key)
        <class 'numpy.distutils.system_info.atlas_3_10_threads_info'>
        NOT AVAILABLE

      atlas_3_10_info:
        <class 'numpy.distutils.system_info.atlas_3_10_info'>
        NOT AVAILABLE

      atlas_threads_info:
        Setting PTATLAS=ATLAS
        <class 'numpy.distutils.system_info.atlas_threads_info'>
        NOT AVAILABLE

      atlas_info:
        <class 'numpy.distutils.system_info.atlas_info'>
        NOT AVAILABLE
```

Figura 3: Captura del error obtenido al intentar instalar SciPy en Windows sin las herramientas externas necesarias

Nota: Esto es debido a que esta librería no está empaquetada en una Wheel y por tanto Python intenta instalarla a través del código fuente para lo que es necesario tener instalado una serie de herramientas. Mas información en el apartado '[Distribución](#)'.

- Opción II: la segunda opción sería la utilización de distribuciones de Python como Anaconda o WinPython, que incluyen un gran número de módulos adicionales en su instalación. Estas herramientas resuelven el problema de la instalación de las librerías ya que vienen incluidas las librerías necesarias para el procesado de señal y permiten al usuario final utilizar todas ellas directamente tras la instalación.



El principal inconveniente de este método, es que estas distribuciones contienen también muchos módulos que no son necesarios para el procesado de señales y que provocaran que la instalación de Python sea más pesada y requiera más recursos para ser utilizada. Por ejemplo, Anaconda contiene un total de 214 módulos externos en su instalación más simple, de los cuales la mayoría de ellos no son necesarios.

Con el módulo labUGR queremos poner a disposición del usuario final un software compacto, que permita crear un entorno de trabajo con todas las funciones necesarias para el procesado y representación de señales digitales y analógicas. Para ello, se utilizarán los módulos NumPy y Matplotlib como base (dos librerías de Python muy consolidadas), añadiendo funciones de otros módulos (principalmente SciPy, pero también de otros módulos como PyAudio AudioRead y mpmath), haciendo especialmente hincapié en la correcta distribución de la librería para que el usuario final no tenga problemas al instalarla.

Para un usuario no muy experimentado en Python, tener que trabajar con diferentes librerías y submódulos puede ser complicado y provocar confusión. Cada librería puede tener una serie de submódulos y es necesario conocer la localización de las diferentes funciones dentro de ellos para poder utilizarlas. Por ello, este proyecto está diseñado para que, con una simple línea de código, el usuario pueda crear un entorno de trabajo con todas las funciones necesarias ya importadas. Por ejemplo, esta sería la diferencia entre la creación y representación de una señal triangular:

Utilizando las librerías por separado:

```
from numpy import linspace, pi
from scipy.signal import sawtooth
from matplotlib.pyplot import plot, show

t = linspace(0, 5, 300)
senal = sawtooth(2*pi*1*t, width=0.5)
plot(t, senal);
show()
```

Utilizando labUGR:

```
from labugr import *

t = linspace(0, 5, 300)
senal = sawtooth(2*pi*1*t, width=0.5)
plot(t, senal);
show()
```

3 Instalación

Para la utilización de la librería labUGR es necesario seguir una serie de paso: instalar Python, instalar la librería en sí e instalar una serie de dependencias externas necesarias para utilizar las funcionalidades relacionadas con señales de audio (el resto de funciones pueden funcionar sin ellas).

3.1 Python

Para poder utilizar esta librería es necesario tener instalado Python 3.4.x, 3.5.x o 3.6.x junto con pip, una herramienta para la instalación de paquetes de Python (recomendada por PyPA). Esta suele venir incluida con la instalación de Python. Aunque la librería es compatible con estas 3 versiones, siempre que sea posible es recomendable instalar 3.6.x en vez de cualquiera de las otras dos versiones ya que en las nuevas versiones se incluyen nuevas funcionalidades y mejoras en el rendimiento (mejoras en [3.4](#), [3.5](#) y [3.6](#)). La instalación de Python varía según el sistema operativo:

- **Windows:** el instalador de Python se encuentra disponible en la página oficial (<https://www.python.org/downloads/release/python-363/>). Basta con descargarse el ejecutable correspondiente a la versión de su sistema operativo: '[Windows x86-64 executable installer](#)' para Windows 64 bits o '[Windows x86 executable installer](#)' para Windows 32 bits.

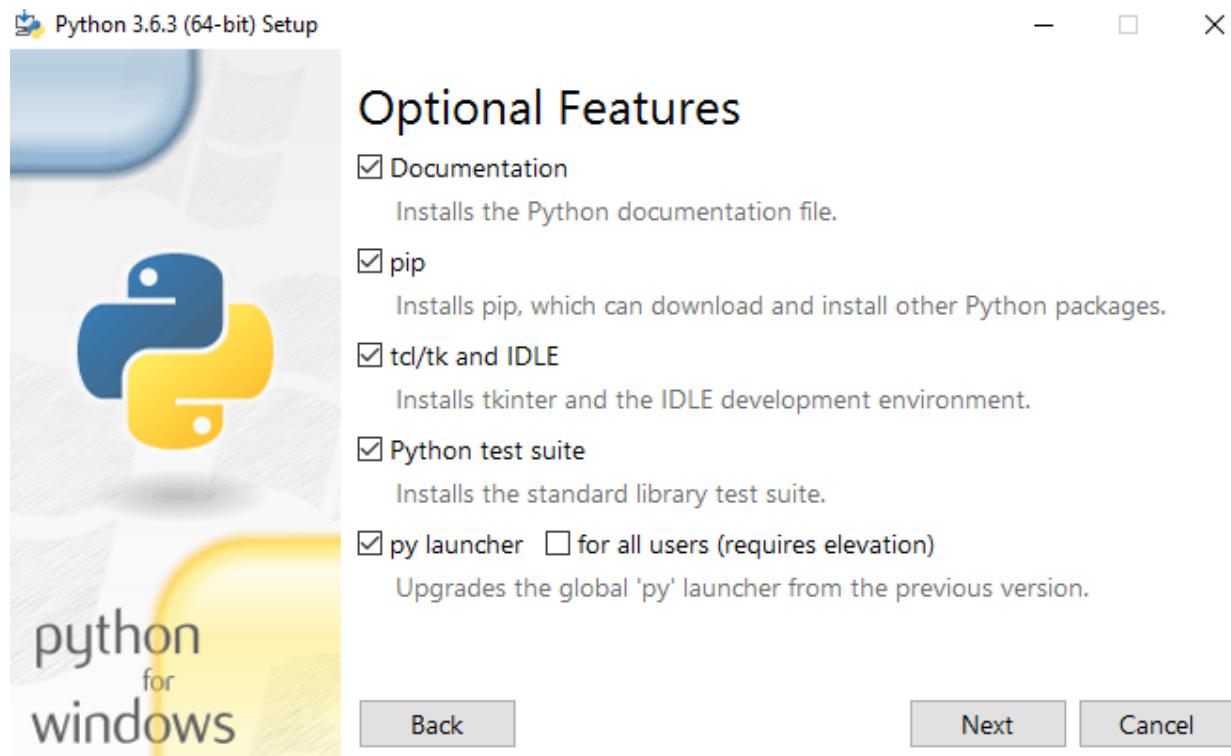


Figura 4: Primer cuadro de diálogo de la instalación de Python en Windows

Al instalar se tiene que seleccionar la opción avanzada y asegurase que la herramienta pip esa incluida para poder instalar librerías fácilmente. También es necesario incluir Python al PATH de Windows (seleccionado la opción de ‘Add Python to environment variables’) para poder utilizar Python y pip directamente desde la terminal.

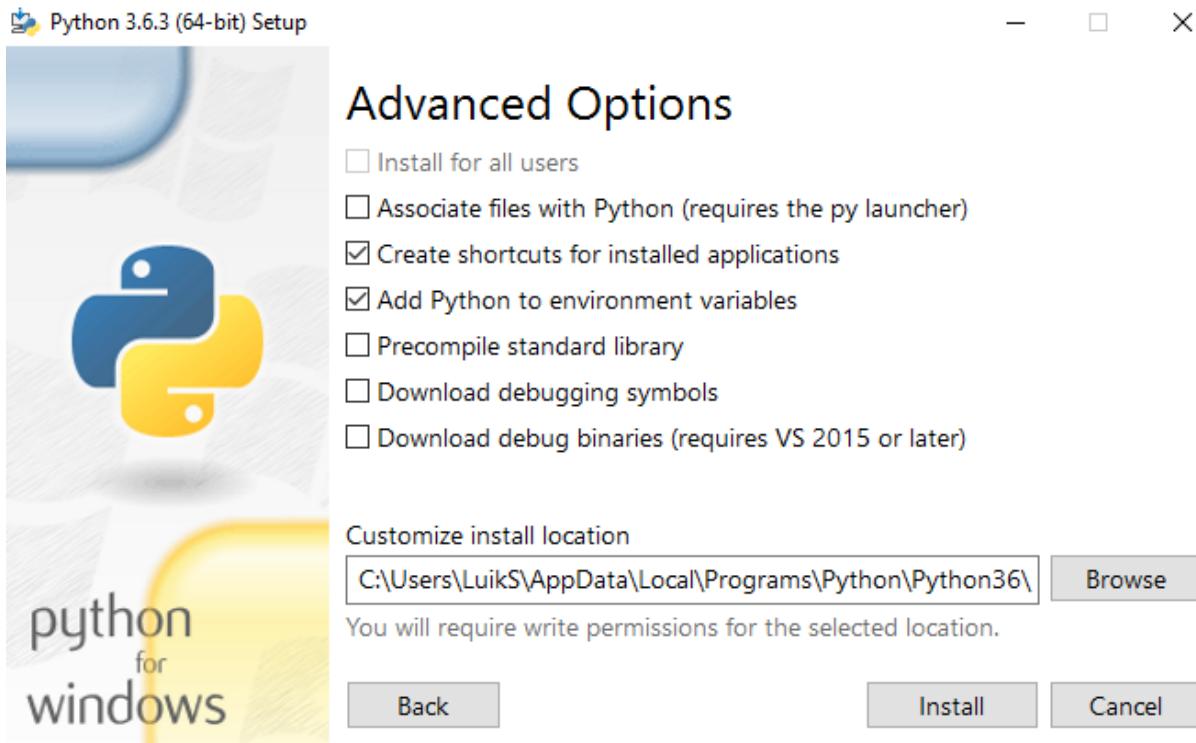


Figura 5: Segundo cuadro de dialogo de la instalación de Python en Windows

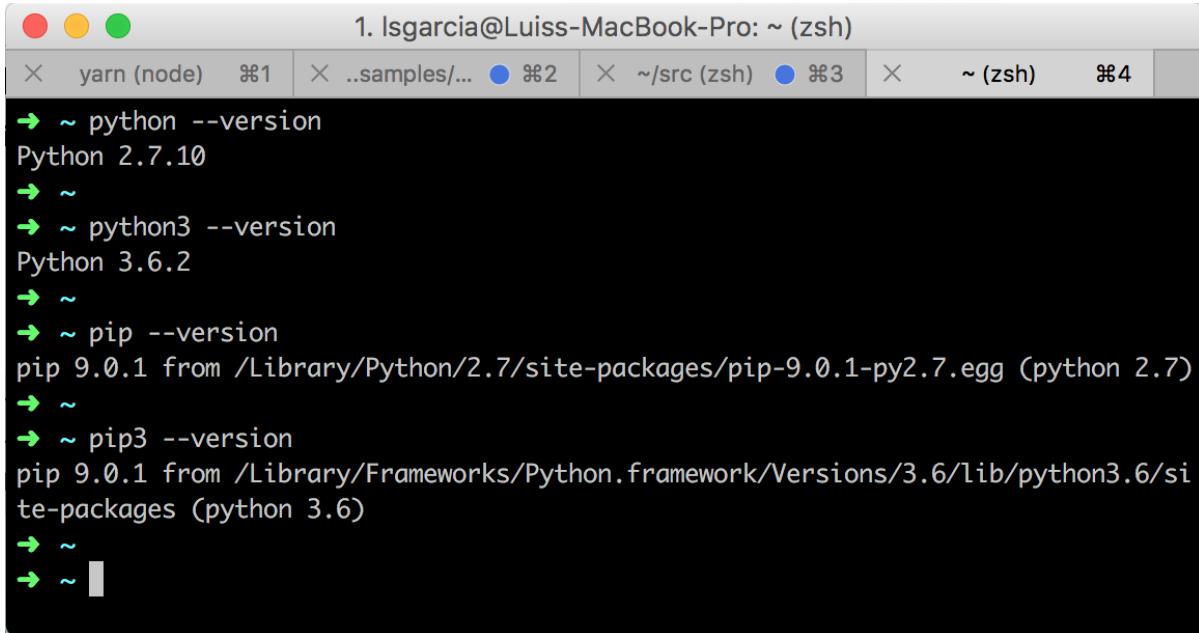
- Mac: la versión para Mac OS se encuentra en el mismo [link](#) que la de Windows. En el caso de Mac, el instalador de Python es el mismo para las versiones de 32 y 64 bits: '[Mac OS X 64-bit/32-bit installer](#)'. Esta instala por defecto la herramienta pip.
- Linux: en el caso de Linux, la instalación varía según la distribución que se está utilizando. En el caso de Ubuntu, Python3 viene preinstalado por defecto. Si la versión es inferior a 3.4, Python 3.6 se puede instalar utilizando el repositorio '**deadsnakes PPA**':


```
sudo add-apt-repository ppa:fkrull/deadsnakes
sudo apt-get update
sudo apt-get install python3.6
```

 Normalmente es necesario instalar pip manualmente con el siguiente comando:


```
sudo apt-get install python3-pip
```

Importante: los sistemas operativos Mac OS y Linux incluyen Python2 como una de las herramientas del sistema por lo que, con los comandos **python** o **pip** este utilizará automáticamente la versión de Python2. Para utilizar Python3 hay que utilizar los comandos **python3** y **pip3**. Con **python --version** y **pip --version** podemos asegurarnos que estamos utilizando la versión correcta.



```
1. lsgarcia@Luiss-MacBook-Pro: ~ (zsh)
x yarn (node) ⌘1 x ..samples/... ⌘2 x ~/src (zsh) ⌘3 x ~ (zsh) ⌘4
→ ~ python --version
Python 2.7.10
→ ~
→ ~ python3 --version
Python 3.6.2
→ ~
→ ~ pip --version
pip 9.0.1 from /Library/Python/2.7/site-packages/pip-9.0.1-py2.7.egg (python 2.7)
→ ~
→ ~ pip3 --version
pip 9.0.1 from /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages (python 3.6)
→ ~
→ ~
```

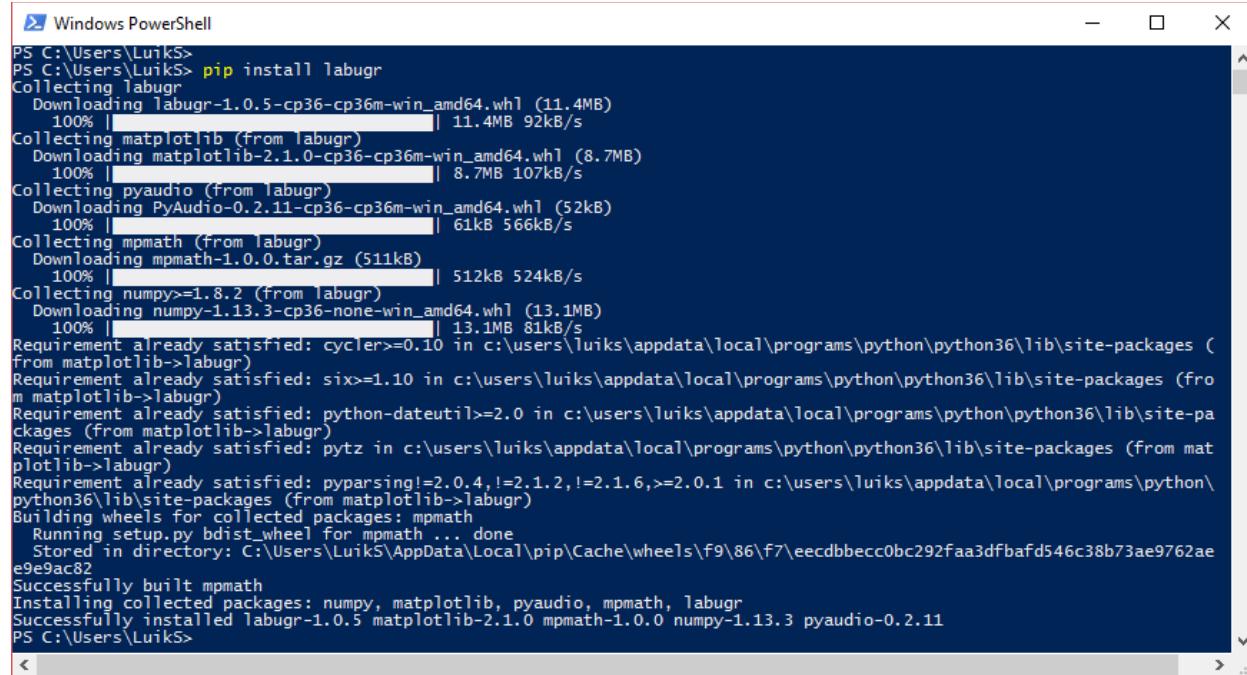
Figura 6: Captura de las diferencias entre pip-python/pip3-python3 en Mac OS

3.2 LabUGR

Para la utilización de la librería labUGR, esta debe ser instalada en el sistema. Uno de los principales objetivos del proyecto es simplificar la instalación de un entorno de trabajo completo en Python para el análisis de señales. El proceso de distribución de la librería es complicado y está descrito en el apartado '[Distribución](#)'. Gracias a este, el único paso necesario para la instalación es la ejecución de un comando en la terminal:

```
pip install labugr
```

La herramienta de instalación de paquetes en Python, pip, descargará automáticamente la versión de labUGR correspondiente al sistema operativo, así como las versiones apropiadas de los módulos externos utilizados por labUGR.



```
Windows PowerShell
PS C:\Users\Luiks> pip install labugr
Collecting labugr
  Downloading labugr-1.0.5-cp36-cp36m-win_amd64.whl (11.4MB)
    100% |██████████| 11.4MB 92kB/s
Collecting matplotlib (from labugr)
  Downloading matplotlib-2.1.0-cp36-cp36m-win_amd64.whl (8.7MB)
    100% |██████████| 8.7MB 107kB/s
Collecting pyaudio (from labugr)
  Downloading PyAudio-0.2.11-cp36-cp36m-win_amd64.whl (52kB)
    100% |██████████| 61kB 566kB/s
Collecting mpmath (from labugr)
  Downloading mpmath-1.0.0.tar.gz (511kB)
    100% |██████████| 512kB 524kB/s
Collecting numpy>=1.8.2 (from labugr)
  Downloading numpy-1.13.3-cp36-none-win_amd64.whl (13.1MB)
    100% |██████████| 13.1MB 81kB/s
Requirement already satisfied: cycler>=0.10 in c:\users\luiks\appdata\local\programs\python\python36\lib\site-packages (from matplotlib->labugr)
Requirement already satisfied: six>=1.10 in c:\users\luiks\appdata\local\programs\python\python36\lib\site-packages (from matplotlib->labugr)
Requirement already satisfied: python-dateutil>=2.0 in c:\users\luiks\appdata\local\programs\python\python36\lib\site-packages (from matplotlib->labugr)
Requirement already satisfied: pytz in c:\users\luiks\appdata\local\programs\python\python36\lib\site-packages (from matplotlib->labugr)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in c:\users\luiks\appdata\local\programs\python\python36\lib\site-packages (from matplotlib->labugr)
Building wheels for collected packages: mpmath
  Running setup.py bdist_wheel for mpmath ... done
  Stored in directory: C:\Users\Luiks\AppData\Local\pip\Cache\wheels\f9\86\f7\eeccdbbecc0bc292faa3dfbaf546c38b73ae9762ae
e9e9ac82
Successfully built mpmath
Installing collected packages: numpy, matplotlib, pyaudio, mpmath, labugr
Successfully installed labugr-1.0.5 matplotlib-2.1.0 mpmath-1.0.0 numpy-1.13.3 pyaudio-0.2.11
PS C:\Users\Luiks>
```

Figura 7: Captura de la instalación de labUGR en Windows

Otra forma de instalar la librería labUGR es utilizando la herramienta de instalación de paquetes incluida en PyCharm, el entorno de desarrollo integrado(IDE) más popular para Python. Entrar en el menú de opciones de intérprete, localizado en File--> Settings--> Project--> Project Interpreter. Pulsar el botón + (arriba a la derecha), buscar labUGR y pulsar en install packages.

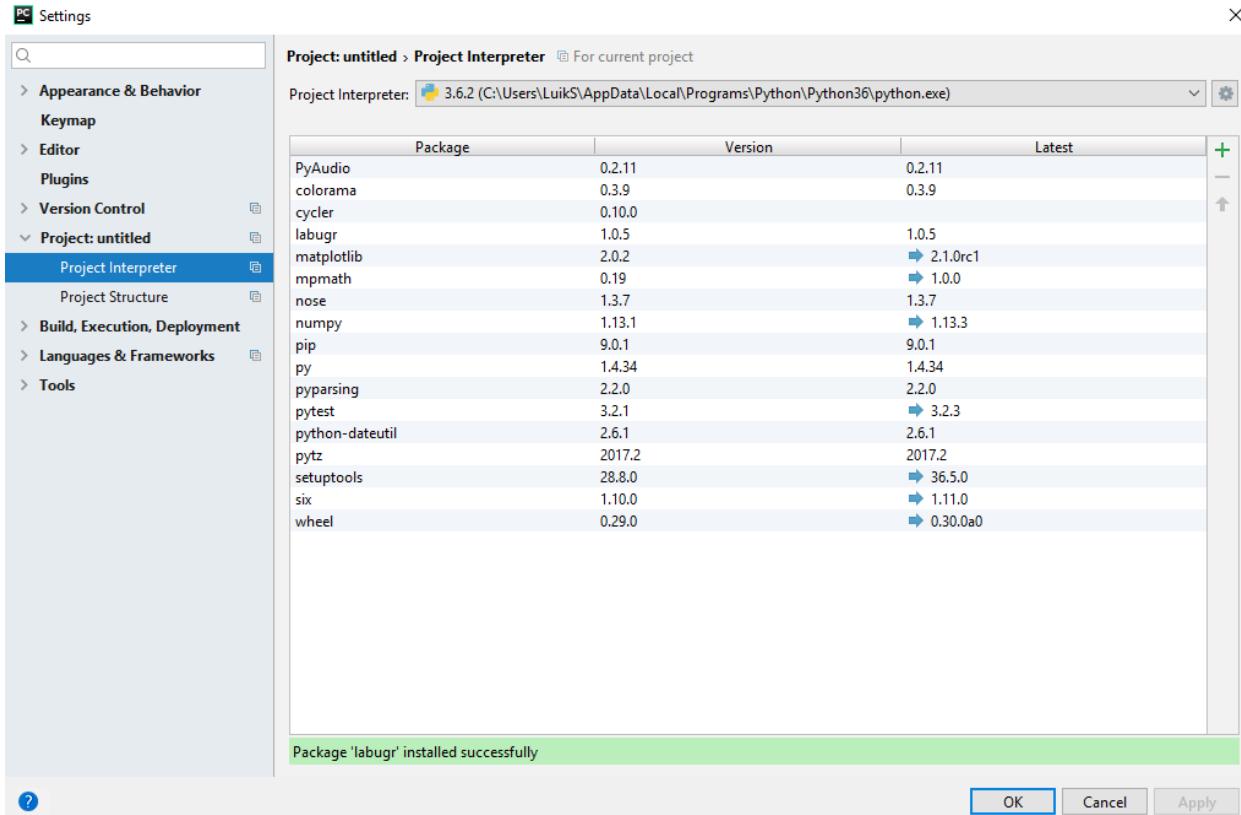


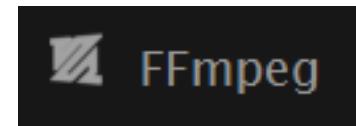
Figura 8: Instalación de labUGR a través de PyCharm

Una vez instalada la librería, es recomendable correr las pruebas para comprobar que todo funciona correctamente. Para ello se debe utilizar la función `test_all()`.

3.3 Dependencias externas

Para poder utilizar ciertas funciones del submódulo audio es necesario tener instaladas una serie de dependencias que no pueden ser incluidas dentro del paquete de labUGR. Estas dependencias son utilizadas por las funciones dentro del módulo para poder tratar con archivos de audio codificados, es decir, archivos con formatos mp3, mp4, aac, ...

- ‘Backend’ de audio: para poder decodificar archivos de audio comprimidos es necesario tener instalado un ‘backend’ en el sistema. Las funciones incluidas en labUGR soportan una serie de diferentes backends para los diferentes sistemas. Por experiencia propia, los más fáciles de instalar son los siguientes:
 - Windows – FFmpeg: basta con descargar el archivo correspondiente al sistema (<https://ffmpeg.zeranoe.com/builds/>) descomprimir la carpeta `bin` en la variable de entorno PATH de Windows.
 - Linux - FFmpeg: instalado con sudo ‘`apt-get install ffmpeg`’.
 - Mac OS – Core Audio: por defecto, Mac incluye por defecto un backend de audio pero si este produce problemas también se puede instalar FFmpeg a través del link anterior.



- PortAudio: PortAudio es una librería multiplataforma de código abierto que permite interactuar con la entrada y la salida de audio del sistema a través de varios lenguajes de programación. Esta es utilizada por el módulo Pyaudio para poder reproducir archivos mp3 o wav desde Python. En Windows no es necesario instalar nada; en Mac OS basta con ejecutar '**brew install portaudio**'; en Linux '**sudo apt-get install portaudio19-dev**'.



La función `test_all()` dentro de la librería labUGR comprueba automáticamente si estas dependencias están instaladas en el sistema y si no es así, muestra en pantalla las instrucciones a seguir para instalarla dependiendo del sistema operativo del usuario.

4 Uso

Uno de los objetivos principales de este proyecto es la creación de un entorno de trabajo contenido desde el cual se pueda acceder a todas las funciones necesarias para el análisis de señales, sin necesidad de tener que importar diferentes librerías y saber en cual se encuentra cada función. Por ello, todas las funciones incluidas en el módulo labUGR, así como las importadas de NumPy y Matplotlib, se pueden acceder al importar la librería labUGR. Una vez que Python y labUGR están instalados en el sistema, basta con importar la librería para utilizar sus funciones.

En Python existen tres formas de importar funciones de un módulo:

- **Método I:** Con el comando '`import labugr`' se importa el módulo entero al entorno. Para utilizar las funciones se utiliza el nombre del módulo como prefijo; por ejemplo, '`labugr.freqs()`' para la función '`freqs`'.
- **Método II:** Con el comando '`from labugr import {funciones}`' se importan funciones específicas del módulo; por ejemplo, '`from labugr import freqs, freqz`' para importar las funciones '`freqs`' y '`freqz`'. Estas se utilizan simplemente con su nombre.
- **Método III:** Con el comando '`from labugr import *`' se importan todas las funciones incluidas dentro del módulo al entorno de trabajo.

Importar un módulo o funciones de el en Python no consume recursos ya que todos los módulos están completamente definidos en el directorio de instalación de Python. El código no va a ser más ligero si el usuario importa funciones concretas en vez de importar el módulo entero, la única diferencia es como se ligan los nombres de las funciones. En el primer método solo se reserva '`labugr`' en el espacio de nombres de variables, pero en los métodos II y III se reservan todos los nombres de las funciones.

Utilizar el método III no está recomendado ya que puede provocar conflictos entre funciones de diferentes módulos y ocultar algunas de ellas. Por ejemplo, si se importa una función del módulo 'a' y otra función del módulo 'b' que tiene el mismo nombre la primera quedará ocultada y solo se podrá utilizar la segunda ya que ha sido la última en importarse. A pesar de ello, la mejor forma de utilizar el módulo labUGR es a través del método III.

from labugr import *

Esta librería está diseñada para crear un entorno contenido, sin problemas de conflictos entre nombres de funciones.

5 Control de versiones

El software utilizado para el control de versiones en este proyecto ha sido Git. Actualmente, Git es la herramienta más popular para el control de versiones en el desarrollo de software, utilizada en un gran número de proyectos tanto comerciales como de código abierto. Este sistema registra los cambios realizados sobre un proyecto a lo largo del tiempo, de modo que se pueden recuperar versiones específicas más adelante y realizar comparaciones entre ellas.



Git está especialmente pensado para facilitar el desarrollo de software en donde un grupo de desarrolladores trabaja en un mismo proyecto. Aun así, Git, junto con un repositorio remoto en GitHub (<https://github.com/lserraga/labUGR>) han sido esenciales para el desarrollo de este proyecto. En la siguiente figura se puede observar el progreso del repositorio a lo largo de su desarrollo:



Figura 9: Gráfico de los 'commits' en la rama master del repositorio remoto de labUGR en GitHub

Con un total de 108 **commits** y 7 ramas diferentes, Git me ha permitido tener un mayor control del código y me ha facilitado el hecho de tener el código actualizado en las diferentes plataformas de desarrollo. Debido a las diferencias entre Mac OS, Linux y Windows, en repetidas ocasiones las funciones añadidas durante el desarrollo en una de las plataformas han provocado que la librería dejase de funcionar en alguna de las otras plataformas.

Para solventar esto y, siguiendo los estándares de uso de Git, he establecido la rama master del repositorio como la rama estable (definiendo la versión como estable si compila en Linux, Mac OS y Windows) y he trabajado en una rama de desarrollador, solo unificando las dos cuando la versión es estable en los tres sistemas. En el siguiente esquema se puede observar la rama de desarrollador en azul y master en negro:

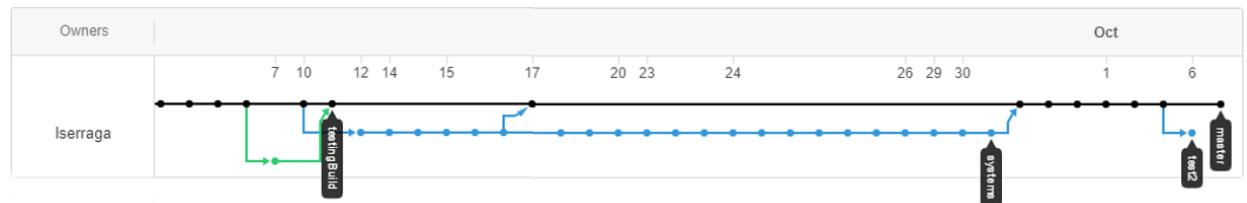


Figura 10: Esquema de las diferentes ramas del repositorio remoto de labUGR en GitHub

6 Estructura

El módulo utilizado para la creación de este proyecto ha sido ***numpy.distutils*** (<https://docs.scipy.org/doc/numpy-1.13.0/reference/distutils.html>). Este es un subpaquete del módulo NumPy de Python que ha sido creado a partir de la herramienta de creación de paquetes ***distutils***, la cual está incluida en Python. Este añade nuevas funciones que facilitan la creación de subpaquetes, el manejo de código que no es de Python, creación de extensiones, ... Se han seguido los estándares para el empaquetado de módulos descritos por PyPA(Python Packaging Authority) en <https://packaging.python.org/tutorials/distributing-packages/>.

El archivo ***setup.py*** del directorio principal es el archivo central del proyecto, en él se establece la configuración del proyecto y se realizan una serie de comprobaciones iniciales para hacer posible la instalación del proyecto. También incluye la información acerca del paquete que será utilizada al distribuirlo. Este es el archivo utilizado por pip a la hora de instalar el paquete.

Este proyecto está dividido en diferentes submódulos para hacer una clara separación de las diferentes funcionalidades, facilitar el uso de tests para comprobar el correcto funcionamiento de la librería y facilitar la adición de nuevos módulos en el futuro. En el caso del submódulo `fftpack`, este está copiado de SciPy en su totalidad, el resto de submódulos han sido creados por mí utilizando funciones de Matplotlib, NumPy, mpmath, SciPy, PyAudio y AudioRead. LabUGR sigue la estructura definida en la figura 11:

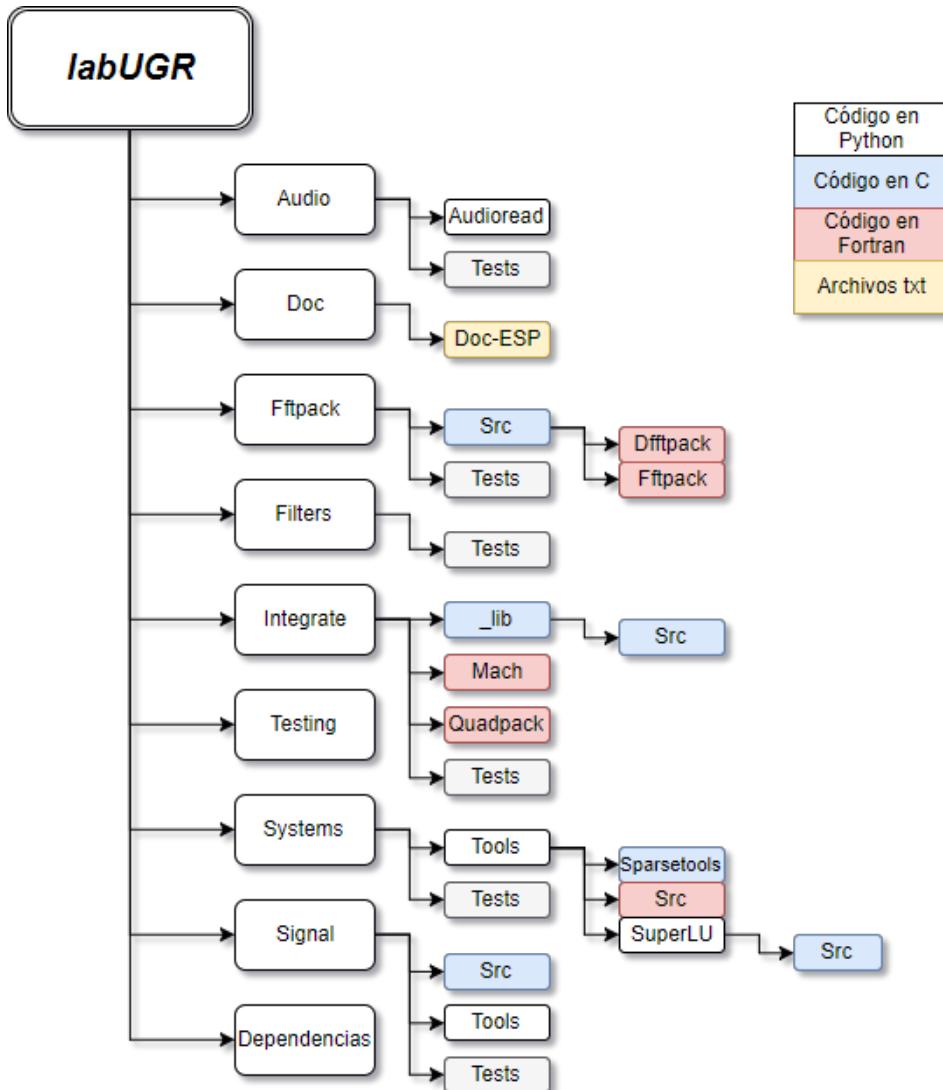


Figura 11: Estructura de la librería labUGR

Dentro de la carpeta labUGR se encuentran todos los submódulos. Cada submódulo, así como el módulo principal, tienen 2 archivos esenciales:

- **`__init__.py`**: este archivo será utilizado por Python para determinar que funciones serán incluidas al importar el módulo desde otra ubicación. En el caso de `labugr__init__.py`, se importan todas las funciones deseadas de los submódulos de labUGR, así como las funciones importadas de Matplotlib y NumPy. Cuando el usuario final importe labUGR, estas funciones serán las que se añadirán al entorno de trabajo y podrán ser utilizadas por él.

Los archivos `__init__.py` encontrados dentro de las carpetas del resto de submódulos contienen las funciones que se desean exportar de cada uno de ellos. Todas las funciones que no están incluidas en este archivo quedan escondidas del usuario final. A su vez, dentro de la mayoría de archivos con código Python se incluye una variable `__all__` que define un vector con el nombre de las funciones a exportar de ese archivo. Por ejemplo, esta es la forma de `labugr\filters__init__.py`:

```

from .filters import *
from .fir_filters import *
from .iir_filters import (
    butter, cheby1, cheby2, iirfilter)
from .spectral import *

excluded = ['excluded', 'filters', 'fir_filters', 'iir_filters', 'helpers',
'spectral']
__all__ = [s for s in dir() if not (s in excluded)]

```

En él se incluyen las funciones definidas en las variables `__all__` de `labugr\filters\filters.py`, `labugr\filters\fir_filters.py` y `labugr\filters\spectral.py`; añade las funciones butter, cheby1, cheb2 y iirfilter de `labugr\filters\iir_filters.py`; excluye los nombres de los archivos, que por defecto se incluyen en el entorno.

- **`setup.py`**: al igual que en el módulo principal, el archivo `setup.py` define la configuración del submódulo. Por ejemplo, esta es la forma de `labugr\signal\setup.py`:

```

def configuration(parent_package='', top_path=None):
    from numpy.distutils.misc_util import Configuration

    config = Configuration('signal', parent_package, top_path)

    config.add_data_dir('tests') # Como tests no es considerado como un
                                # paquete, hay que anadirlo como
                                # directorio de datos
    config.add_subpackage('tools')

    config.add_extension('sigtools',
        sources=['src/sigtoolsmodule.c', 'src/firfilter.c',
                 'src/medianfilter.c', 'src/lfilter.c.src',
                 'src/correlate_nd.c.src'],
        depends=['src/sigtools.h'],
        include_dirs=['src/'])

    return config

if __name__ == '__main__':
    from numpy.distutils.core import setup
    setup(**configuration(top_path='').todict())

```

En él, se define el nombre del subpaquete ('signal'); se añade un directorio de datos para tests para que estos sean importados durante la instalación de la librería; se añade otro subpaquete que será utilizado por las funciones de este submódulo ('tools'); añade una nueva extensión de código en c que genera una serie de funciones en Python, que el resto del submódulo puede utilizar.

Python, al ser un lenguaje de programación de muy alto nivel y dinámico, es excelente para escribir código fácil de entender y sencillo. En cambio, a la hora de realizar operaciones de bajo nivel, como operaciones matemáticas en bucles ‘for’ o ‘while’, este puede llegar a ser hasta 100 veces más lento que el mismo código programado en lenguajes estáticos como C y Fortran. Es por ello que la mayoría de funciones del módulo labUGR utilizan módulos escritos en C y Fortran para realizar las operaciones de bajo nivel. En las estadísticas del repositorio de GitHub de labUGR se puede comprobar que solo alrededor del 18% del código incluido en la librería es Python:



Figura 12: Distribución de los diferentes lenguajes de programación utilizados en labUGR

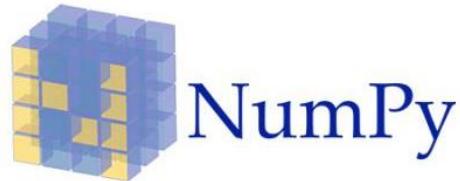
Estas extensiones se añaden a través de los archivos setup.py de los subpaquetes, utilizando los módulos **ctypes**(C y C++) y **f2py**(FORTRAN 77/90/95), que incluye Python por defecto, como interfaz entre el código compilado en C/Fortran y el código en Python. Esta librería utiliza también Cython para generar parte de ese código en C.

7 Submódulos

En este apartado se describen con detalle los submódulos de los que se compone la librería labUGR, incluyendo una descripción de las funciones que componen cada submódulo, así como una serie de ejemplos. También se nombran las funciones importadas de los módulos NumPy y Matplotlib, con una pequeña descripción (en la ‘práctica 0’ del apartado [Prácticas](#) se incluyen ejemplos de utilización de muchas de estas funciones).

7.1 Funciones importadas de NumPy

NumPy (<http://www.numpy.org/>) es uno de los paquetes fundamentales para el cálculo numérico en Python. Este añade un mayor soporte para el uso de matrices y vectores, así como un gran repertorio de funciones matemáticas de alto nivel.



7.1.1 Funciones para la creación y manipulación de vectores y matrices

- Creación de matrices: `array(object)`, matriz especificando elemento a elemento; `zeros(shape)`, matriz de ceros con dimensiones igual a ‘`shape`’, `ones(shape)`, matriz de unos; `eye(shape)`, matriz unitaria; `full(shape, value)`, matriz con todos los valores iguales a ‘`value`’; `fromfunction(function, shape)`, matriz con valores obtenidos a partir de una función ‘`function`’; `copy(a)`, matriz copia de la matriz ‘`a`’ (esta función es importante ya que utilizando `x = y`, tanto `x` como `y` representan la misma matriz y un cambio en `y` se verá reflejado en `x`).
- Creación de matrices a partir de distribuciones: `arange(start, stop, step)`, valores espaciados uniformemente dentro de un intervalo entre ‘`start`’ y ‘`stop`’ y una separación entre valores ‘`step`’; `linspace(start, stop, num)`, igual que `arange` pero especificando el número de valores ‘`num`’ y no la distancia entre ellos; `logspace(start, stop, num)`, valores espaciados uniformemente en una escala logarítmica; `geomspace(start, stop, num)`, valores espaciados uniformemente en una escala geométrica.
- Manipulación de matrices: `reshape(a, newshape)`, transforma la matriz ‘`a`’ en una matriz con dimensiones ‘`newshape`’; `transpose(a)`, transpuesta de una matriz ‘`a`’; `concatenate(a1, a2)`, concatena dos matrices ‘`a1`’ y ‘`a2`’; `delete(arr, obj)`, elimina la submatriz o columnas/filas en la posición ‘`obj`’ dentro de una matriz ‘`arr`’; `insert(arr, obj, values)`, añadir la submatriz o columnas/filas definida por ‘`values`’ en la posición ‘`obj`’ dentro de una matriz ‘`arr`’; `sort(a)`, devuelve una versión ordenada de la matriz ‘`a`’; `split(ary, num)`, dividir una matriz ‘`ary`’ en ‘`num`’ submatrices con las mismas dimensiones.

- Análisis de matrices: `unique(ar)`, devuelve los valores únicos dentro de una matriz ‘`ar`’; `where(condition)`, devuelve las posiciones de los valores que cumplen una condición ‘`condition`’ dentro de la matriz; `shape(a)`, forma de una matriz (número de columnas, filas, ...); `ndim(a)`, dimensión de la matriz; `size(a)`, tamaño de una matriz (número de elementos); `real(val)`, parte real de los elementos de la matriz o elemento ‘`val`’; `imag(val)`, parte imaginaria de los elementos de la matriz o elemento ‘`val`’; `angle(val)`, fase en radianes de los elementos de la matriz o elemento ‘`val`’; `conj(val)`, conjugado de los elementos de la matriz o elemento ‘`val`’; `diag(v, k=0)`, diagonal principal o diagonal secundaria ‘`k`’ de una matriz ‘`v`’.
- Comparación de matrices: `allclose(a, b, atol)`, comprueba si todos los valores de las matrices ‘`a`’ y ‘`b`’ son iguales con una tolerancia absoluta ‘`atol`’; `isclose(a, b, atol)`, comprueba uno por uno si los valores de dos matrices son iguales con una cierta tolerancia, devuelve una matriz con valores booleanos (True/False); `all(a)`, comprueba si todos los valores de una matriz ‘`a`’ son True, también se puede pasar una condición (`a>2`); `any(a)`, comprueba si alguno de los valores de una matriz ‘`a`’ son True, también se puede pasar una condición (`a>2`); `maximum(x1, x2)`, devuelve una matriz con los valores máximos elemento a elemento entre dos matrices ‘`x1`’ y ‘`x2`’ de mismas dimensiones; `minimum(x1, x2)`, devuelve una matriz con los valores mínimos elemento a elemento entre dos matrices ‘`x1`’ y ‘`x2`’ de mismas dimensiones.

7.1.2 Funciones matemáticas

Importadas de NumPy, estas funciones adaptan las funciones matemáticas básicas, la mayoría de ellas ya incluidas en el módulo ‘math’, para que sean compatibles con las matrices y vectores de NumPy. Estas funciones, al igual que los operadores matemáticos tradicionales (+, -, *, /, <, >, ...) operan con matrices elemento a elemento, a excepción de funciones específicas como `dot` y `matrix_power`.

- Funciones trigonométricas: `sin(x)`, `cos(x)`, `tan(x)`, `arcsin(x)`, `arccos(x)`, `arctan(x)`, `sinc(x)`.
- Funciones hiperbólicas: `sinh(x)`, `cosh(x)`, `tanh(x)`, `arcsinh(x)`, `arccosh(x)`, `arctanh(x)`.
- Conversión de ángulos: `deg2rad(x)`, de grados decimales a radianes; `rad2deg(x)`, de radianes a grados decimales.
- Redondeo: `floor(x)`, redondeo por defecto de ‘`x`’; `ceil(x)`, redondeo por exceso; `round(x, decimals)`, redondeo estándar a ‘`decimals`’ cifras.
- Algebra lineal: `matrix_power(M, n)`, potencia ‘`n`’ de una matriz cuadrada ‘`M`’ (el operador `**` actúa elemento a elemento); `det(a)`, determinante de una matriz ‘`a`’; `solve(a, b)`, función que resuelve un sistema de ecuaciones `a*x=b`; `inv(a)`, inversa multiplicativa de una matriz ‘`a`’ (esta debe ser invertible); `outer(a, b)`, producto exterior de dos matrices ‘`a`’ y ‘`b`’.
- Funciones lógicas: `logical_and(x1, x2)`, operador lógico and entre los elementos de las matrices ‘`x1`’ y ‘`x2`’ (elemento a elemento); `logical_or(x1, x2)`, operador lógico or; `logical_not(x1, x2)`, operador lógico not; `logical_xor(x1, x2)`, operador lógico xor.

- Variables: inf, ; pi, ; e, .
- Otras funciones: `exp(x)`, exponencial de 'x'; `log(x)`, logaritmo natural; `log10(x)`, logaritmo en base 10; `log2(x)`, logaritmo en base 2; `sqrt(x)`, raíz cuadrada; `abs(x)`, valor absoluto; `sum(x)`, sumatorio de todos los elementos; `gradient(x)`, gradiente; `cross(a, b)`, producto vectorial cruzado de dos vectores 'a' y 'b'; `dot(a, b)`, multiplicación matricial de dos matrices compatibles 'a' y 'b' (el operador * actúa elemento a elemento).

Las funciones descritas contienen un gran número de parámetros opcionales que se pueden consultar en su documentación oficial encontrada en los vínculos agregados a los nombres de las funciones. El resto de funciones incluidas en el módulo NumPy (<https://docs.scipy.org/doc/numpy/reference/routines.html>) se pueden utilizar utilizando el prefijo '`np.`' (por ejemplo `np.diagflat`)

7.2 Funciones importadas de Matplotlib

Matplotlib (<https://matplotlib.org/>) es otro de los módulos fundamentales para esta librería. Este permite producir gráficos de una manera sencilla y muy similar a como se realiza en Matlab. Matplotlib contiene un gran número de funcionalidades, en labUGR se han incluido las esenciales para la introducción al análisis de señales en Python:



- Distribución de gráficos: `figure()`, creación de una nueva figura (nueva ventana); `subplot(nrows, ncols, plot_number)`, divide la figura en una tabla con dimensiones '`nrows`'x'`ncols`' seleccionando la posición '`plot_number`' para mostrar la gráfica; `show()`, mostrar todas las figuras en pantalla; `close()`, cerrar las figuras.
- Tipos de gráficos: `plot(x, y, form)`, gráfico estándar de dos vectores 'x' e 'y' utilizando la forma definida en '`form`' (por ejemplo 'bo' para círculos azules); `bar(x, y)`, gráfico de barras verticales; `barh(x, y)`, gráfico de barras horizontales; `stem(x, y)`, gráfico de líneas verticales; `step(x, y)`, gráfico con puntos unidos mediante escalones; `semilogy(x, y)`, gráfico estándar con el eje vertical en escala logarítmica; `semilogx(x, y)`, gráfico estándar con el eje horizontal en escala logarítmica; `vlines(x, ymin, ymax)`, añade líneas verticales entre '`ymin`' y '`ymax`' en las posiciones definidas en 'x'; `hlines(y, xmin, xmax)`, añade líneas horizontales entre '`xmin`' y '`xmax`' en las posiciones definidas en 'y'.
- Configuración de los ejes: `axis(type)`, asigna '`type`' a las características de los ejes ('off', 'equal', 'tight'); `xscale(scale)`, escala del eje horizontal ('linear', 'log', 'logit', 'symlog'); `yscale(scale)`, escala del eje vertical; `xlim(xmin, xmax)`, establece los límites del eje horizontal; `ylim(xmin, xmax)`, establece los límites del eje vertical; `xticks(ticks)`, establecer las etiquetas del eje horizontal; `yticks(ticks)`, establecer las etiquetas del eje vertical.

- **Dfb:** `title(text)`, título de el gráfico; `legend(labels)`, leyenda del gráfico utilizando las etiquetas en ‘label’ según el orden en el que se han graficado las funciones (también se puede definir una etiqueta con `label=""` al crear un gráfico con `plot`, `stem` ,...); `xlabel(label)`, descripción del eje horizontal; `ylabel(label)`, descripción del eje vertical; `grid()`, modifica las líneas de división de un gráfico.

Las funciones descritas contienen un gran número de parámetros opcionales que se pueden consultar en su documentación oficial encontrada en los vínculos agregados a los nombres de las funciones. El resto de funciones incluidas en el módulo pyplot de Matplotlib (https://matplotlib.org/api/pyplot_api.html) se pueden utilizar utilizando el prefijo ‘`plt.`’ (por ejemplo `plt.fill`).

7.3 Integración

El submódulo **`integrate`** contiene una serie de funciones para la integración numérica basadas en las librerías de fortran quadpack y mach.

- `quad(func, a, b, args=(), full_output=0, epsabs=1.49e-8, epsrel=1.49e-8, limit=50, points=None, weight=None, wvar=None, wopts=None, maxp1=50, limlst=50)`: función para la integración numérica de una dimensión de una función **`func(x)`**, entre los puntos **`a`** y **`b`**. Este intervalo puede incluir infinitos (**`-infinity`** y **`+infinity`**). El resto de argumentos son opcionales e incluyen opciones avanzadas para definir la tolerancia de los errores en los cálculos, introducir puntos de interrupción en posibles discontinuidades en la integración, ...
- `dblquad(func, a, b, gfun, hfun, args=(), epsabs=1.49e-8, epsrel=1.49e-8)`: función para la integración doble de una función **`func(y, x)`**, entre los puntos **`a`** y **`b`** para ‘**`x`**’ y entre los puntos **`gfunc(x)`** y **`hfunc(x)`** para ‘**`y`**’. La función **`gfunc`** determina el límite inferior y **`hfunc`** el superior para la curva de ‘**`y`**’.
- `tplquad(func, a, b, gfun, hfun, qfun, rfun, args=(), epsabs=1.49e-8, epsrel=1.49e-8)`: función optimizada para la integral triple de una función **`func(z, y, x)`**, **`J`**, entre los puntos **`a`** y **`b`** para ‘**`x`**’, entre los puntos **`gfun(x)`** y **`hfun(x)`** para ‘**`y`**’ y entre los puntos **`qfun(x,y)`** y **`rfun(x,y)`** para ‘**`z`**’.

Estas tres funciones devuelven el valor numérico de la integración, así como una estimación del error absoluto en el resultado. Ejemplos:

- Integral de $x^5 e^{-x}$ entre $x=0$ y $x=+\infty$. Esta es una integral conocida y el resultado es la factorial del exponente de x , en este caso, $\int_0^\infty x^5 e^{-x} dx = 5! = 120$.

```
>>> def funcion(x):
...     return x**5 * e**(-x)
...
>>> resultado, error = quad(funcion, 0, np.inf)
>>> print(resultado, error)
120.0000000000003 2.2281813317356577e-07
```

- Integral doble de xy entre $x=0$ y $x=0.5$, con límites de integración $y=2$ y $y=1-2x$. En este caso, los límites superiores para y se tienen que definir como funciones con parámetro x . Una forma más compacta para definir funciones en Python es utilizando lambdas (http://www.secnetix.de/olli/Python/lambda_functions.hawk). Analíticamente, el resultado es $\int_{x=0}^{0.5} \int_{y=0}^{1-2x} xy dy dx = \frac{1}{96} = 0.010416$

```
>>> funcion = lambda y, x: x*y
>>> limiteSuperior = lambda x: 1-2*x
>>> limiteInferior = lambda x: 0
>>> resultado, error = dblquad(funcion, 0, 0.5, limiteInferior,
limiteSuperior)
>>> print(resultado, error)
0.01041666666666668 4.101620128472366e-16
```

7.4 Transformada de Fourier

El submódulo fftpack contiene una serie de funciones para la transformada discreta de Fourier, la transformada de Hilbert, transformadas de Fourier reales y transformadas de Fourier de tiempo reducido. Principalmente están extraídas del módulo fftpack de SciPy, a excepción de stft y istft que provienen del submódulo signal de SciPy y las funciones fftshift, ifftshift y fftfreq que provienen del módulo fft de NumPy. Las siguientes funciones están incluidas en labUGR:

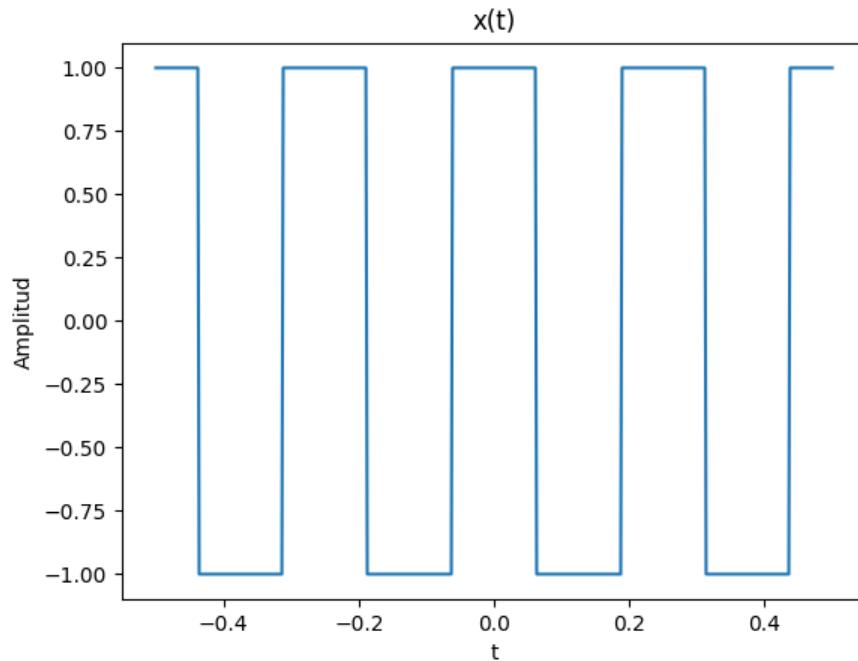
- fft(x, n=None, axis=-1, overwrite_x=False)**: función para obtener la transformada discreta de Fourier de una señal x compleja o real, utilizando el algoritmo de la transformada rápida de Fourier. Los valores de la transformada se obtienen utilizando la sumatoria siguiente: $y[j] = \sum_{k=0}^{n-1} x[k] * e^{\frac{-2\pi i j k}{n}}$ para $j = 0 \dots n - 1$. Si n no se especifica, esta es igual al tamaño de x ; si n es menor que el tamaño de x , esta se acorta; y si es mayor, se añaden ceros. Esta función devuelve un espectro en frecuencia simétrico de la señal x , con tanto frecuencias negativas como positivas. Si n no se especifica, los valores del resultado desde 1 a $n/2$ son las frecuencias positivas, de $n/2$ a $n-1$ las frecuencias negativas y el primer término $y[0]$ el valor de frecuencia 0. El resultado está formado por números complejos.

- **`ifft(x, n=None, axis=-1, overwrite_x=False)`**: función para obtener la inversa de la transformada discreta de Fourier de una señal x compleja o real, utilizando el algoritmo de la transformada rápida de Fourier. Los valores de la transformada se obtienen utilizando la sumatoria siguiente: $y[j] = \frac{\sum_{k=0}^{n-1} x[k] * e^{\frac{2\pi i * j * k}{n}}}{n}$ para $j = 0 \dots n - 1$. Al igual que `fft()`, con n se puede especificar el tamaño de la transformada inversa.
- **`fftn(x, shape=None, axes=None, overwrite_x=False)`**: función para obtener la transformada discreta de Fourier de n dimensiones de una matriz x de dimensión n.
- **`ifftn(x, shape=None, axes=None, overwrite_x=False)`**: función para obtener la inversa de la transformada discreta de Fourier de n dimensiones de una matriz x de dimensión n.
- **`fftshift(x, axes=None)`**: con esta función podemos reordenar el resultado de una transformada de Fourier para que el espectro de la señal esté centrado en la frecuencia de valor 0.
- **`ifftshift(x, axes=None)`**: función inversa a `fftshift()`, reordena el espectro para que empiece en la frecuencia 0.
- **`fftfreq(n, d=1.0)`**: función para generar un vector de frecuencias de longitud n para poder representar correctamente los valores de la transformada de Fourier. Con d especificamos la distancia entre los valores del vector de frecuencias (periodo de muestreo). El vector generado puede tener dos formas dependiendo si n es par: $f = [0, 1, \dots, n/2-1, -n/2, \dots, -1] / (d*n)$; o si n es impar: $f = [0, 1, \dots, (n-1)/2, -(n-1)/2, \dots, -1] / (d*n)$. Si utilizamos esta función para generar el vector de frecuencias, no es necesario utilizar la función `fftshift()`, ya que al graficar la transformada, esta se muestra automáticamente centrada en 0.

Ejemplo de la transformada rápida de Fourier de una señal periódica y par cuadrada de 3Hz muestreada a 50Hz durante 1 segundo:

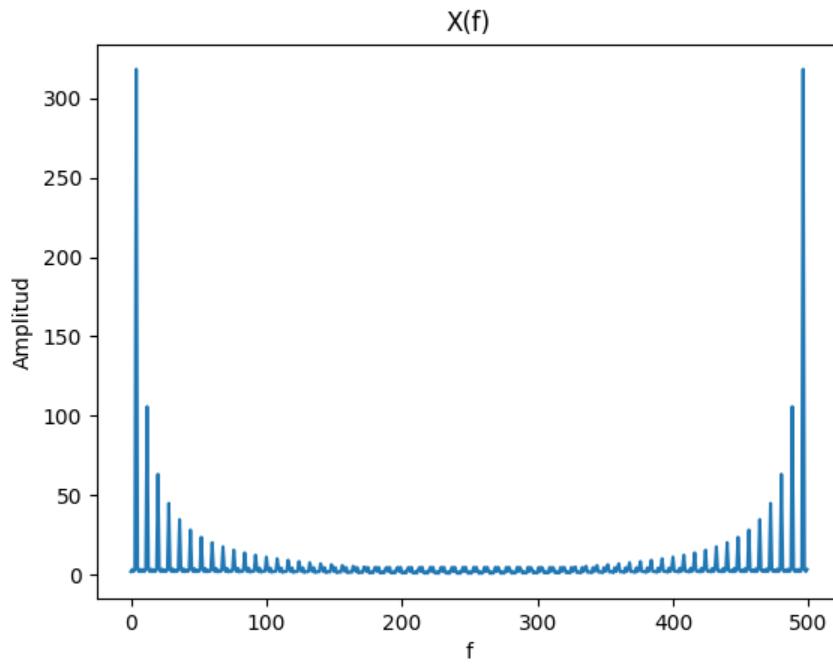
Primero creamos el vector de tiempos y la señal cuadrada (desplazada para que sea par):

```
Fs = 500
f = 4
T = 1/f
t = linspace(0, 1, 1*Fs)
h = square(2*pi*f*(t+1/4))
```



Con `fft()` calculamos la transformada de Fourier. Al complejo el resultado, utilizamos `abs()` para obtener el módulo de esos valores:

```
H = abs(fft(h))
```



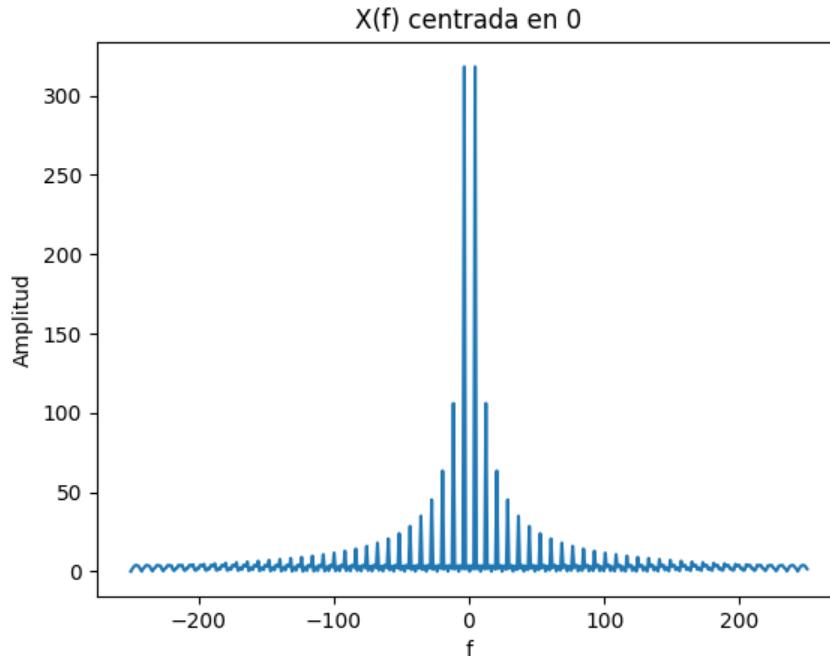
A la izquierda se encuentra el espectro positivo de la señal y a la derecha el negativo. Esta representación es confusa por lo que es mejor o centrar el espectro en 0 o representar solo la parte positiva. Para centrar la representación en 0 podemos utilizar `fftfreq()` o `fftshift()`.

Con `fftshift()` necesitamos crear un vector de frecuencias desde $-Fs/2$ a $Fs/2$ con una longitud igual al vector de tiempos:

```
f = linspace(-Fs/2, Fs/2, len(t))
H0 = np.fft.fftshift(H)
```

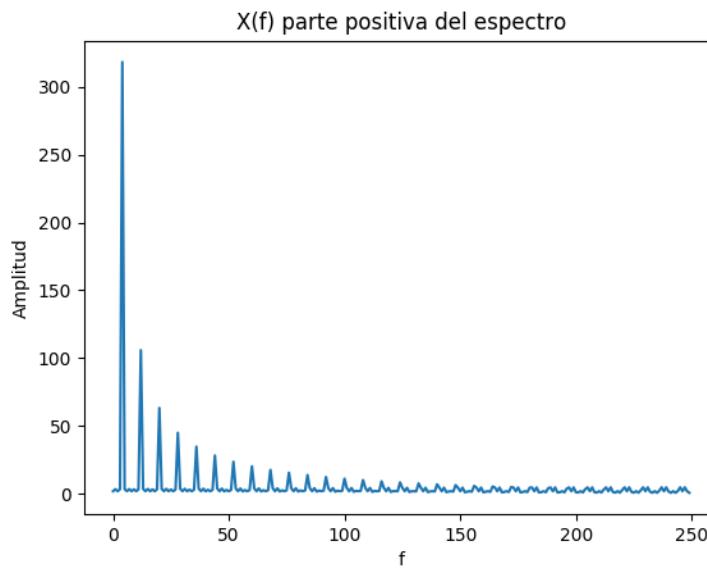
Con el vector de frecuencias creado con `fftfreq()` basta con graficar la transformada con este vector para que aparezca centrado. El tamaño del vector debe ser igual al del vector de tiempos y la separación entre muestras es el periodo de muestreo ($1/Fs$):

```
f = np.fft.fftfreq(len(t), 1/Fs)
```



Para solo mostrar la parte positiva del espectro basta con solo graficar los valores de la transformada de 0 a $Fs/2$:

```
plot(H[:250])
```



El desarrollo en serie de Fourier de una señal periódica par solo tiene coeficientes impares. Como podemos observar en la gráfica, los coeficientes se encuentran en $1f_0, 3f_0, 5f_0, \dots$ siendo f_0 la frecuencia de la señal cuadrada(4Hz). Para ver el código fuente → ejemplo_fft.py.

7.4.1 Transformadas reales

Las transformadas del seno y del coseno son formas de la transformada de Fourier que solo utilizan números reales.

- `dct(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False)`: función para obtener la transformada discreta del coseno de una señal x. A diferencia de la transformada de Fourier, la `dct()` utiliza solo los términos del coseno. Hay 8 tipos de dcts, de los cuales los tres primeros están disponibles (determinado con el parámetro `type`), estos utilizan las siguientes sumatorias:

- Tipo I: $y[k] = x[0] + (-1)^k x[n - 1] + 2 \sum_{j=0}^{n-2} x[j] \cos\left(\frac{\pi k j}{n-1}\right)$ para $k = 0 \dots n$.
- Tipo II: $y[k] = 2 \sum_{j=0}^{n-1} x[j] \cos\left(\frac{\pi(2j+1)k}{2n}\right)$ para $k = 0 \dots n$.
- Tipo III: $y[k] = x[0] + 2 \sum_{j=1}^{n-1} x[j] \cos\left(\frac{\pi j(2k+1)}{2n}\right)$ para $k = 0 \dots n$.

Con el parámetro `norm`, podemos utilizar la normalización ortogonal (solo en los tipos II y III). Esta transformada es prácticamente equivalente a la transformada de Fourier discreta de una secuencia real y par de doble longitud. Esta transformada se utiliza principalmente para compresión de audio y video. Es mucho más eficiente que la transformada del seno en esta aplicación ya que se necesitan menos coeficientes del coseno para aproximar señales comunes.

- `idct(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False)`: función para obtener la inversa de la transformada discreta del coseno de una señal x. Los tipos de transformadas discretas del coseno están relacionadas, por ejemplo, la tipo II es la inversa de la tipo III con un factor $2n$. `Idct()` se encarga de mapear estas relaciones para obtener la transformada inversa.

- `dst(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False)`: función para obtener la transformada discreta del seno de una señal x. A diferencia de la transformada de Fourier, la `dst()` utiliza solo los términos del seno. Hay 8 tipos de dcts, de los cuales los tres primeros están disponibles (determinado con el parámetro `type`), estos utilizan las siguientes sumatorias:

- Tipo I: $y[k] = 2 \sum_{j=0}^{n-1} x[j] \sin\left(\frac{\pi(j+1)(k+1)}{n+1}\right)$ para $k = 0 \dots n$.
- Tipo 2: $y[k] = 2 \sum_{j=0}^{n-1} x[j] \sin\left(\frac{\pi(j+0.5)(k+1)}{n}\right)$ para $k = 0 \dots n$.
- Tipo 3: $y[k] = (-1)^k x[n - 1] + 2 \sum_{j=0}^{n-2} x[j] \sin\left(\frac{\pi(j+1)(k+0.5)}{n}\right)$ para $k = 0 \dots n$.

Esta transformada es prácticamente equivalente a la parte imaginaria de la transformada de Fourier discreta de una secuencia real e impar de doble longitud. Esta transformada se utiliza principalmente en la compresión de imágenes y para resolver ecuaciones diferenciales parciales.

- `idst(x, type=2, n=None, axis=-1, norm=None, overwrite_x=False)`: función para obtener la inversa de la transformada discreta del seno de una señal x. Al igual que `idct`, `idst` utiliza las relaciones entre los tipos de transformada para obtener la inversa.

7.4.2 Transformada de Fourier de tiempo reducido

La transformada de Fourier de tiempo reducido es una transformada especial de Fourier que se utiliza para determinar los cambios en frecuencia y fase en una señal no periódica. La señal se divide en ventanas y se obtiene el espectro en frecuencia para cada ventana de tiempo, luego se puede observar la evolución del espectro a través del tiempo.

- `stft(x, fs=1.0, window='hann', nperseg=256, noverlap=None, nfft=None, detrend=False, return_onesided=True, boundary='zeros', padded=True, axis=-1)`: función para obtener la transformada de Fourier de tiempo reducido de una señal x. Los parámetros más importantes que hay que tener en cuenta son:
 - Fs: frecuencia de muestreo de la señal x, por defecto es 1Hz.
 - Window: ventana a utilizar para dividir la señal en tramos. Por defecto es Hann, pero se puede utilizar cualquier ventana de esta librería o una propia.
 - Nperseg: ancho de la ventana a utilizar. Por defecto es 256 muestras. Si la longitud de la señal no es múltiple de nperseg, por defecto se añaden 0 al inicio y final de la señal. Esto se puede controlar con los parámetros boundary y padded. Hay que tener en cuenta que si `return_onesided=True`, nperseg es igual a 129.
 - Return_onesided: si es true, los espectros resultantes solo mostrarán las partes positivas (por defecto). Si es False, el resultado contendrá los espectros completos (partes positiva y negativa).
 - Noverlap: determina el número de elementos que se solapan entre los segmentos para cumplir con los requerimientos de COLA (ver `check_COLA`). Por defecto es `nperseg/2`.

Esta función devuelve 3 matrices:

- Un vector de frecuencias que va desde 0 a la mitad del valor de la frecuencia de muestreo (si `one_sided=true`). La longitud es igual a `nperseg`. Si la función es `one_sided`, la longitud del vector de frecuencias será 129 (solo la parte positiva).
- Un vector de tiempos que especifica los intervalos de tiempo en los que se ha dividido la señal. La longitud de este vendrá determinada por la relación entre `Fs` y `nperseg`. Debido a los términos que se repiten para cumplir con COLA, el número de segmentos puede ser mayo que simplemente dividiendo y redondeando al entero superior `Fs` y `nperseg`.
- Una matriz con los valores de la transformada de Fourier para esos segmentos. La matriz tiene dimensiones `NxM` siendo `N` la longitud del vector de frecuencias y `M` la longitud del vector de tiempos de los segmentos.

Por ejemplo, utilizando la señal cuadrada anterior de 4Hz de frecuencia muestreada a 450Hz, pero añadiendo ruido aleatorio (distribuido uniformemente entre -0.2 y 0.2):

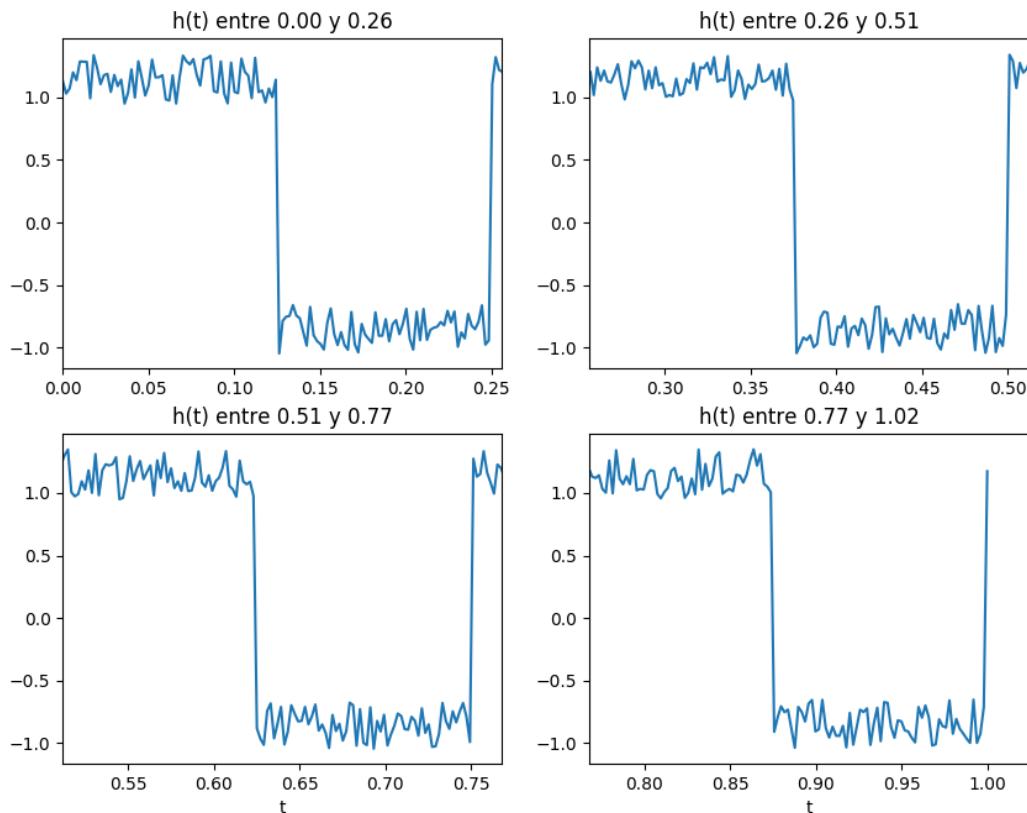
```

Fs = 450
f = 4
T = 1/f
t = linspace(0, 1, 1*Fs)
h = square(2*pi*f*(t+1/4)) + rand(len(t))/2.5-0.2
# Transformada
frec, tiempo, Zxx = stft(h, Fs)
print(len(frec), len(tiempo), shape(Zxx)) #129 5 (129, 5)

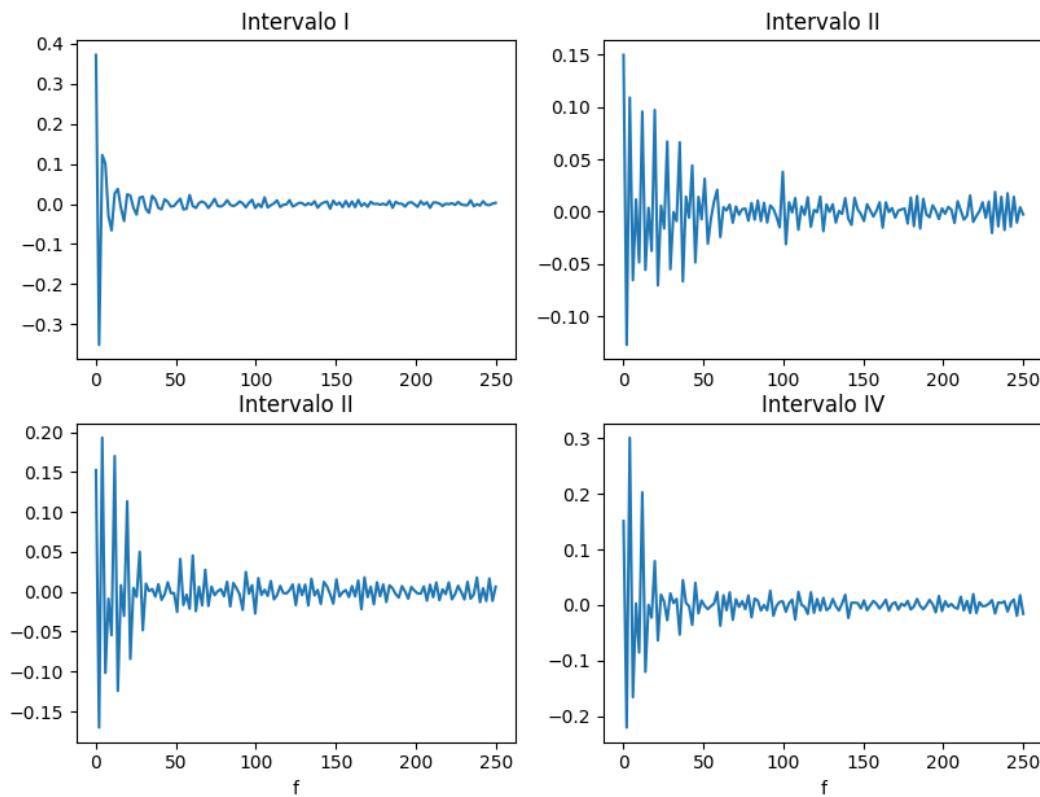
```

Al solo mostrar la parte positiva del espectro, la longitud del vector de frecuencias es 129. Ya que la frecuencia de muestreo es 450 y el vector de frecuencias es de 129, necesitamos 4 intervalos (en el tiempo están los extremos de los intervalos). La matriz de transformadas tiene un total de 129 filas y 5 columnas. Por ejemplo, el valor de la fila 70 - columna 3 es el valor de la transformada para la frecuencia número 70 en el vector de frecuencias, en el tercer intervalo.

Esta es la gráfica de la señal en los cuatro intervalos:



La matriz Z_{xx} que devuelve la función está pensada para ser utilizada para representar frecuencia a través del tiempo, para lo que se necesita un tercer eje (por ejemplo, el color) para representar la magnitud de las frecuencias. Para este ejemplo representaremos los espectros para cada intervalo por separado, para acceder fácilmente a los valores de Z_{xx} basta con hacer la traspuesta. Para los valores del espectro del primer intervalo seria: $Z_{xx}.T[0]$. Obtenemos las siguientes graficas.



Para ver el código fuente del ejemplo → `ejemplo_stft.py`.

- `check_COLA(window, nperseg, nooverlap, tol=1e-10)`: constant overlap add o COLA, es una serie de restricciones para la forma de crear los segmentos de una stft para asegurar que cada punto de la señal está igualmente ponderado y evitar aliasing. Si la Ventana utilizada en la stft cumple con COLA, se podrá recuperar la señal original mediante una istft. Esta función determina si una combinación de tipo de ventana (window), anchura de esta(nperseg) y el número de puntos de superposición entre segmentos, cumple con las restricciones de COLA. En el ejemplo anterior hemos utilizado los valores por defecto de stft (hann, 256, 128), podemos comprobar que COLA se cumple:

```
print(check_COLA(windows.hann(256, sym=False), 256, 128)) # True
```

- `istft(Zxx, fs=1.0, window='hann', nperseg=None, nooverlap=None, nfft=None, input_onesided=True, boundary=True, time_axis=-1, freq_axis=-2)`: función para obtener la transformada inversa de Fourier de tiempo reducido de una matriz de espectros Zxx. Si los parámetros window, nperseg y nooverlap utilizados al hacer la stft cumplen con COLA, el resultado de esta función será una estimación con el método de errores mínimos cuadrados de la señal original.

Para el ejemplo anterior, hemos comprobado que se cumple con COLA por lo que la señal se puede reconstruir. Con `allclose()` comprobamos que todo los valores de la señal original y la reconstruida son iguales con una tolerancia de 1e-05.

```
tiempo_rec, reconstruida = istft(Zxx, Fs)
print(allclose(h, reconstruida[:1*Fs], rtol=1e-05)) # True
```

7.4.3 Transformada de Hilbert

La transformada de Hilbert es una herramienta matemática principalmente utilizada para describir la envolvente compleja de una señal modulada por una portadora real.

- `hilbert(x, _cache=_cache)`: función para la transformada de Hilbert de una señal periódica x. El resultado se obtiene con la convolución de la señal x con $\frac{1}{\pi t}$:

$$\hat{s}(t) = (h * x)(t) = \frac{1}{\pi} \int_{-\infty}^{\infty} \frac{s(\tau)}{t - \tau} d\tau.$$
- `ihilbert(x)`: función para la transformada inversa de Hilbert de una señal periódica x. El resultado se obtiene con la convolución de la señal x con $-\frac{1}{\pi t}$ o lo que es lo mismo: $s(t) = -H(\hat{s}(t))$.

7.5 Señales

El submódulo ***signal*** contiene una serie de funciones para la creación de ventanas, la creación de formas de onda y funciones para el análisis espectral, correlación y convolución de señales.

7.5.1 Ventanas

Conjunto de funciones para la creación de ventanas. En análisis de señales, una ventana es una función matemática que tiene valores dentro del intervalo escogido y es 0 fuera de él. Cuando otra función o secuencia de datos es multiplicado por ella, los valores fuera del intervalo de la ventana serán 0 y el resultado en el intervalo de la ventana será visto modificado por la forma de esta.

Estas ventanas tienen multitud de aplicaciones en el análisis de señales y sistemas. La principal aplicación es en el análisis espectral de señales; permiten la extracción de un intervalo concreto de una señal o secuencia para poder obtener el espectro de ese intervalo. Mas adelante se mostrarán ejemplos del uso de ventanas en la creación de filtros y en el cálculo de transformadas de Fourier de tiempo reducido.

El resultado dependerá del tipo de ventana utilizada por lo que tienen diferentes aplicaciones. Por ejemplo, una ventana rectangular ('boxcar') es útil para analizar señales más cortas que el intervalo de la ventana, a diferencia de la Hamming, que es útil para señales más largas. Otro ejemplo es la ventana Kaiser, la cual es útil para detectar dos señales de casi la misma frecuencia, pero de amplitudes diferentes. En este proyecto están incluidas las siguientes ventanas:

[Boxcar](#), [triang](#), [parzen](#), [bohman](#), [blackman](#), [nuttall](#), [blackmanharris](#), [flattop](#), [bartlett](#), [hanning](#), [barthann](#), [hamming](#), [kaiser](#), [gaussian](#), [general_gaussian](#), [chebwin](#), [cosine](#), [hann](#), [exponential](#) y [tukey](#).

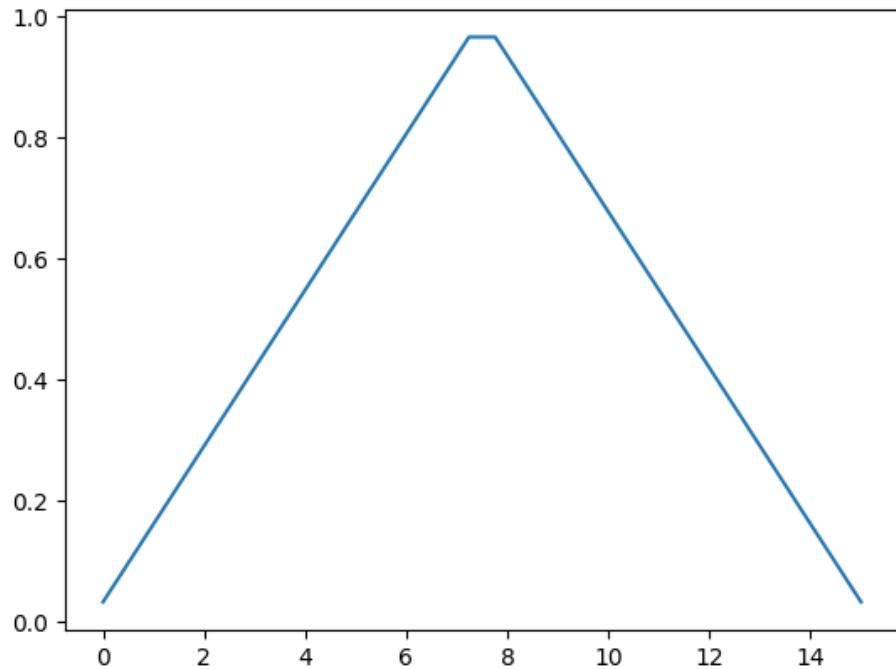
Estas funciones se encuentran dentro del submódulo 'windows', y para utilizarlas hay que seguir esta nomenclatura: **windows.<nombre_ventana>**. También es posible utilizar la siguiente función:

- **get_window**(window, Nx, fftbins=True): siendo **window** una de las ventanas anteriores (algunas de ellas necesitan otros parámetros), **Nx** el número de muestras para crear la ventana y **fftbins** determina si la ventana es periódica (True) para el uso en análisis espectral o simétrica (False) para el uso en diseño de filtros. Esta función devuelve una ventana de tamaño **Nx** y de tipo **window**.
Con el parámetro **fftbins**, se determina si la ventana será periódica (fftbins=True) lo cual permite utilizarla en operaciones con espectros obtenidos con la transformada de Fourier; o simétricas (fftbins=False) para su uso en la creación de filtros. Por defecto, las ventanas creadas son periódicas.

Ejemplos:

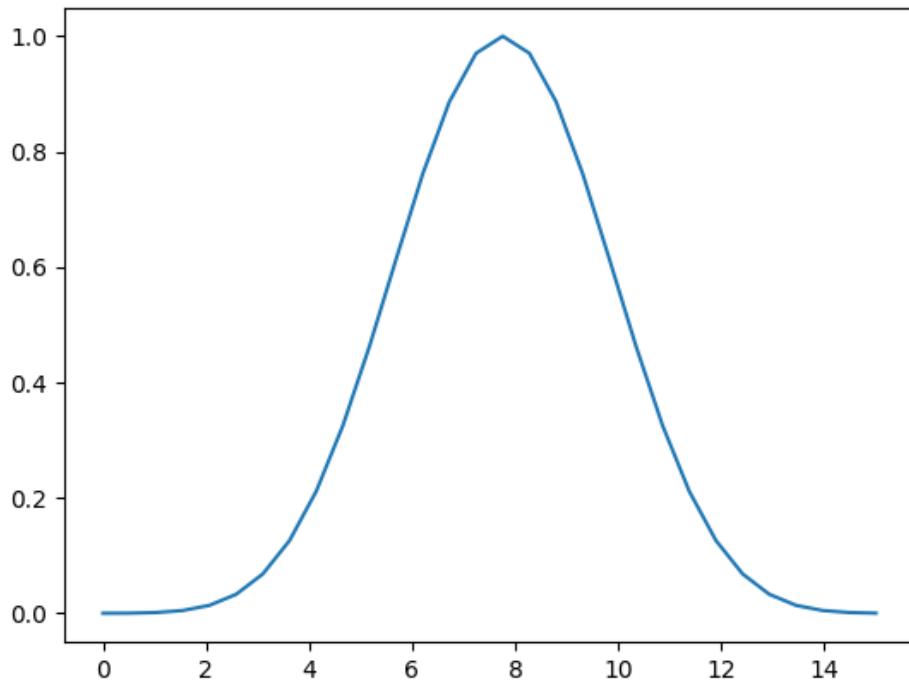
- Ventana triangular simétrica entre 0 y 15.

```
>>> t = linspace(0, 15, 30)
>>> ventana = windows.triang(30)
>>> plot(t, ventana)
[<matplotlib.lines.Line2D object at 0x000001DABA069240>]
>>> show()
```



- Ventana Kaiser periódica con una beta igual a 14, entre 0 y 15 utilizando la función `get_window()`.

```
>>> t = linspace(0, 15, 30)
>>> ventana = get_window('kaiser', 14), 30)
>>> plot(t, ventana)
[<matplotlib.lines.Line2D object at 0x000001DABBD11F98>]
>>> show()
```



7.5.2 Formas de Onda

A la hora de crear un vector que represente una forma de onda, lo primero que hay que hacer siempre es crear un vector de tiempos: para ello basta con utilizar la función `linspace()`. Este vector de tiempos será utilizado por la función para generar otro vector con igual longitud con los valores de la onda. Por defecto, el periodo de estas formas de onda es 2π , pero este se puede modificar multiplicando el vector de tiempos por la frecuencia angular ω (`función(ωt)`). El periodo de estas formas de onda será $T = \frac{2\pi}{\omega}$.

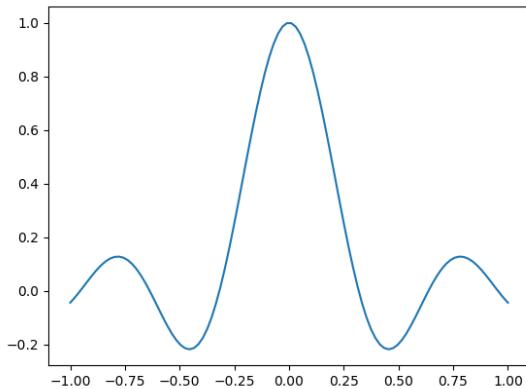
Debido a la forma con la cual Python opera con matrices, cualquiera de las funciones matemáticas se puede utilizar para crear la onda (`sin()`, `cos()`,). Para la creación de otras formas de onda, las siguientes funciones están incluidas en labUGR:

- `sawtooth(t, width=1)`: permite la creación de una señal periódica de dientes de sierra. La señal crece de -1 a 1 entre 0 y el periodo multiplicado por la anchura (`width`). Con valores de anchura inferiores a 1, la señal decrece después de alcanzar el valor de 1. Con `width=0.5` se genera una onda triangular.
- `square(t, duty=0.5)`: permite la creación de una señal rectangular. Con el parámetro `duty`, se establece el ciclo de trabajo de la señal, siendo 0.5 por defecto (cuadrada).
- `gausspulse(t, fc=1000, bw=0.5, retquad=False, retenv=False)`: permite la creación de un pulso Gaussiano siendo `fc` la frecuencia central y `bw` el ancho de banda del pulso (ambos en Hz). Por defecto, la función solo devuelve los valores reales de la señal; si `retquad=True` también devolverá la parte imaginaria de la señal; si `retenv=True` también devolverá la envolvente.
- `unit impulse(shape, idx=None)`: permite la creación de un pulso unitario o función delta. Con `shape` se establece la longitud del vector generado que contendrá un pulso en el elemento con índice `idx`. Por defecto, el pulso se encontrará en el primer elemento (índice 0) pero el índice se puede modificar con `idx='mid'` para un vector con el pulso en el centro).

También se pueden utilizar las funciones vistas en el apartado [7.1](#). Ejemplos:

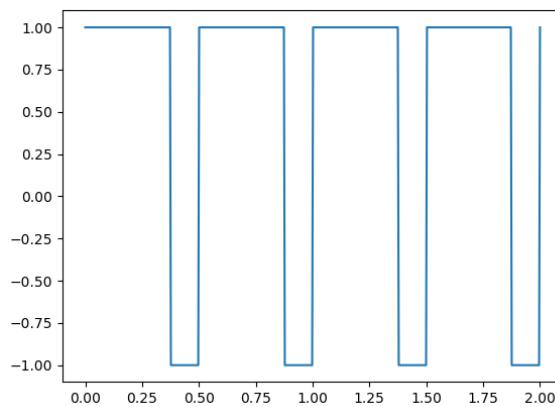
- Función sinc con un periodo de 2 segundos muestreada a 100HZ durante dos segundos.

```
>>> t = linspace(-1, 1, 100)
>>> senal = sinc(2*pi*0.5*t)
>>> plot(t, senal);
>>> show()
```



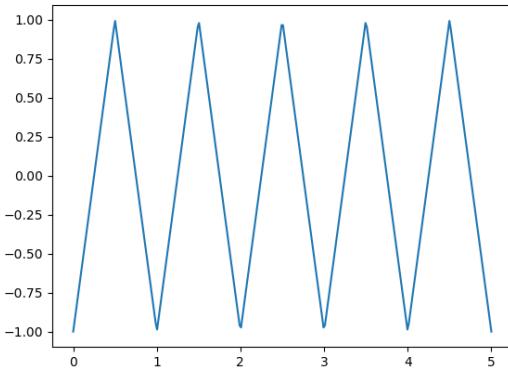
- Señal con una frecuencia de 2Hz muestreada a 250Hz durante 2 segundos con un ciclo de trabajo del 75%. Con el parámetro **duty** podemos asignar la anchura de la señal rectangular.

```
>>> t = linspace(0, 2, 500)
>>> senal = square(2*pi*2*t, duty=0.75)
>>> plot(t, senal);
>>> show()
```



- Señal triangular de 1Hz de frecuencia, muestreada a 60Hz durante 5 segundos.

```
>>> t = linspace(0, 5, 300)
>>> senal = sawtooth(2*pi*1*t, width=0.5)
>>> plot(t, senal);
>>> show()
```



7.5.3 Análisis espectral

Conjunto de funciones para el análisis de las características espectrales de una señal. Hacen uso de las transformadas de Fourier y las ventanas definidas en los apartados anteriores.

- `periodogram(x, fs=1.0, window='boxcar', nfft=None, detrend='constant', return_onesided=True, scaling='density', axis=-1)`: función para la estimación de la densidad espectral de potencia (DEP) utilizando un periodograma. La DEP nos informa de cómo está distribuida la potencia de la señal entre sus diferentes frecuencias. Esta se calcula con la transformada de Fourier de la autocorrelación de la señal:

$$DEP(x) = F\{x(t) * x(-t)\} = X(f) \cdot X^*(f) = |X(f)|^2$$

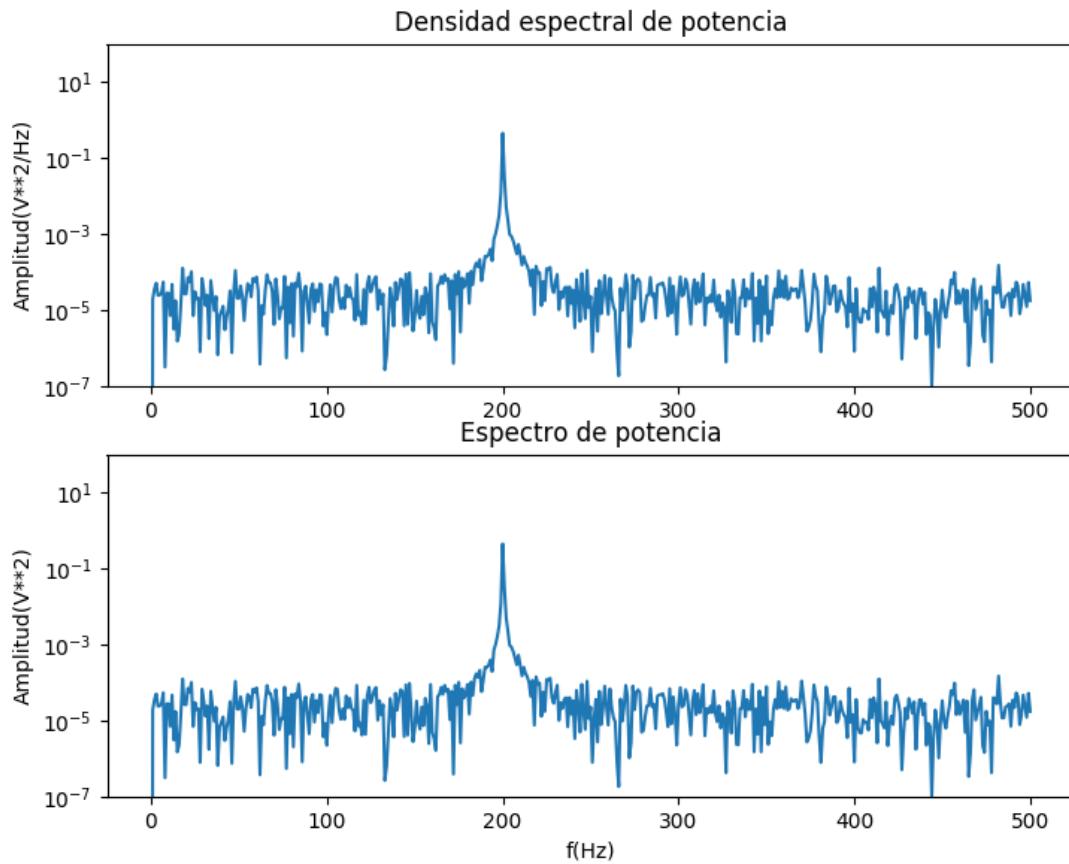
Esta función utiliza el método de Bartlett para hacer los cálculos. Este método se basa en dividir la señal en segmentos que no se solapan y calcula la DEP de cada segmento, haciendo el promedio de estos periodogramas para obtener el resultado final. Fs es la frecuencia de muestreo de la señal, window la ventana específica que se va a utilizar para crear los segmentos y nfft para definir la longitud de la fft.

Esta función también nos permite calcular el espectro de potencia asignando el parámetro `sclaing='spectrum'`. A diferencia del DEP, donde los valores están dados en unidades de potencia partido Hz (W/Hz, mW/Hz, etc.), el espectro de potencia utiliza unidades de potencia (W, mW, etc.). Esta función devuelve un vector de frecuencias y un vector con el DEP o el espectro de potencia de la señal.

Ejemplo: señal sinusoidal de amplitud 1V con una frecuencia igual a 200Hz muestreada a durante 1 segundo, con ruido aleatorio distribuido uniformemente entre -0.2 y 0.2.

```
Fs = 1000
f = 200
T = 1/f
t = linspace(0, 1, 1*Fs)
h = sin(2*pi*f*t) + rand(len(t))/2.5-0.2
# Calculo de la densidad espectral de potencia
freccs, DEP = periodogram(h, Fs)
# Calculo del espectro de potencia
freccs2, EP = periodogram(h, Fs, scaling='spectrum')
```

Para grafica la DEP y el EP utilizamos la función `semilogy()` que nos permite representar la eje de y en una escala logarítmica y el eje de x en una escala lineal.



En la gráfica podemos ver el pico de potencia de la señal en 200Hz y el resto representa la potencia añadida por el ruido aleatorio. Para ver el código fuente → [ejemplo_periodograma.py](#). En [ejemplo_audio.py](#) hay otro ejemplo de la utilización de esta función.

- `spectrogram(x, fs=1.0, window=('tukey', .25), nperseg=None, noverlap=None, nfft=None, detrend='constant', return_onesided=True, scaling='density', axis=-1, mode='psd')`: función para obtener el spectrograma de una señal x. Este nos permite visualizar el cambio de los componentes en frecuencia de una señal no estacionaria a través del tiempo. Similar a la transformada de Fourier de tiempo reducido (STFT), esta función divide la señal en segmentos y aplica los cálculos en cada segmento. Tiene un total de 5 modos diferentes (se especifica utilizando el parámetro mode):
 - psd: corresponde a la densidad espectral de potencia y utiliza el periodograma para calcular la DEP de cada segmento de tiempo.
 - complex: devuelve los valores complejos del espectro en frecuencias a través del tiempo obtenidos de una STFT.
 - magnitude: devuelve los valores absolutos de la magnitud de una STFT.
 - angle: devuelve el ángulo complejo de una STFT.
 - phase: devuelve el ángulo complejo de una STFT en el rango de $180^\circ \leq x < 180^\circ$.

El resultado de esta función sigue la misma estructura que el resultado de stft. Devuelve un vector con las frecuencias, un vector con los extremos de los segmentos de tiempos y una matriz NxM con los resultados para cada segmento. En [ejemplo_audio.py](#) se encuentra un ejemplo de la utilización de esta función para obtener el espectro de una señal no periódica.

- [`coherence`](#)(x, y, fs=1.0, window='hann', nperseg=None, nooverlap=None, nfft=None, detrend='constant', axis=-1): función para calcular la estimación de la magnitud de coherencia espectral entre dos señales discretas x e y. La coherencia nos permite examinar la relación y semejanza entre los espectros de las dos señales. Se suele utilizar para el cálculo de la transferencia de potencia entre la entrada y salida de un sistema lineal. La coherencia se define como:

$$C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}$$

Siendo P_{xx} y P_{yy} las estimaciones de densidad espectral de x e y respectivamente y P_{xy} la estimación de la densidad espectral de potencia cruzada de x e y. Esta función utiliza el método Welch para hacer los cálculos, muy parecido al método Bartlett visto anteriormente. Con Welch se establecen un número de puntos que se solapan en cada segmento (nooverlap). Esto permite incrementar la exactitud de las densidades. Este número es por defecto nperseg(longitud de cada segmento)/2 pero si se utilizan ventanas más estrechas se debe utilizar un número mayor.

- [`csd`](#)(x, y, fs=1.0, window='hann', nperseg=None, nooverlap=None, nfft=None, detrend='constant', return_onesided=True, scaling='density', axis=-1): función para estimar la densidad espectral de potencia cruzada entre dos señales x e y utilizando el método Welch. El resultado se obtiene conjugado el resultado de la transformada de Fourier de x por la transformada de Fourier de y. $csd = [TF\{x(t)\} \cdot TF\{y(t)\}]^*$. Devuelve un vector de frecuencias y un vector con la densidad espectral cruzada o el espectro de potencia cruzada dependiendo del valor de scaling (density para el primero y spectrum para el segundo)

7.5.4 Convolución - correlación

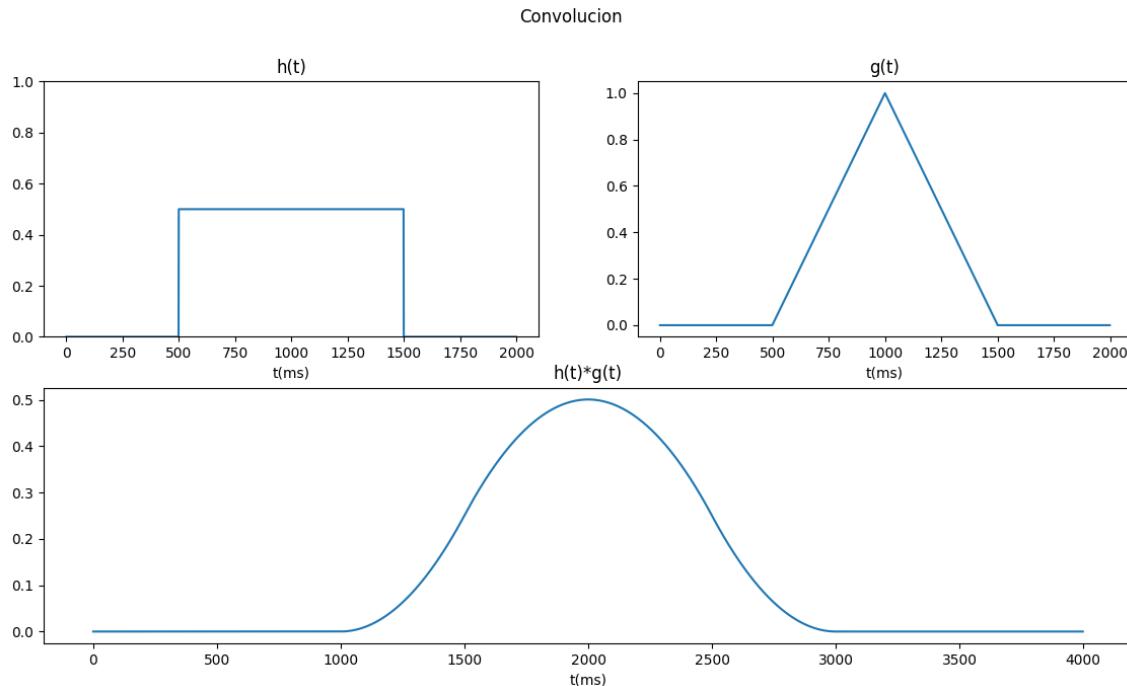
La convolución y la correlación son funciones matemáticas muy utilizadas en el análisis de señales. La correlación permite medir la similitud entre dos señales y la convolución permite obtener la salida de un sistema a partir de la entrada y la respuesta al impulso de este.

- [`convolve`](#)(in1, in2, method='full', method='auto'): función para la convolución entre dos señales descritas por los vectores **in1** e **in2** los cuales deben tener la misma longitud. Existen dos formas de calcular la convolución de dos señales: el método directo (**method='direct'**) mediante la ecuación $y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot h[n-k]$ o utilizando la transformada de Fourier (**method='fft'**) $y[n] = x[n] * h[n] = F^{-1}\{X[\omega] \cdot H[\omega]\}$ utilizando las propiedades de la convolución. Si el método no se especifica, **convolve** utiliza una función interna (**choose_conv_method**) para escoger el método que será más rápido.

Esta función devuelve un vector con los valores de la convolución con un tamaño igual a la suma de los tamaños de **in1** e **in2**. Si se especifica **mode='same'** el resultado tendrá el mismo tamaño que **in1**. El resultado está escalado por **in2** por lo que hay que dividirlo por la sumatoria de esta. En [ejemplo_convolve.py](#) se puede encontrar un ejemplo de la convolución de una señal cuadrada con una triangular:

```
convolucion = convolve(cuadrada, triangular)/sum(cuadrada)
```

El resultado es el siguiente:



- [correlate](#)(in1, in2, mode='full', method='auto'): función para la correlación entre dos señales descritas por los vectores **in1** e **in2** los cuales deben tener la misma longitud. Correlate tiene los mismos parámetros y un resultado con la misma estructura que convolve.

7.6 Sistemas

El submódulo `systems` incluye un conjunto de funciones para el diseño, simulación y análisis de sistemas LTI (sistemas lineales e invariantes en el tiempo) tanto discretos como continuos.

Este submódulo soporta 3 tipos de representación de sistemas LTI:

- **Función de transferencia**: siendo $X(s)$ la entrada de un sistema LTI y $Y(s)$ la salida en el dominio de

$$G(s) = \frac{Y(s)}{X(s)} = \frac{\sum_{k=0}^M b_k s^k}{\sum_{k=0}^N a_k s^k}$$

Laplace, la función de transferencia se obtiene dividiendo ambos elementos. La clase que define este objeto es la siguiente: `TransferFunction(num, den, dt=0)`. Este se define con dos vectores, el primero con los elementos del numerador y el segundo con los elementos del denominador. Por ejemplo: $H(s) = \frac{s^2 + 3s + 3}{s^2 + 2s + 1}$ Se presentaría

de la siguiente manera:

```
>>> num = [1, 3, 3]
>>> den = [1, 2, 1]
>>> TransferFunction(num, den)
TransferFunctionContinuous(
    array([ 1.,  3.,  3.]),
    array([ 1.,  2.,  1.]),
    dt: None
)
```

- **Ceros, polos y ganancia**: la representación de un sistema LTi en ceros, polos ganancia se obtiene

$$H(s) = k \frac{\prod_{i=1}^m (s - z_i)}{\prod_{i=1}^n (s - p_i)}$$

a partir de la función de transferencia. Z_i son los ceros de la función, p_i los polos y k la ganancia. La clase que define este objeto es la siguiente: `ZerosPolesGain(z, p, k, dt=0)`. Los ceros y los polos deben ser vectores y la ganancia un número real (float). Por ejemplo: $H(s) = 5 \frac{(s-1)(s-2)}{(s-3)(s-4)}$ Se presentaría de la siguiente manera:

```
>>> ceros = [1, 2]
>>> polos = [3, 4]
>>> ganancia = 5
>>> ZerosPolesGain(ceros, polos, ganancia)
ZerosPolesGainContinuous(
    array([1, 2]),
    array([3, 4]),
    5,
    dt: None
)
```

- **Variables de estado**: describe un sistema LTI a partir de un conjunto de entradas, salidas y

$v'(t) = \mathbf{A} v(t) + \mathbf{b} x(t)$ variables de estado relacionadas por ecuaciones diferenciales de primer orden que se combinan en una ecuación diferencial matricial de primer orden. La clase que define este objeto es la siguiente:

`StateSpace(a, b, c, d, dt=0)`. \mathbf{A} tiene que ser una matriz cuadrada N x N, \mathbf{b} un vector columna N x 1, \mathbf{c} un vector fila 1 x N y d un número real. Por ejemplo:

```

>>> a = array([[0, 1], [0, 0]])
>>> b = array([[0], [1]])
>>> c = [1, 0]
>>> d = 0
>>> StateSpace(a, b, c, d)
StateSpaceContinuous(
    array([[0, 1],
           [0, 0]]),
    array([[0],
           [1]]),
    array([[1, 0]]),
    array([[0]]),
    dt: None
)

```

Todos los ejemplos anteriores crean sistemas LTI continuos en el tiempo. Para crear sistemas discretos basta con especificar ***dt***, el periodo de muestreo. Por ejemplo: `ZerosPolesGain(ceros, polos, ganancia, dt=0.1)`.

7.6.1 Conversión de sistemas LTI

- `tf2ss(num, den)`: conversión de un sistema desde la función de transferencia a la representación en variables de estado.
- `ss2tf(A, B, C, D, input=0)`: conversión de un sistema desde las variables de estado a la función de transferencia.
- `zpk2ss(z, p, k)`: conversión de un sistema desde los ceros/polos/ganancia a la representación en variables de estado.
- `ss2zpk(A, B, C, D, input=0)`: conversión de un sistema desde las variables de estado a ceros polos ganancia.
- `cont2discrete(system, dt, method="zoh", alpha=None)`: transforma un sistema continuo a discreto con un periodo de muestreo igual a ***dt***. Con **method** podemos especificar el método que se utilizará para discretizar el sistema.
- `normalize(b, a)`: función utilizada para normalizar los coeficientes del numerador **b** y del denominador de un sistema.

Para la conversión entre los diferentes tipos de representación también se pueden utilizar los métodos de la clase Sistema. Estos métodos son: `.to_ss()`, `.to_tf()` y `.to_zpk()`. Por ejemplo, el sistema $H(s) = \frac{s^2 + 3s + 3}{s^2 + 2s + 1}$ convertido a ceros/polos/ganancia:

```

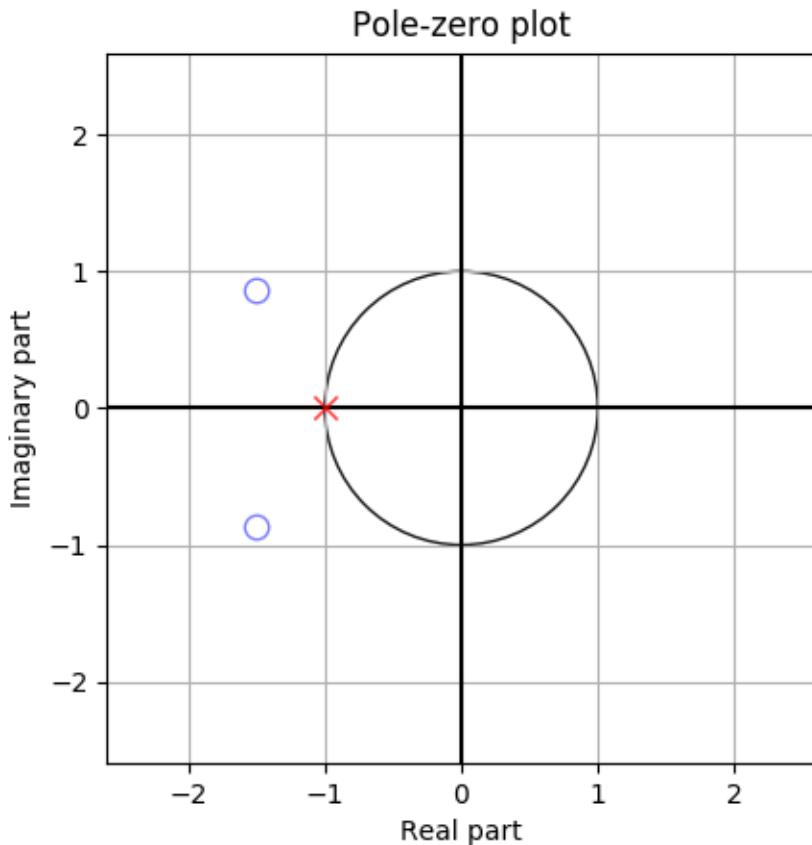
>>> num = [1, 3, 3]
>>> den = [1, 2, 1]
>>> TransferFunction(num, den).to_zpk()
ZerosPolesGainContinuous(
    array([-1.5+0.8660254j, -1.5-0.8660254j]),
    array([-1., -1.]),
    1.0,
    dt: None
)

```

7.6.2 Trazado en el plano z

- `zplane(system, ax=None)`: traza la representación del sistema '**system**' en el plano complejo de z . El Sistema puede ser dado en su representación por ceros/polos/ganancia, la función de transferencia o por sus variables de estado. Con `ax` podemos especificar un gráfico de Matplotlib donde representar el sistema. **Creada por mí**. Por ejemplo, para representar el sistema anterior bastaría con hacer:

```
zplane(TransferFunction(num, den))
```



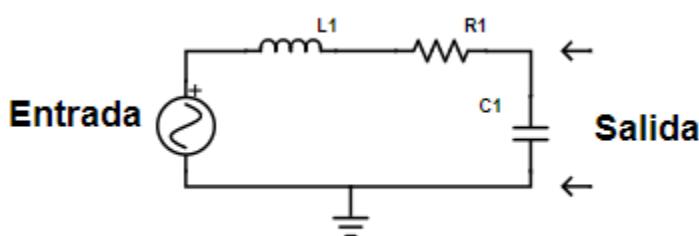
7.6.3 Sistemas LTI continuos

Tras haber utilizado una de las funciones anteriores para crear un sistema LTI continuo, podemos utilizar las siguientes funciones:

- `impulse(system, X0=None, T=None, N=None)`: respuesta al impulso de un sistema. Con **X0** podemos especificar el estado inicial (por defecto 0), con **T** un vector de tiempos o también con **N** podemos especificar el número de puntos a calcular (si no son especificados, la función los calcula automáticamente). Devuelve un vector de tiempos y otro vector con la respuesta del sistema.
- `step(system, X0=None, T=None, N=None)`: respuesta al escalón de un sistema. Tiene los mismos parámetros que `impulse()`. La respuesta sigue la misma estructura que la de `impulse()`.

- **`freqresp`**(`system`, `w=None`, `n=10000`): respuesta en frecuencia de un sistema. Con `w` podemos especificar un vector de frecuencias o utilizar `n` para determinar el número de puntos en frecuencia a calcular. Si estos valores no son incluidos, la función determinará un rango de frecuencias apropiado que incluya ceros y polos. Devuelve un vector de frecuencias (rad/s) y un vector con las magnitudes (son valores complejos con parte real y parte imaginaria).
- **`bode`**(`system`, `w=None`, `n=100`): respuesta en frecuencia de un sistema basada en el diagrama de bode. Esta función es análoga a **`frqresp()`**, pero devuelve 3 vectores en vez de 2: un vector de frecuencias (rad/s), un vector con los valores de la magnitud (dB) y un vector con los valores de la fase (decimal).
- **`lsim`**(`system`, `U`, `T`, `X0=None`, `interp=True`): función para simular la respuesta de un sistema a una entrada descrita por un vector de tiempos `T` y un vector de valores `U`. Con `X0` podemos especificar el estado inicial (por defecto 0) y con `interp`, el tipo de interpolación a utilizar para la entrada (linear o zero-order-hold). Devuelve un vector de tiempos (igual al parámetro `T`), un vector con los valores de la respuesta del sistema y un vector con la evolución en el tiempo del vector estado.

Por ejemplo, si tenemos el siguiente circuito RLC en series con unos valores de $L=0.1H$, $C=200\mu F$ y $R=50\Omega$ como muestra la siguiente figura:



Sabemos que la función de transferencia de este tipo de sistemas es la siguiente:

$$H(s) = \frac{1}{LC} \frac{1}{s^2 + \frac{R}{L}s + \frac{1}{LC}}$$

Con el siguiente código creamos el sistema en Python:

```
# Parametros del circuito
L = 0.1
C = 200e-6
R = 50

# Frecuencia de corte
w0 = 1/sqrt(L*C)

# Sistema definido por la funcion de transferencia
num = 1/(L*C)
den = [1, R/L, 1/(L*C)]
H = TransferFunction(num , den)
```

También sabemos que este circuito se comporta como un filtro paso baja de segundo orden con una frecuencia de corte $\omega_0 = \frac{1}{\sqrt{LC}} = 223.61 rad/s$, una ganancia de 1 y dos polos en $\omega = 361.8 rad/s$ y en $\omega = 138.2 rad/s$ en este caso reales. Los polos se pueden obtener con:

```
abs(H.poles) # [ 361.80339887  138.19660113]
```

A continuación utilizamos las funciones `impulse()` para la respuesta impulsiva, `step()` para la respuesta al escalón, `freqresp()` para la respuesta en frecuencia y `bode()` para el diagrama de bode:

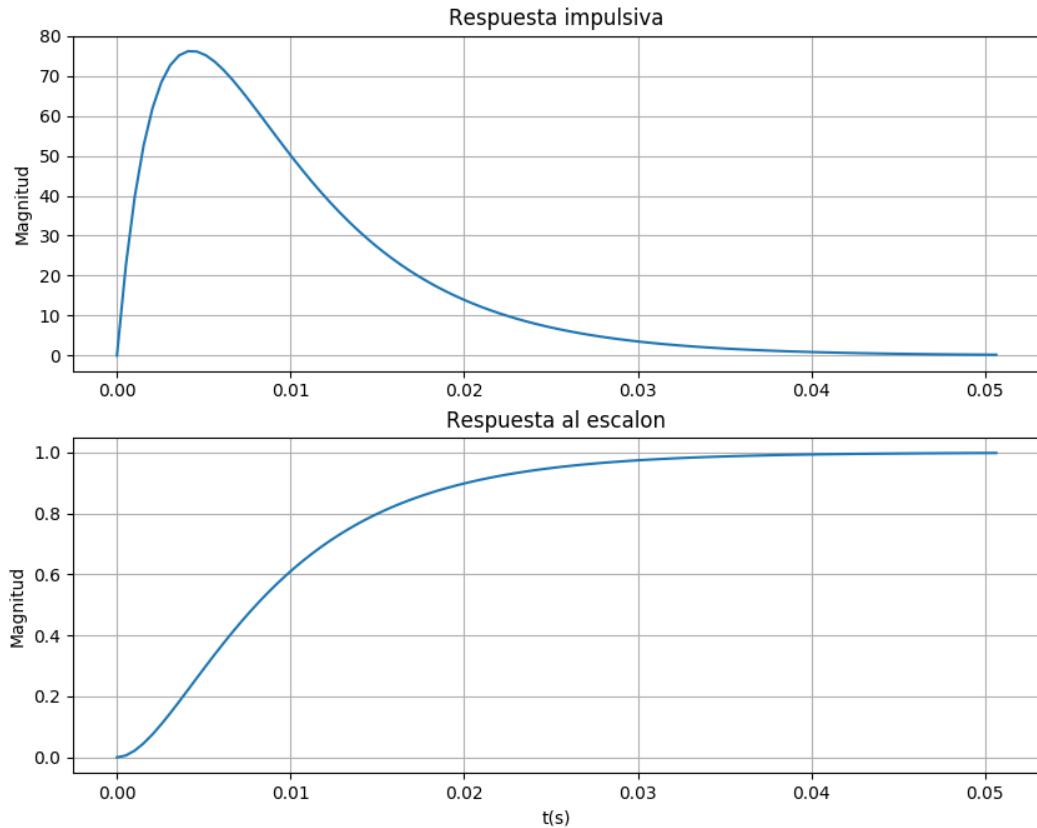
```
# Respuesta impulsiva
t_imp, y_imp = impulse(H)

# Respuesta al escalon
t_esc, y_esc = step(H)

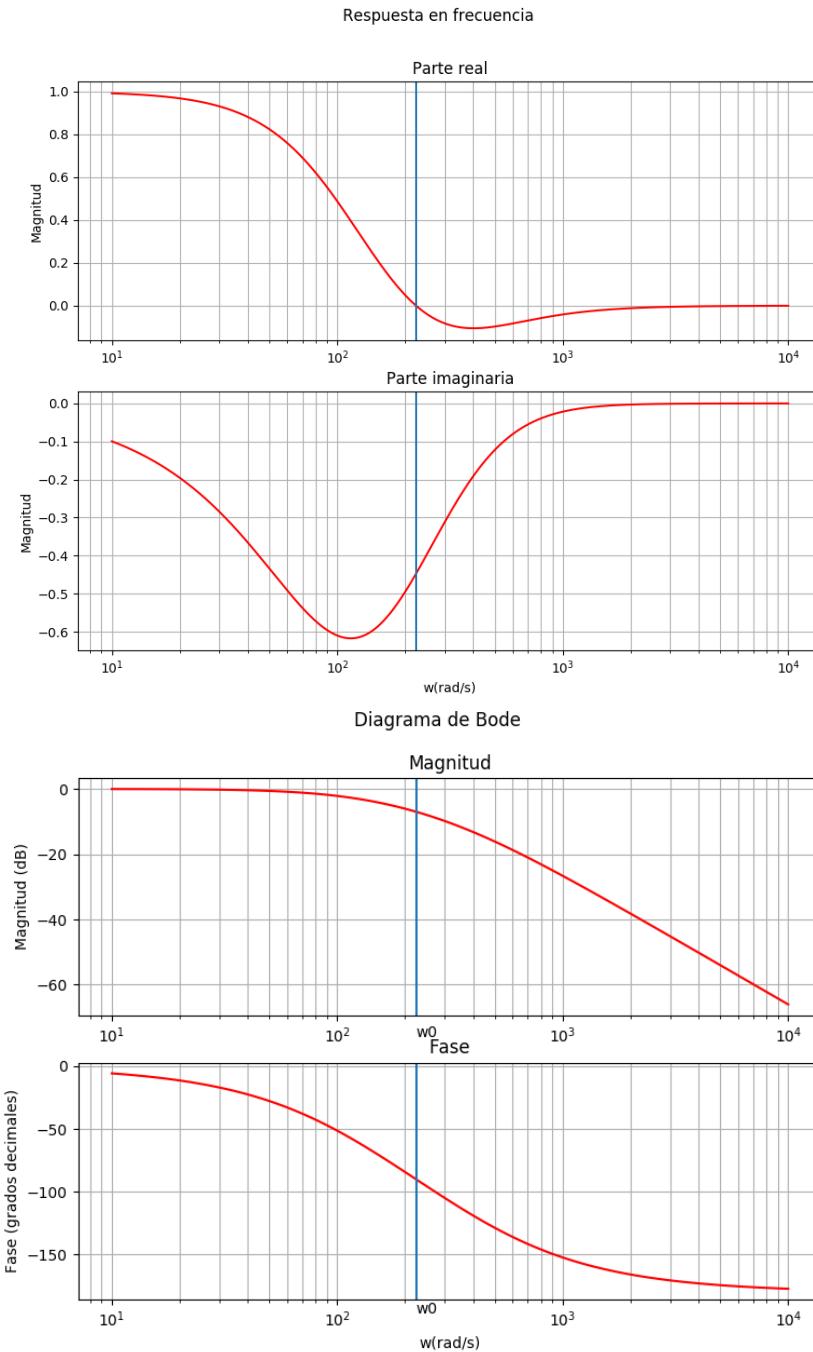
# Representacion en frecuencia
w_freqs, magn_freqs = freqresp(H)

# Diagrama de Bode
w_bode, magn_bode, fase_bode = bode(H)
```

Estas son las gráficas que obtenemos como resultado:



En el caso de la respuesta en frecuencia y del diagrama de Bode, es útil representar el eje de las frecuencias en una escala logarítmica para que el resultado sea fácil de interpretar. Para ello utilizamos la función `semilogx()`. También es útil marcar donde se encuentra la frecuencia de corete del sistema.

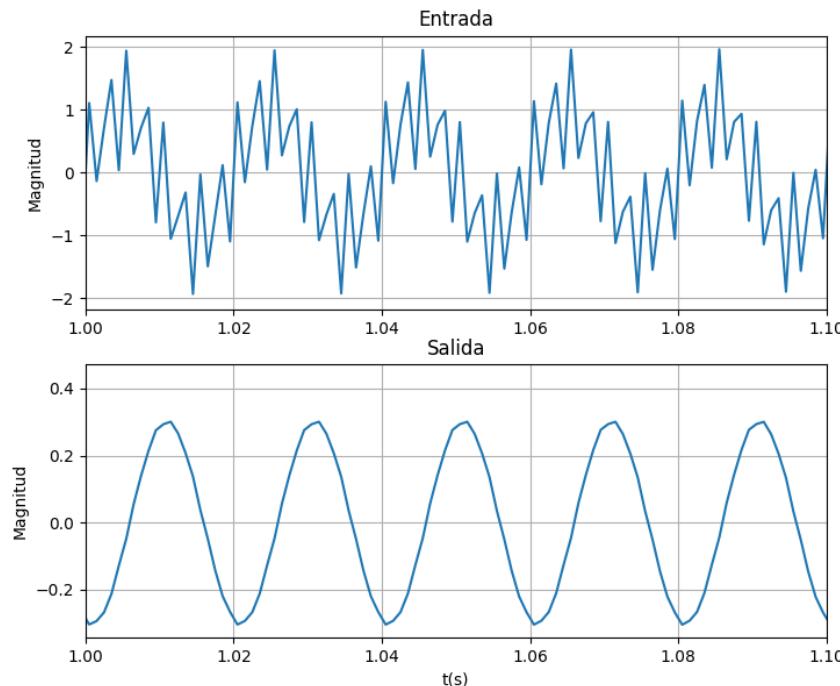


En el diagrama de bode podemos ver claramente que, en efecto, el circuito se comporta como un filtro paso bajo de segundo orden con una de -40dB/dec en frecuencias mayores de la frecuencia de corte. En realidad, la ganancia es de -20dB/dec entre el primer polo y el segundo y de -40dB/dec a partir de ahí.

A continuación también podemos simular el sistema utilizando la función `lsim()`. Para ello creamos una señal formada por dos componentes sinusoidales con frecuencias 50Hz y 400 Hz muestreados a 2kHz.

```
# Senal de entrada
t = linspace(0, 2, 2000)
f1 = 50
f2 = 400
x = sin(2*pi*f1*t) + sin(2*pi*f2*t)
# Simulacion del sistema
t_sim, y_sim, ev = lsim(H, x, t)
```

Simulación del sistema



Como podemos comprobar en la gráfica, el sistema filtra los componentes en frecuencia de la señal de entrada superiores a la frecuencia de corte. Para ver el código fuente del ejemplo -> [ejemplo_LTIcont.py](#).

7.6.4 Sistemas LTI discretos

Tras haber utilizado una de las funciones anteriores para crear un sistema LTI discreto (especificando el periodo de muestreo), podemos utilizar las siguientes funciones análogas a las funciones para los sistemas continuos:

- [`dimpulse`](#)(system, x0=None, t=None, n=None).
- [`dstep`](#)(system, x0=None, t=None, n=None).
- [`dfreqresp`](#)(system, w=None, n=10000, whole=False). Si **w** no está especificada, el resultado se calculará utilizando las frecuencias de 0 a la frecuencia de Nyquist. Con **whole=True** podemos especificar que se calculen utilizando todas las frecuencias de 0 a 2π .
- [`dbode`](#)(system, w=None, n=100).
- [`dlsim`](#)(system, u, t=None, x0=None).

Hay que tener en cuenta que los valores de los vectores de frecuencias que devuelven las funciones dbode y dfreqresp están normalizadas (rad/muestra).

7.7 Filtros

El submódulo ***filters*** incluye todas las funciones necesarias para el diseño de filtros, tanto analógicos como digitales, FIR e IIR, así como su análisis espectral y funciones para filtrar señales usando los filtros creados.

7.7.1 Diseño de filtros

7.7.1.1 Filtros de Respuesta Finita al Impulso (FIR)

Conjunto de funciones para la creación de filtros digitales de respuesta finita al impulso. La respuesta de los filtros FIR a una entrada finita no se mantiene de manera indefinida y llega a 0 en una cantidad de tiempo finita. A diferencia de los filtros IIR, en la función de transferencia de un filtro FIR solo hay coeficientes en el numerador (ceros) pero ningún polo, lo que los hace estables y de fase lineal. Dependiendo del número de coeficientes y de la simetría de la respuesta impulsiva, existen 4 tipos de filtros FIR:

- Tipo I: número de coeficientes impar y respuesta simétrica.
- Tipo II: número de coeficientes par y respuesta simétrica.
- Tipo III: número de coeficientes impar y respuesta antisimétrica.
- Tipo IV: número de coeficientes par y respuesta antisimétrica.

La precisión de estos filtros depende principalmente del número de coeficientes, aumentando a medida que este número se hace mayor, aunque esto provoca que sea necesario un mayor poder de computación para utilizarlo. Hay tres métodos diferentes para la creación de los filtros, diseño por ventanas, diseño mediante el muestreo en frecuencia y el diseño de rizado óptimo/constante:

- **`firwin`**(*numtaps*, *cutoff*, *window='hamming'*, *pass_zero=True*, *nyq=1.0*, *width=None*): función para el diseño de un filtro FIR mediante ventanas. Dado el número de coeficientes con ***numtaps***, esta genera los coeficientes para generar un filtro óptimo de orden ***numtaps-1*** (número de ceros) y de tipo I si ***numtaps*** es par o de tipo II si ***numtaps*** es impar. Para conseguir que el filtro tenga una respuesta estable y sea lineal, se multiplica por la ventana establecida en ***window*** para acotar la respuesta y solo entonces pasa a ser un filtro FIR. Dependiendo de la combinación de los parámetros ***cutoff*** (frecuencias de corte) y ***pass_zero*** se generan los diferentes tipos de filtros:
 - Paso-baja: la frecuencia de corte es un único valor y *pass_zero=True*. Por ejemplo: `firwin(coeficientes, fc)`
 - Paso-alta: la frecuencia de corte es un único valor y *pass_zero=False*. Por ejemplo: `firwin(coeficientes, fc, pass_zero=False)`
 - Paso-banda: ***cutoff*** es un vector con los límites en frecuencia de la banda ($[f_1, f_2]$) y *pass_zero=True*. Por ejemplo: `firwin(coeficientes, [fc1, fc2], pass_zero=True)`

- **Elimina-banda:** *cutoff* es un vector con los límites en frecuencia de la banda ($[f_1, f_2]$) y *pass_zero=False*. Por ejemplo: `firwin(coeficientes, [fc1, fc2])`

Con el parámetro *nyq*, se establece la frecuencia de Nyquist, la cual es la mitad de la frecuencia de muestreo. Las unidades del parámetro *cutoff* son las mismas que las de *nyq* y deben de ser menores en magnitud. Esto quiere decir que, si *nyq* no se especifica, los valores de las frecuencias de corte deberán de estar comprendidos entre 0 y 1 y para obtener el valor real de las frecuencias se deberán multiplicar por $fs/2$.

Por ejemplo, hay dos formas de crear un filtro elimina banda de orden 3 con una banda de paso entre 100Hz y 200Hz y una frecuencia de muestreo de 800Hz:

```
>>> firwin(3, [0.25, 0.5])
array([-0.01014642,  1.02029285, -0.01014642])
>>> firwin(3, [100, 200], nyq=400)
array([-0.01014642,  1.02029285, -0.01014642])
```

El resultado de esta función es un vector con los coeficientes del numerador del filtro de longitud igual a *numtaps*.

- **`firwin2`**(*numtaps*, *freq*, *gain*, *window='hamming'*, *nyq=1.0*, *antisymmetric=False*): función para el diseño de un filtro FIR mediante el método de muestreo en frecuencias. Dados un vector de frecuencias *freq* y un vector de ganancias *gain*, que representan la respuesta en frecuencia del filtro deseado, esta función genera un filtro FIR de orden *numtaps*-1. Para ello utiliza la transformada inversa de Fourier sobre el espectro de frecuencias para obtener la respuesta impulsiva del filtro en el eje de tiempos. Para acotar la respuesta impulsiva del filtro, la función utiliza la ventana especificada en el parámetro *window*.

Al igual que con en el parámetro *cutoff* de `firwin()`, las frecuencias en el vector *freq* están normalizadas, escaladas por el valor de la frecuencia de Nyquist. Si *nyq* no se especifica, los valores de *freq* deben estar comprendidos entre 0 y 1; si está especificado, las frecuencias deberán estar comprendidas entre 0 y *nyq*.

Con las diferentes combinaciones de *numtaps*(par o impar) y *antisymmetric*(True para una respuesta simétrica o False para una respuesta antisimétrica) se pueden generar los 4 tipos de filtros FIR. El tipo de filtro impone restricciones en el vector de ganancias: para tipo II, el último elemento debe ser 0; para tipo III, el primer y último elemento deben ser 0; para tipo IV, el primer elemento debe ser 0.

La función devuelve un vector con los coeficientes del numerador del filtro FIR con una longitud igual a *numtaps*. El filtro resultante tendrá una respuesta en frecuencia aproximada a la establecida en los parámetros y esta aproximación será más precisa con un mayor número de coeficientes.

Por ejemplo, para crear un filtro paso alta de orden 4, muestreado a 800 Hz y con una frecuencia de corte de 200Hz utilizamos el siguiente código:

```
>>> freqs = [0, 200, 400]
>>> ganancia = [0, 1, 1]
>>> firwin2(5, freqs, ganancia, nyq=400)
```

```
array([-0.00853553, -0.11084404,  0.75        , -0.11084404, -0.00853553])
```

La ganancia de la respuesta en frecuencia será aproximadamente 1 de 200 a 400 Hz y decrecerá linealmente de 200 a 0Hz. En este ejemplo se ha utilizado un espectro muy simple para generar el filtro, pero la principal ventaja de este método es que se pueden crear filtros muy complejos simplemente diseñado la respuesta en frecuencia deseada.

- **firls**(numtaps, bands, desired, nyq=1.): función para el diseño mediante el método del rizado óptimo/constante. Utiliza el algoritmo de optimización matemática de los mínimos cuadrados para obtener el filtro cuya respuesta en frecuencia se aproxima mejor a la respuesta deseada. De manera similar a firwin2(), el espectro deseado se define con los vectores **bands**, para establecer las bandas de frecuencia, y **desired**, para establecer la ganancia en esas bandas de frecuencia. El mayor inconveniente de esta función es que con ella solo se pueden crear filtros lineales de tipo I, de respuesta simétrica y numero de coeficientes impar. Por lo tanto, el valor de **numtaps** debe ser impar. Al igual que en las funciones anteriores, las frecuencias de las bandas están normalizadas por la frecuencia de Nyquist, **nyq**.

Por ejemplo, un filtro paso banda de orden 11, que deja pasar a los componentes en la banda de frecuencia de 200 a 300Hz, con una frecuencia de muestreo de 1000Hz:

```
>>> bandas = [0, 100, 200, 300, 400, 500]
>>> magnitud_deseada = [0, 0, 1, 1, 0, 0]
>>> firls(11, bandas, magnitud_deseada, nyq=500)
array([-7.32279872e-18,   8.57974597e-02,   5.69458288e-17,
       -2.66704097e-01,  -1.53857412e-16,   3.66089094e-01,
      -1.53857412e-16,  -2.66704097e-01,   5.69458288e-17,
       8.57974597e-02,  -7.32279872e-18])
```

Para obtener un filtro más preciso, la respuesta deseada debe ser más precisa y contener más elementos. Por ejemplo, creando un vector de frecuencias de 200 elementos con su vector de ganancias correspondiente:

```
>>> bandas = linspace(0, 500, 200)
>>> ganancia = ones(len(bandas))
>>> ganancia[bandas<200] = 0
>>> ganancia[bandas>300] = 0
```

7.7.1.2 Filtros de Respuesta infinita al Impulso (IIR)

Conjunto de funciones para la creación de filtros de respuesta infinita al impulso. La respuesta de los filtros IIR a una entrada finita se compondrá por un número infinito de términos no nulos y nunca convergerá a 0. Estos filtros están definidos por un numero de ceros y polos que determinaran su respuesta. A diferencia de los filtros FIR, los filtros IIR pueden ser análogos o digitales y, aunque su respuesta puede ser inestable, estos filtros permiten generar sistemas con una respuesta similar a la de filtro FIR del mismo tipo, pero utilizando un número de coeficientes menor, lo que reduce el poder de computación necesario para utilizarlos.

La fase de estos filtros no se puede controlar y suele ser no lineal. Dependiendo de la distribución de los ceros y los polos en la circunferencia unidad se pueden dar 3 tipos de filtros digitales (discretos):

- Fase mínima: todos los ceros y polos se encuentran dentro de la circunferencia unidad. Este filtro será estable y causal.
- Fase máxima: todos los ceros se encuentran en el exterior de la circunferencia unidad y los polos dentro. Este filtro será estable pero no causal.
- Inestable: cuando algún polo esté fuera de la circunferencia unidad, el filtro será inestable

En el caso de filtros analógicos (continuos), estos pueden ser estables, aunque haya polos fuera de la circunferencia unidad, siempre y cuando estos se encuentren en la mitad izquierda del plano Z.

El diseñador del filtro es responsable de elegir los parámetros adecuados para que el filtro obtenido sea estable. Dependiendo de la aplicación del filtro, será necesario un filtro de fase máxima o fase mínima. Estos filtros pueden ser analógicos o digitales, aunque estos últimos están derivados de los analógicos. Hay tres filtros IIR incluidos en esta librería:

- **butter**(N, Wn, btype='low', analog=False, output='ba'): función para el diseño de un filtro IIR de Butterworth digital o analógico(**analog=True**). Estos filtros son utilizados para producir una respuesta con ganancia constante hasta la frecuencia de corte y luego disminuye a -20N dB/dec siendo **N** el orden del filtro. El parámetro **btype**, especifica el tipo de filtro a diseñar: '**lowpass**' para paso baja, '**highpass**' para paso alta, '**bandpass**' para paso banda o '**bandstop**' para elimina banda.

El parámetro **Wn** debe ser un vector con las frecuencias críticas del filtro. Una frecuencia crítica es el punto en el espectro en el que la ganancia es $1/\sqrt{2}$ la ganancia en la banda de paso (-3dB). En el caso de filtros analógicos, las unidades del vector serán rad/s. En los filtros digitales, los valores de **Wn** están normalizados por la frecuencia de Nyquist (fs/2) y deben estar acotados entre 0 y 1. En el caso de los filtros paso baja y alta, **Wn** solo puede tener un valor; para filtros paso banda y elimina banda, **Wn** debe definir los extremos de la banda (2 valores).

El parámetro **output** determina el tipo de salida de la función: con '**ba**' la salida será un vector con los coeficientes del numerador y un vector con los coeficientes del denominador; con '**zpk**' la salida será un vector con los ceros del filtro, un vector con los polos y un número real con la ganancia. Ejemplos:

- Filtro analógico paso alta de orden 5 con una frecuencia de corte de 500Hz, representado por sus coeficientes:

```
>>> num, den = butter(5, 500*2*pi, btype='highpass', analog=True)
>>> num
array([ 1.,  0.,  0.,  0.,  0.,  0.])
>>> den
array([ 1.0000000e+00,   1.01664074e+04,   5.16779196e+07,
       1.62350972e+11,   3.15222440e+14,   3.06019685e+17])
```

- Filtro digital paso banda de orden 5 con una banda de paso entre 300 y 400Hz, muestreado a 1000Hz y representado por los ceros, polos y ganancia:

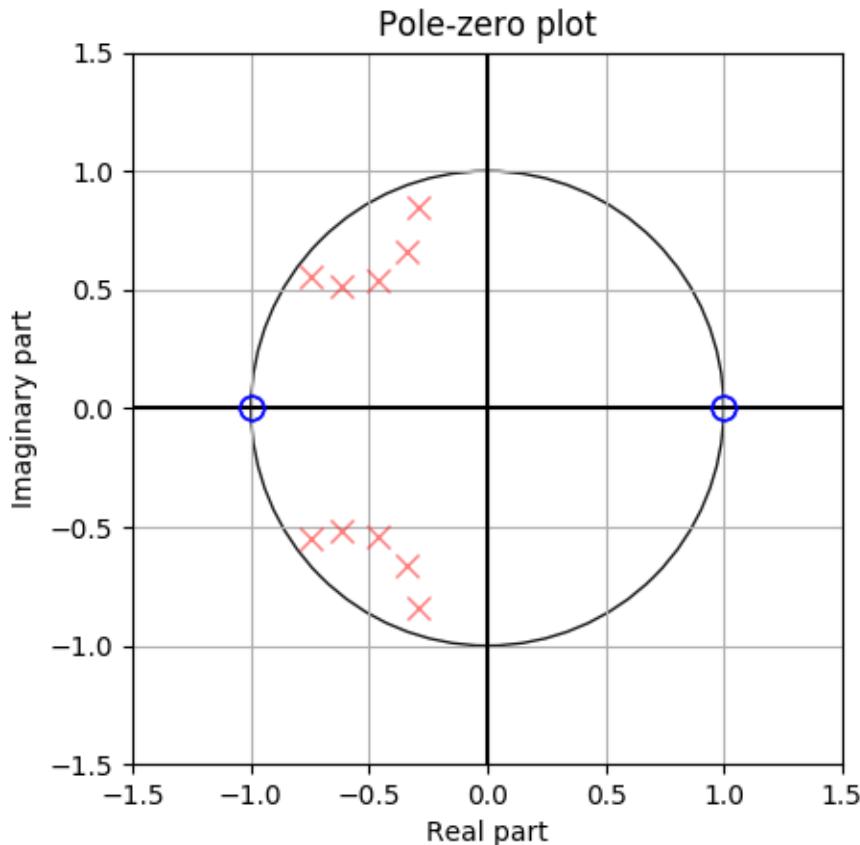
```
>>> ceros, polos, ganancia = butter(5, [300/500, 400/500],
>>> btype='bandpass', output='zpk')
>>> ceros
array([ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j, -1.+0.j, -
       1.+0.j,
       -1.+0.j, -1.+0.j, -1.+0.j])
>>> polos
array([-0.29087212-0.84461039j, -0.33793526-0.65917459j,
```

```

-0.46646902-0.54030742j, -0.33793526+0.65917459j,
-0.29087212+0.84461039j, -0.75030415+0.55222626j,
-0.61896042+0.51446337j, -0.46646902+0.54030742j,
-0.61896042-0.51446337j, -0.75030415-0.55222626j])
>>> ganancia
0.0012825810789606855

```

- `cheby1(N, rp, Wn, btype='low', analog=False, output='ba')`: función para el diseño de un filtro IIR de Chebyshev de tipo I, digital o analógico. Estos filtros son utilizados para conseguir una caída más pronunciada en las frecuencias críticas, generando a su vez un rizado constante en la banda de paso. Los parámetros utilizados son los mismos que en butter(), excepto `rp`, que determina el rizado máximo de la banda de paso, especificado en decibelios (número positivo).
- `cheby2(N, rs, Wn, btype='low', analog=False, output='ba')`: función para el diseño de un filtro IIR de Chebyshev de tipo II, digital o analógico. A diferencia del tipo I, el rizado se produce en la banda de rechazo. Los parámetros utilizados son los mismos que en butter(), excepto `rs`, que determina el rizado máximo de la banda de rechazo, especificado en decibelios (número positivo). Con la función zplane() podemos comprobar si el filtro es estable o no. La representación del filtro de Butterworth anterior en la circunferencia unidad es la siguiente:



El filtro es estable, causal y de fase mínima. En los sistemas discretos, si todos los polos se encuentran dentro de la circunferencia unidad del plano z , este será estable.

7.7.2 Simulación de filtros

- [Ifilter_zi](#)(**b**, **a**): función para obtener los valores de las condiciones iniciales de un filtro digital FIR o IIR descrito por los valores del numerador **b** y del denominador **a**. Las condiciones iniciales son equivalentes a obtener la respuesta del filtro a la señal escalón. Al establecer estas condiciones al aplicar un filtro, la salida de este empezara en el mismo valor que el primer valor de la entrada. La salida de esta función es un vector con el estado inicial del filtro, su longitud dependerá del número de coeficientes.
- [Ifilter](#)(**b**, **a**, **x**, **zi=None**): función para aplicar un filtro digital, tanto IIR como FIR, representado por sus coeficientes del numerador **b** y del denominador **a**, a una secuencia de datos de entrada **x**. Con el parámetro **zi**, se puede establecer un vector con las condiciones iniciales del filtro, que se obtienen con la función [Ifilter_zi\(\)](#). Por defecto, las condiciones iniciales serán nulas y se considerara que estado inicial es de reposo. Las condiciones iniciales del filtro se deben multiplicar por el primer elemento de la entrada (**zi=iniciales*x[0]**).

Esta función devuelve un vector con los valores de la salida del filtro **y**, si las condiciones iniciales están especificadas, también devolverá un vector con los valores de los retardos introducidos por el filtro.

Esta función utiliza el método directo transpuesto de forma II para realizar la aplicación del filtro. La siguiente ecuación describe este método:

$$y[n] = \frac{\sum_{i=0}^M b[i]x[n-i] - \sum_{j=1}^N a[j]x[n-j]}{a[0]}$$

Siendo **y[n]** la salida del filtro, **x[n]** la entrada, **a[j]** los valores de los coeficientes del denominador (N el número total) y **b[i]** los valores de los coeficientes del numerador (M el número total).

- [filtfilt](#)(**b**, **a**, **x**, **padtype='odd'**, **padlen=None**): función para aplicar un filtro digital FIR o IIR dos veces a una señal de entrada. Esta función permite obtener una salida sin ningún cambio de fase respecto a la entrada. Esta función sigue los siguientes pasos:
 - Añade valores de relleno en los extremos de la señal de entrada según los parámetros **padtype** y **padlen**.
 - Utiliza la función [Ifilter_zi\(\)](#) para establecer las condiciones iniciales del filtro en la primera pasada y las escala por **x[0]**.
 - Aplica el filtro en una dirección.
 - Vuelve a calcular las condiciones iniciales del filtro, pero ahora escaladas por el ultimo valor de la entrada.
 - Vuelve a aplicar el filtro, pero ahora en la dirección contraria.
 - Los valores de relleno son eliminados.

El resultado es análogo a un filtro de fase 0 con un orden doble al filtro inicial.

7.7.3 Análisis espectral

- `freqs(b, a, worN=None)`: respuesta en frecuencia de un filtro analógico definido por los coeficientes del numerador **b** y los del denominador **a**. Con **worN** podemos especificar un vector de frecuencias (rad/s) donde queremos obtener la respuesta en frecuencia. Si este no es especificado o es un número, la respuesta se calculará para las 200 (o el número especificado) frecuencias alrededor de las partes ‘interesantes’ de la respuesta (determinada por los ceros y los polos).
La función devuelve 2 valores: un vector de frecuencias (rad/s) y un vector de igual longitud con la magnitud de la respuesta en esas frecuencias.
- `freqz(b, a=1, worN=None, whole=False)`: respuesta en frecuencia de un filtro digital definido por los coeficientes del numerador **b** y los del denominador **a**. **worN** especifica el rango de frecuencias normalizadas en el que calcular la respuesta, por defecto calculará 512 valores igualmente espaciados en el círculo unidad. Con **whole=False**, solo se calculará las frecuencias entre 0 y la frecuencia de Nyquists (de 0 a π); con **whole=True** se calculará todo el rango de frecuencias (de 0 a 2π).
La función devuelve 2 valores: un vector de frecuencias normalizadas y un vector de igual longitud con la magnitud de la respuesta en esas frecuencias.
- `freqs_zpk(z, p, k, worN=None)`: función equivalente a `freqs()` pero utilizando la representación del filtro en ceros **z**, polos **p** y ganancia **k**.
- `freqz_zpk(z, p, k, worN=None, whole=False)`: función equivalente a `freqz()` pero utilizando la representación del filtro en ceros **z**, polos **p** y ganancia **k**.

La clave para realizar un correcto análisis espectral de un filtro, es escoger la función adecuada dependiendo si el filtro es digital o es analógico, y tener en cuenta las unidades de las frecuencias (si está normalizada o no) al representar la respuesta. Si se utiliza la función incorrecta se obtendrá un resultado completamente erróneo. Es útil también utilizar la función `semilogy()` para que el eje y esté representado en una escala logarítmica.

7.7.4 Representación de filtros

- `tf2zpk(b, a)`: devuelve la representación en ceros, polos y ganancia de un filtro lineal representado por los coeficientes del numerador **a** y los coeficientes del denominador **b**.
- `zpk2tf(z, p, k)`: devuelve la representación en coeficientes del numerador y denominador de un filtro lineal representado por los ceros **z**, polos **p** y ganancia **k**.

7.8 Audio

En el submódulo **Audio** se incluyen una serie de funciones para el tratado de archivos de audio. Utilizando los módulos pyaudio (<https://people.csail.mit.edu/hubert/pyaudio/>), incluido como un paquete externo, y audioread (<https://pypi.python.org/pypi/audioread/2.1.4>), incluido dentro de la librería labUGR, he creado una serie de funciones que adaptan las funciones básicas de audio incluidas en numpy para poder ser utilizadas también con archivos de audio comprimidos(mp3, mp4,...). Para poder ser utilizadas es necesario seguir los pasos descritos en el apartado de instalación.

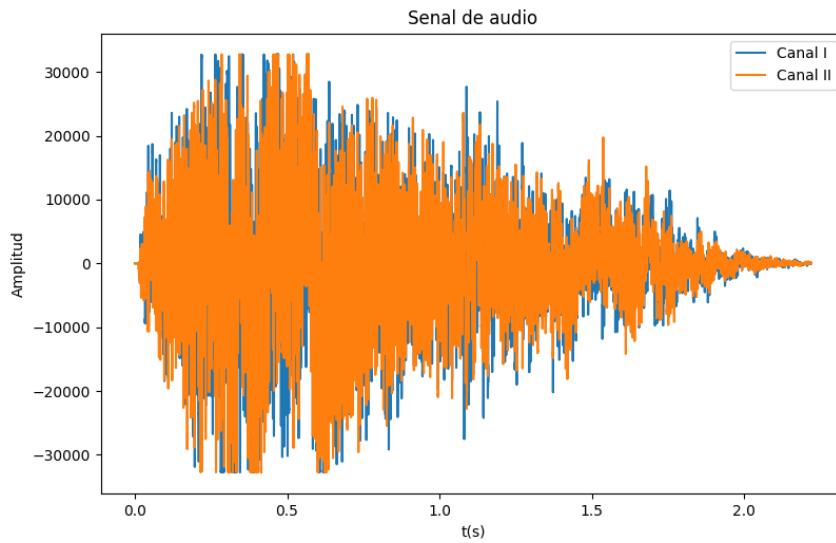
- **decode(archivo)**: función para decodificar un archivo de audio comprimido generando un archivo de audio descomprimido en formato '.wav'. Esta función utiliza el módulo audioread para seleccionar un backend disponible en el sistema y lo utiliza para decodificar el archivo. Si el archivo no se puede decodificar, o no hay ningún backend disponible, se producirá un error.
- **read(archivo)**: función para obtener los datos de un archivo de audio en un formato con el que puede ser utilizado por el resto de funciones de la librería para, por ejemplo, el análisis espectral de sus componentes en frecuencia. El archivo puede estar en formato wav o en un formato comprimido, en este caso, la función primero lo decodifica en un archivo wav temporal.
La función devuelve dos valores: un número con la frecuencia de muestreo de la señal y un vector con los valores de la señal.
- **write(filename, rate, data)**: función para guardar una señal descrita por el vector **data** y la frecuencia de muestreo **rate** en un archivo con nombre **filename**.
- **play(archivo)**: función para reproducir un archivo de audio wav o comprimido. Al igual que read, si el archivo de audio esta comprimido primero lo decodifica en un archivo wav. Esta función utiliza el módulo pyaudio para la reproducción del audio.

Por ejemplo, en **ejemplo_audio.py** se encuentra un ejemplo utilizando la función read. Esta descarga un archivo de audio mp3 automáticamente (<http://soundbite.com/grab.php?id=1272&type=mp3>) y muestra una serie de gráficos (valores en el tiempo, periodograma y espectrograma). Hay que tener en cuenta que la mayoría de archivos de audio (incluyendo el del ejemplo) son estéreo por lo que hay que separar los valores de los 2 canales antes de hacer el análisis.

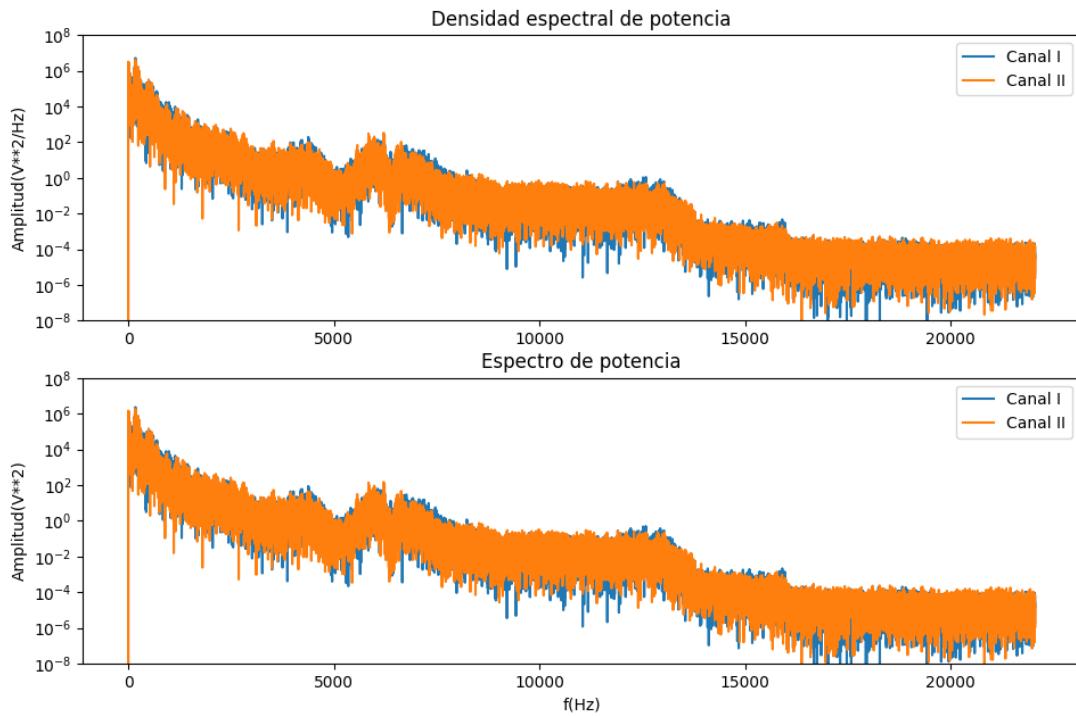
```
fs, datos = read(nombre) # fs=44100Hz, datos=vector con los valores
duracion = len(datos)/fs # 2.2197278911564626 segundos

# Separacion de los canales de audio
canal1 = datos[:, 0] # valores del canal izquierdo
canal2 = datos[:, 1] # valores del canal derecho
```

Representación de la señal en el dominio del tiempo:

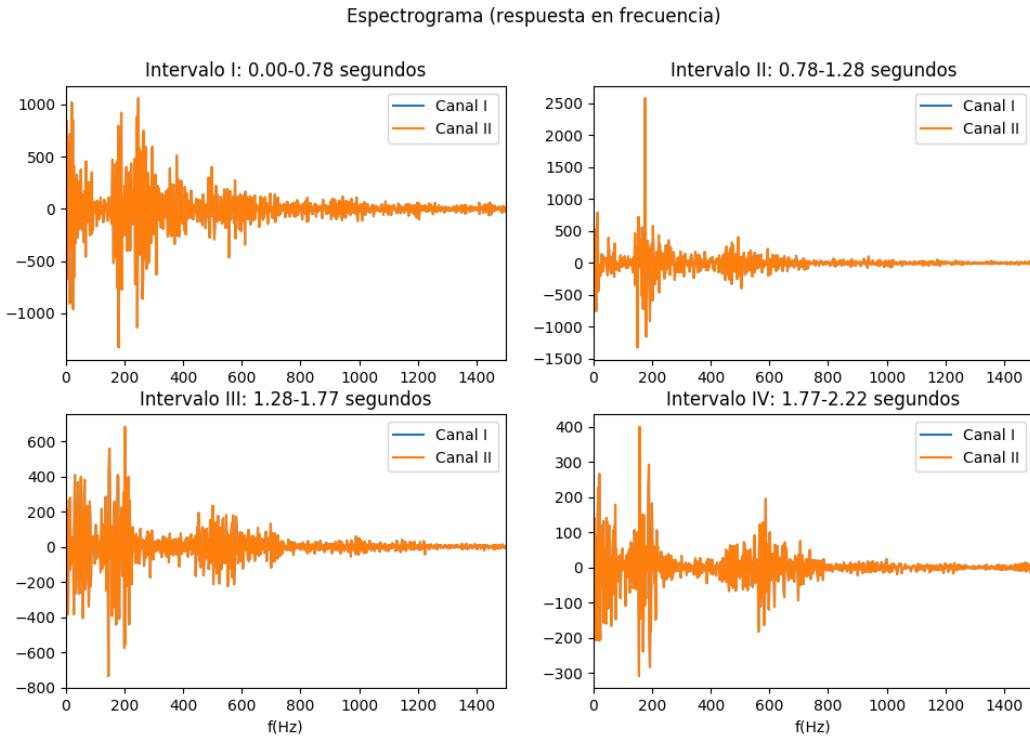


Periodograma:



Para obtener la respuesta en frecuencia de la señal mediante un spectrograma (utiliza la transformada discreta) especificamos el modo ***complex*** y una longitud de la ventana de modo que se generen 4 intervalos.

```
muestras_ventana = int(len(datos)/4)
frecuencias, tiempos, magnitud = spectrogram(canall, fs, mode='complex',
nperseg=muestras_ventana)
```



Por encima de 1.5kHz los componentes en frecuencia de la señal tienen valores muy pequeños por ello han sido excluidos.

7.9 Doc

Este submódulo contiene una serie de scripts que recogen la documentación de todas las funciones del módulo labUGR encontradas en `'funcion'.__doc__` y las traduce utilizando el módulo googletrans (<http://py-googletrans.readthedocs.io/en/latest/>) de Python, una API para traducir archivos utilizando el traductor de Google. También incluye una función, `ayuda()`, análoga a la función `help()` de Python que utiliza los archivos traducidos anteriormente para mostrar la documentación en español de las funciones.

7.10 Testing

Este submódulo contiene una serie de funciones utilizadas en el resto de submódulos para las pruebas unitarias. También incluye una función, `test_all()`, que recoge todas las pruebas unitarias de todos los submódulos y comprueba que se pasan correctamente. Esta función también comprueba que todas las dependencias externas están instaladas en el sistema y muestra en pantalla si alguna dependencia no está instalada.

7.10.1 Unit testing

Las pruebas unitarias o por su término en inglés, Unit Testing, son una serie de pruebas que tienen como finalidad la comprobación del correcto funcionamiento de las unidades de código que forman un proyecto. A diferencia de otras metodologías de realización de pruebas para el desarrollo de software como las conocidas como pruebas de caja negra, que verifican el funcionamiento del proyecto entero, las pruebas unitarias se basan en descomponer las funciones del programa en unidades individuales y aisladas para comprobar que su funcionamiento es el correcto.

Añadir pruebas unitarias a un proyecto aporta un gran número de beneficios y facilita el correcto desarrollo e integración de este. En el caso el módulo de labUGR, seguir una metodología de pruebas unitarias es ideal para estar asegurados que la instalación de este ha sido correcta. LabUGR contiene más de 150 funciones principales, así como también un gran número de funciones auxiliares que no están destinadas para el uso por parte del usuario final pero que son la base y núcleo del proyecto. Comprobar manualmente que todas estas funciones funcionan correctamente llevaría mucho tiempo y sería propenso a fallos.

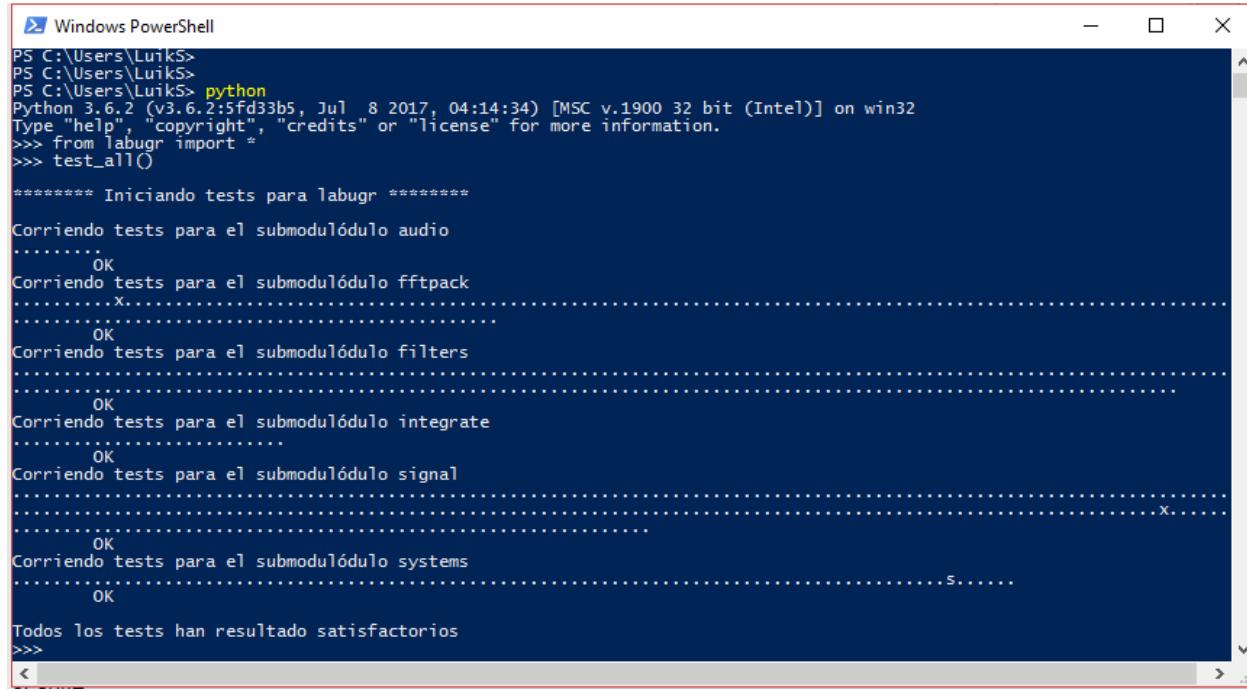
Por ello, he decidido incluir los tests incluidos en las librerías originales en mi proyecto. Estas funciones utilizan los módulos externos `pytest` (<https://docs.pytest.org/en/latest/>) y `nose` (<http://nose.readthedocs.io/en/latest/>) para realizar estas pruebas. Para instalar esos módulos podemos ejecutar el siguiente comando: `pip install labugr[test]`. Cada submódulo de este proyecto contiene una carpeta tests; al utilizar pytest en una carpeta, este módulo analiza todos los archivos de extensión .py que contienen la palabra test al inicio de su nombre y a su vez recoge las funciones que empiezan por test. Una vez las pruebas son recogidas, estas son ejecutadas y comprueban que el resultado de las funciones a probar es correcto.

```
class _TestConvolve(object):

    def test_basic(self):
        a = [3, 4, 5, 6, 5, 4]
        b = [1, 2, 3]
        c = convolve(a, b)
        assert_array_equal(c, array([3, 10, 22, 28, 32, 32, 23, 12]))
```

El ejemplo anterior es una prueba para la función `convolve()` que se encuentre en el archivo '`labugr/signal/tests/test_conv_corr.py`'. Esta comprueba que la convolución de los dos vectores utilizando la función `convolve()` es igual al resultado precalculado.

En el submódulo **testing** está incluida la función `test_all()` que hace uso de pytest, así como de las carpetas `tests` en los otros submódulos, para recoger y comprobar todas las pruebas incluidas en el proyecto. Si la instalación ha sido correcta el resultado en la terminal será el siguiente (cada punto es una prueba realizada):



```
PS C:\Users\LuikS>
PS C:\Users\LuikS>
PS C:\Users\LuikS> python
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from labugr import *
>>> test_all()

***** Iniciando tests para labugr *****
Corriendo tests para el submodulódulo audio
.....
      OK
Corriendo tests para el submodulódulo fftpack
.....
      OK
Corriendo tests para el submodulódulo filters
.....
      OK
Corriendo tests para el submodulódulo integrate
.....
      OK
Corriendo tests para el submodulódulo signal
.....
      OK
Corriendo tests para el submodulódulo systems
.....
      OK

Todos los tests han resultado satisfactorios
>>>
```

Figura 13: Captura de un resultado satisfactorio de todas las pruebas de labUGR

8 Compilación

Ejecutando el comando `pip install .` dentro de la carpeta principal del código fuente del proyecto (<https://github.com/lserraga/labUGR>) se puede compilar e instalar la librería. Para ello se necesitan una serie de herramientas instaladas en el sistema:

8.1 Numpy y Cython

Para compilar la librería labUGR se necesitan los módulos de Python NumPy y Cython. Ambos módulos se encuentran en el repositorio PyPI y se pueden utilizar usando pip: `pip install cython numpy` (o utilizando pip3 en el caso de Mac y Linux). Las funciones dentro de numpy.distutils son utilizadas para la instalación del módulo labUGR ya que proveen más claridad que las funciones para instalar paquetes que por defecto incluye Python (distutils).

El módulo Cython (<http://cython.org/>) permite utilizar extensiones de C con Python. Este es un compilador estático que permite escribir código en Python que puede interactuar con código escrito en C. Este módulo es utilizado por muchas funciones de SciPy incluidas en el módulo labUGR para acelerar la ejecución de diversas funciones que, solo utilizando Python, serían muy lentas. Los archivos de Cython (*.pyx), deben de ser compilados antes de instalar labUGR. Esto se realiza mediante el script `cythonize.py` encontrado en `labugr\scripts` y que forma parte del código fuente de SciPy.



8.2 Compiladores C++ y Fortran

Al incluir código en Fortran y C++, es necesario tener los compiladores para estos lenguajes instalados en el sistema a la hora de compilar el proyecto desde el código fuente. La mayoría de distribuciones de Linux, así como en Mac OS, tienen incluido por defecto el compilador de C++ - GCC (<https://gcc.gnu.org/>) por lo que solo es necesario instalar un compilador para Fortran. Las instrucciones se pueden encontrar en <https://gcc.gnu.org/wiki/GFortranBinaries>. Por ejemplo, en Linux Ubuntu basta con ejecutar `sudo apt-get install gfortran`.

En el caso de Windows, hay diferentes programas que implementan los compiladores de GCC: visual C++ 14(), Cygwin, MinGW, entre otros... Tras probar diferentes opciones, MinGW es el mas fácil de instalar y el que ha producido menos problemas y el que he utilizado. MinGW (<http://www.mingw.org/>) es un entorno de desarrollo minimalista que implementa los compiladores y herramientas de GNU-GCC en Windows. Las instrucciones para utilizar MinGW son las siguientes:

- Descargar el instalador desde <https://sourceforge.net/projects/mingw-w64/>. Este permite escoger la arquitectura del sistema durante la instalación (i686 para 32 bits y x86_64 para 64 bits).
- Añadir la carpeta `bin\` de MinGW a la variable de entorno PATH de Windows para que otros programas del sistema puedan utilizarlas. Por ejemplo, en el caso de la instalación de 32 bits, bin

se encuentra en **C:\Program Files (x86)\mingw-w64\i686-4.8.1-posix-dwarf-rt_v3-rev2\mingw32\bin**.

- Asignar MinGW a Python: Por defecto, Python utiliza el compilador de Microsoft Visual C++ 14 para compilar código en C y Fortran por lo que, aunque MinGW esté instalado, se producirá un error al intentar compilar la librería.

```
g95.exe:f77: labugr\fftpack\src\dfftpack\zffti.f
g95.exe:f77: labugr\fftpack\src\dfftpack\zffti1.f
error: Microsoft Visual C++ 14.0 is required. Get it with "Microsoft Visual C++ Build Tools": http://landinghub.visualstudio.com/visual-cpp-build-tools
```

Figura 14: Captura del error al no especificar MinGW como compilador

Para que Python utilice MinGW hay que modificar, o crear si no existe, el archivo **distutils.cfg** dentro de la carpeta **python/lib/distutils** y añadir el siguiente código:

```
[build]
compiler = mingw32
```

IMPORTANTE: En un sistema de 64 bits se pueden tener instaladas ambas versiones de MinGW y varias versiones de Python, pero para poder compilar la librería correctamente la variable del entorno PATH debe de estar configurada correctamente. Por ejemplo, si se desea compilar la librería en Python 3.6-32bits, los directorios de **Python36-32**, **Python36-32\Scripts** y **mingw64\bin** deben estar en PATH:

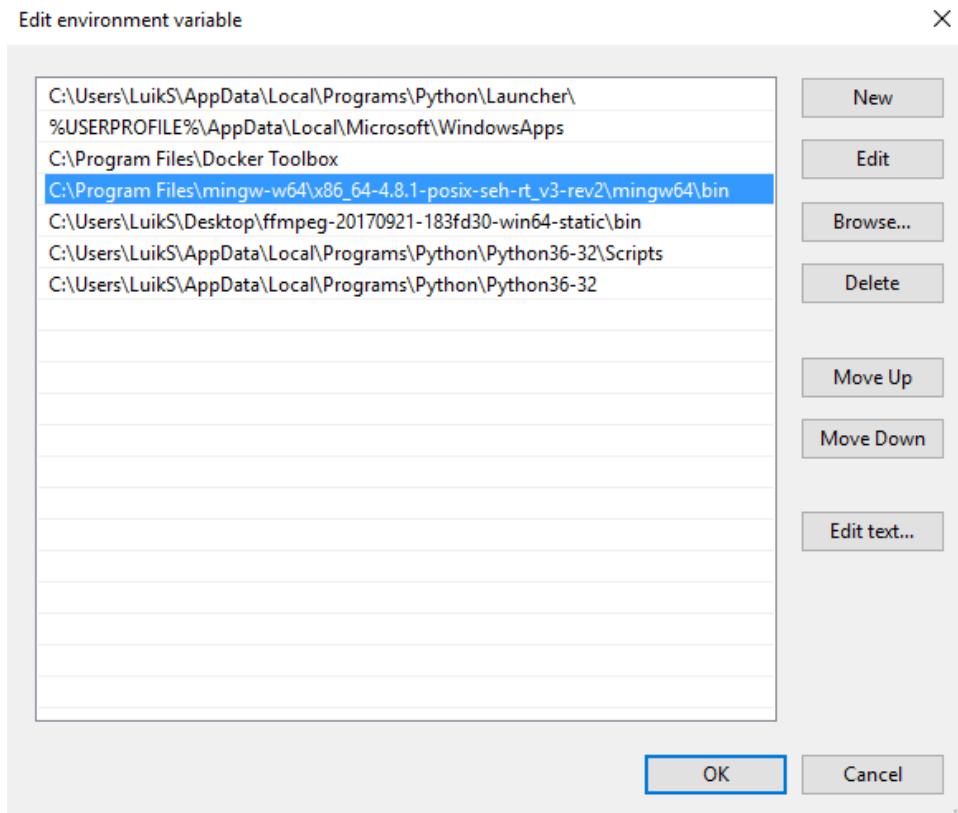


Figura 15: Ejemplo de la configuración de la variable de entorno PATH en Windows

8.3 LAPACK y BLAS

A parte de los compiladores de C y Fortran, este módulo necesita de unas librerías matemáticas especiales de bajo nivel necesarias para algunas de las funciones incluidas en labUGR (relacionadas con la integración, computación de matrices, ...). Aunque hay diferentes posibles implementaciones, todas las librerías están basadas en dos conjuntos de rutinas: BLAS y LAPACK.

BLAS (<http://www.netlib.org/blas/>) es un conjunto de rutinas de bajo nivel para realizar operaciones comunes de álgebra lineal como la suma de vectores, multiplicación escalar, producto escalar combinaciones lineales y multiplicación de matrices. Estas rutinas son utilizadas por librerías de más alto nivel como LAPACK para optimizar el cálculo matricial. Las rutinas de BLAS pueden hacer uso de elementos de hardware especiales para realizar los cálculos.

Las rutinas de BLAS se categorizan en tres subconjuntos llamados "niveles", que corresponden con el grado del polinomio en las complejidades de los algoritmos y tiempo de ejecución esperado dependiendo del tamaño de la entrada. Hay un total de tres niveles:

- Nivel 1: conjunto de operaciones que tienen principalmente un orden de complejidad lineal, $O(n)$. Es decir, para una entrada suficientemente grande n , el tiempo de computación incrementará linealmente a medida que el tamaño de la entrada aumenta.
- Nivel 2: conjunto de operaciones que tienen principalmente un orden de complejidad cuadrático, $O(n^2)$.
- Nivel 3: conjunto de operaciones que tienen principalmente un orden de complejidad cúbica, $O(n^3)$. Estas son las operaciones más complejas y que necesitan un poder de computación mayor.

LAPACK (<http://www.netlib.org/lapack/>) es una librería de código abierto bajo la licencia BSD, escrita en Fortran 90 que contiene una serie de rutinas para resolver sistemas de ecuaciones, sistemas de ecuaciones con el método de regresión por mínimos cuadrados, problemas de vectores propios y valores propios (eigenvectores), factorizaciones de matrices y una serie de rutinas para optimizar operaciones con matrices, matrices banda, densas, triangulares,... Esta está optimizada para llamar a subrutinas de BLAS de nivel 3 para realizar estas operaciones. Esto permite obtener una mayor aceleración de la computación de estas rutinas en arquitecturas específicas, incrementando el rendimiento.

La instalación de estas librerías depende del sistema operativo. En el caso de Mac OS, LAPACK-BLAS están incluidas por defecto y no es necesario realizar ningún paso extra. En el caso de distribuciones de Linux no es complicado instalarlas manualmente, por ejemplo, en Ubuntu bastaría con utilizar el siguiente comando: ***sudo apt-get install libblas-dev liblapack-dev***. Sin embargo, en Windows es bastante complicado instalar estas librerías y es una de las razones por las cuales el módulo scipy es complicado de instalar en Windows.

Hay una serie de proyectos que proveen formas de conseguir versiones optimizadas de LAPACK-BLAS en Windows:

- [Intel Math Kernel Library \(MKL, <https://software.intel.com/en-us/mkl>\)](https://software.intel.com/en-us/mkl): implementaciones de LAPACK-BLAS desarrolladas por Intel (solo funcionan sistemas con procesadores de Intel). Esta es la implementación con el mejor rendimiento, pero estas son de pago y, aunque se pueden conseguir licencias gratis para estudiantes, esto imposibilitaría la distribución del módulo de labUGR sin obligar al usuario final a instalar esta librería manualmente.
- [ATLAS and LAPACK\(<http://math-atlas.sourceforge.net/>\)](http://math-atlas.sourceforge.net/): ATLAS es una versión optimizada de rutinas de BLAS y LAPACK que son portables. La forma más fácil de utilizar esta librería es utilizando junto al compilador Mingw-64. Una de las mayores ventajas de esta librería es que existen versiones pre-compiladas para Windows 32 y 64 bits, las cuales no necesitan instalación.
- [OpenBLAS and LAPACK\(<https://github.com/xianyi/OpenBLAS>\)](https://github.com/xianyi/OpenBLAS): OpenBLAS es una versión optimizada de BLAS de código abierto que especialmente fácil de usar con el compilador Cygwin y tiene un rendimiento similar a MKL.
- [BLAS and LAPACK](#): también se pueden instalar las versiones no optimizadas de LAPACK Y BLAS a través del compilador MSVC (Microsoft Visual C++ 2014).

Para este proyecto he decidido utilizar la librería ATLAS. A pesar de que esta no es la librería que está mejor optimizada y otras librerías como OpenBLAS o MKL la superan en rendimiento, el hecho de tener versiones pre-compiladas facilita mucho la utilización de esta. Ambas versiones (32 y 64 bits) de la librería ATLAS están incluidas en el código fuente del proyecto y son utilizadas a la hora de compilar la librería desde el código fuente.

Para poder especificar la localización de estas librerías para que setuptools pueda utilizarlas a la hora de compilar labUGR debemos crear un archivo llamado **site.cfg** en el directorio del proyecto con la ubicación con la versión correspondiente de la librería. Durante el proceso de compilación, setuptools intenta encontrar este archivo en el directorio fuente, y utiliza los valores que hay dentro.

Estas se encuentran en la carpeta atlas-3.10.1-sse2-32 para la versión de 32 bits y en la carpeta atlas-3.11.38-sse2-64 para la versión de 64 bits. Este archivo es generado dinámicamente al utilizar **setup.py** utilizando el siguiente código:

```
#Directorio de trabajo
directorio = os.path.abspath(os.path.dirname(__file__))

#Funcion para determinar si python es 32 o 64 bits
def get_bitness():
    bits, _ = architecture()
    return '32' if bits == '32bit' else '64' if bits == '64bit' else None

#Directorio atlas necesario para compilar desde el código fuente en windows
atlas_compil = """[atlas]
include_dirs = {dir}\atlas-builds\{version}\include
library_dirs = {dir}\atlas-builds\{version}\lib
atlas_libs = numpy-atlas
lapack_libs = numpy-atlas
"""

#Con site.cfg podemos especificar compiladores
with open(os.path.join(directorio,'site.cfg'), 'w') as f:
    #Solo queremos utilizar site.cfg cuando estamos en windows
    if os.name == 'nt':
        if (get_bitness()=='32'):
            atlas_compil=atlas_compil.format(version="atlas-3.10.1-sse2-32",
                                              dir=directorio)
        else:
            atlas_compil=atlas_compil.format(version="atlas-3.11.38-sse2-64",
                                              dir=directorio)
    f.write(atlas_compil)
    #Si no es windows, cuando with cierra el archivo, este se queda en blanco
```

Este código genera el archivo **site.cfg** en el caso en el que el sistema sea Windows ('nt'). Por ejemplo, el resultado final tiene la siguiente forma:

```
[atlas]
include_dirs = C:\Users\LuikS\Desktop\labugr\atlas-builds\atlas-3.10.1-sse2-32\include
library_dirs = C:\Users\LuikS\Desktop\labugr\atlas-builds\atlas-3.10.1-sse2-32\lib
atlas_libs = numpy-atlas
lapack_libs = numpy-atlas
```

8.4 Modo desarrollador

La herramienta pip contiene un modo de instalación de paquetes conocido como modo desarrollador. Instalando un paquete de Python con pip añade todos los archivos de las funciones y los archivos compilados en el directorio de instalación de Python. En este modo, cuando se realizan cambios en el código fuente estos no se ven reflejados en el paquete hasta que este se vuelva a compilar e instalar.

Para que los cambios del código fuente se apliquen dinámicamente hay que utilizar el modo desarrollador de pip (***pip3 install -e .***). Este modo no copia los archivos en el directorio de Python, sino que estos son guardados en una carpeta ***build*** en el directorio principal del paquete y se enlaza con Python a través de un archivo creado en el directorio de instalación de Python (***labugr.egg-link***) y una entrada en el archivo ***easy-install.pth*** con la localización del código fuente del paquete.

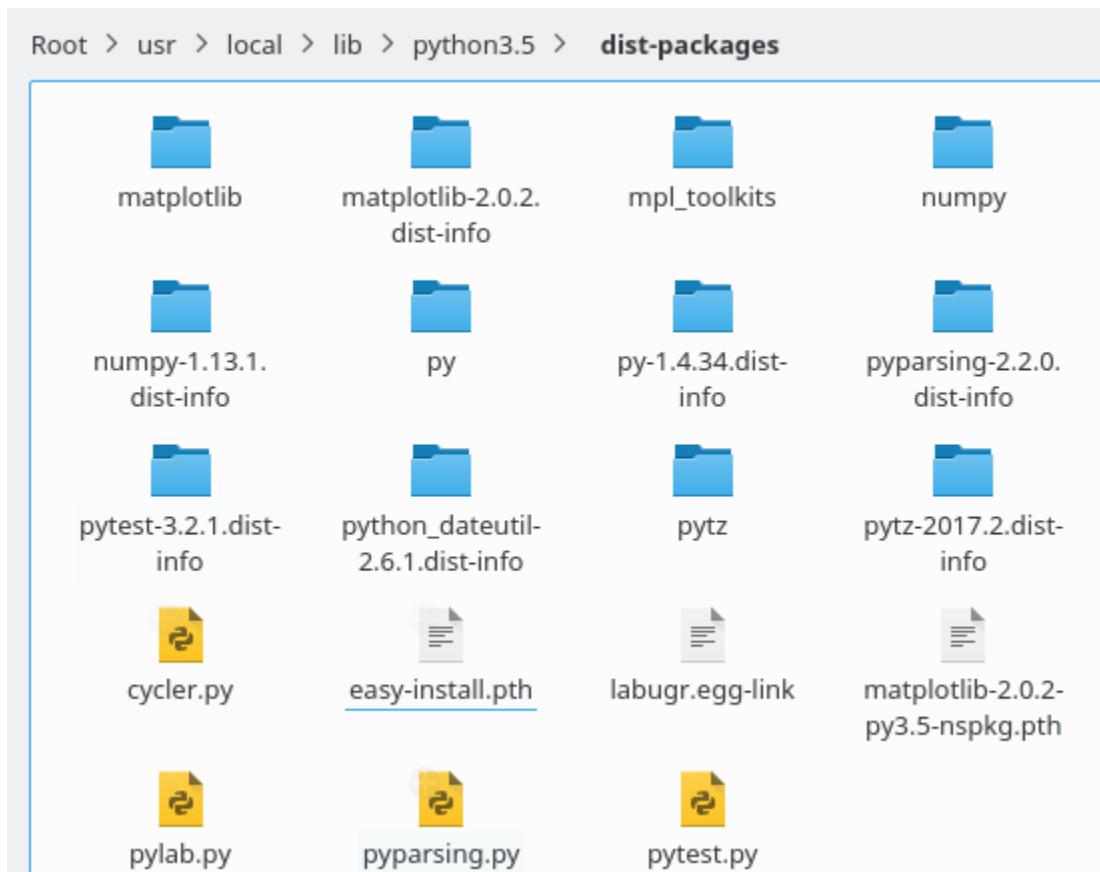


Figura 16: Carpeta 'dist-packages' al instalar labUGR en modo desarrollador



Figura 17: Contenidos del archivo 'easy-install.pth' al instalar labUGR en modo desarrollador

Para desinstalar un paquete instalado de este modo hay que eliminar la carpeta ***build*** del directorio fuente, el archivo ***labugr.egg-link*** y la entrada en el archivo ***easy-install.pth***.

9 Distribución

Una de las claves fundamentales de Python, como la de la mayoría de proyectos open source, es el hecho de poder distribuir tu código fácilmente para que otros desarrolladores puedan utilizarlo en sus propios proyectos. Python proporciona un repositorio público gratuito para alojar proyectos en la nube, 'the Python Package Index' o 'PyPi', que es la forma estándar para compartir paquetes de Python.

En el momento en que 'labUGR' se registró por primera vez en PyPi, había un total de 107.345 paquetes alojados en el repositorio. Tres meses después la suma asciende a 118.426 y crece aún más cada día.

Module Counts

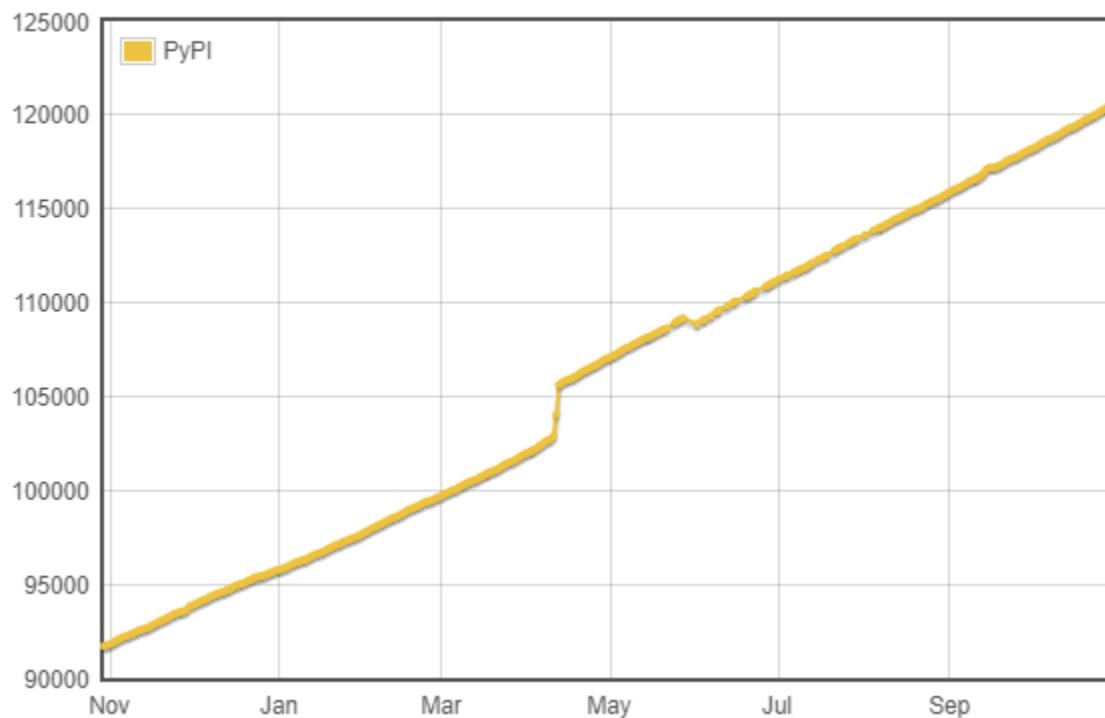


Figura 18: Número de módulos de Python alojados en el repositorio PyPi entre noviembre de 2016 y Octubre de 2017

Como hemos podido comprobar en el apartado anterior, instalar la librería labUGR desde el código fuente es complicado y necesita numerosas herramientas adicionales a Python, especialmente en el caso de Windows. Por ello, es necesario encontrar una forma de empaquetar el proyecto e incluir todas las dependencias en un paquete que se puede instalar fácilmente. Esto se puede hacer gracias a la herramienta Wheel (<https://pypi.python.org/pypi/wheel>) de Python.

9.1 Wheel

Wheel es un formato de compresión similar a zip, con extensión una extensión '.whl' que permite incluir código precompilado específico para los diferentes sistemas operativos directamente en el paquete. El proyecto labUGR contiene código precompilado en C y Fortran, así como dependencias de librerías externas como Atlas. Empaquetar todas estas dependencias en una Wheel permite que el usuario final pueda instalar la librería sin necesidad de tener compiladores para C y Fortran ni la librería Atlas instalada.

Para crear una Wheel para un paquete es necesario tener instalado el módulo Wheel de Python y utilizar el comando (***pip wheel --no-deps***). Este genera un archivo Wheel con la librería empaquetada que será compatible con la versión de Python y el sistema operativo utilizado. La opción (--no-deps) es utilizada para que pip no genere Wheels para los módulos externos de Python. Una vez empaquetado el proyecto, este se puede instalar con pip (***pip install .\labugr-1.0.5-cp36-cp36m-win32.whl***).

```

Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Users\LuikS> cd ..\Desktop\labugr\
PS C:\Users\LuikS\Desktop\labugr> Measure-Command {pip install .}

Days          : 0
Hours         : 0
Minutes       : 4
Seconds       : 28
Milliseconds  : 667
Ticks         : 2686671727
TotalDays     : 0.00310957375810185
TotalHours    : 0.0746297701944444
TotalMinutes  : 4.47778621166667
TotalSeconds  : 268.6671727
TotalMilliseconds : 268667.1727

PS C:\Users\LuikS\Desktop\labugr>
PS C:\Users\LuikS\Desktop\labugr>
PS C:\Users\LuikS\Desktop\labugr>
PS C:\Users\LuikS\Desktop\labugr>
PS C:\Users\LuikS\Desktop\labugr> pip uninstall labugr -y
Uninstalling labugr-1.0.5:
  Successfully uninstalled labugr-1.0.5
PS C:\Users\LuikS\Desktop\labugr> Measure-Command {pip install .\labugr-1.0.5-cp36-cp36m-win32.whl}

Days          : 0
Hours         : 0
Minutes       : 0
Seconds       : 7
Milliseconds  : 38
Ticks         : 70385825
TotalDays     : 8.14650752314815E-05
TotalHours    : 0.00195516180555556
TotalMinutes  : 0.1173097083333333
TotalSeconds  : 7.0385825
TotalMilliseconds : 7038.5825

PS C:\Users\LuikS\Desktop\labugr>

```

Figura 19: Diferencias entre los tiempos de instalación utilizando Wheel y código fuente

Utilizar este método de empaquetado no solo facilita la instalación de la librería para el usuario final, sino que también lo convierte en un proceso mucho más rápido. En el ejemplo mostrado en la captura anterior podemos ver que instalando la librería desde el código fuente tarda alrededor de 4 minutos y medio, en cambio, utilizando una Wheel tarda 7 segundos. Dependiendo de las características del hardware del sistema, la instalación mediante el código fuente podría tardar mucho más.

9.1.1 Etiquetas para las Wheels

El nombre del archivo Wheel (.whl) determina para qué sistemas esta es compatible (detallado en el [PEP 427](#)). Al construir una Wheel, la propia herramienta asigna automáticamente una serie de etiquetas que más tarde serán usadas por la herramienta de instalación de paquetes pip para seleccionar la Wheel adecuada para un sistema. El formato para el nombre del archivo Wheel es el siguiente:

{distribución}-{versión}-{versión Python}-{versión abi}-{plataforma}.whl

- Distribución: el nombre del paquete, en el caso de este proyecto, labugr.
- Versión: versión del paquete, 1.0 por ejemplo.
- Versión Python: versión o versiones de Python con las que el paquete es compatible. Por ejemplo: py27 para 2.7.x o py3, para todas las versiones de python3.
- Versión abi: define la interfaz de aplicación binaria que utiliza el paquete. Esta es la interfaz básica de menor nivel entre la librería de Python y el sistema operativo y determina detalles acerca de la forma de llamar a las funciones, que formato o la forma de hacer llamadas al sistema. Esta etiqueta es necesaria cuando se incluye código diferente de Python en el paquete.
- Plataforma: define la plataforma con la que paquete es compatible. Por ejemplo: win32 para Windows de 32 bits.

Las últimas 3 etiquetas son conocidas como etiquetas de compatibilidad, especificadas en el [PEP 425](#). En el caso de que todo el código del paquete sea Python existen Wheels universales (**py2.py3-none-any.whl**) o específicas para Python 2 (**py2-none-any.whl**) o Python3 (**py3-none-any.whl**). Estos paquetes serán compatibles con cualquier sistema operativo siempre y cuando tengan instalada la versión apropiada de Python.

En el caso de este proyecto, al incluir código compilado en C y Fortran, es necesario incluir las etiquetas de compatibilidad y crear Wheels para cada versión de Python en cada plataforma. En la figura 20 se puede observar una lista de las Wheels compiladas disponibles para la versión 1.0.5 de labUGR en el repositorio de PyPi ([repositorio](#)).

File	Type	Py Version	Uploaded on	Size
labugr-1.0.5-cp34-cp34m-macosx_10_12_x86_64.whl (md5)	Python Wheel	cp34	2017-10-01	3MB
labugr-1.0.5-cp34-cp34m-manylinux1_i686.whl (md5)	Python Wheel	cp34	2017-10-01	13MB
labugr-1.0.5-cp34-cp34m-manylinux1_x86_64.whl (md5)	Python Wheel	cp34	2017-10-01	15MB
labugr-1.0.5-cp34-cp34m-win32.whl (md5)	Python Wheel	cp34	2017-10-01	8MB
labugr-1.0.5-cp34-cp34m-win_amd64.whl (md5)	Python Wheel	cp34	2017-10-01	10MB
labugr-1.0.5-cp35-cp35m-macosx_10_12_x86_64.whl (md5)	Python Wheel	cp35	2017-10-01	3MB
labugr-1.0.5-cp35-cp35m-manylinux1_i686.whl (md5)	Python Wheel	cp35	2017-10-01	13MB
labugr-1.0.5-cp35-cp35m-manylinux1_x86_64.whl (md5)	Python Wheel	cp35	2017-10-01	15MB
labugr-1.0.5-cp35-cp35m-win32.whl (md5)	Python Wheel	cp35	2017-10-01	8MB
labugr-1.0.5-cp35-cp35m-win_amd64.whl (md5)	Python Wheel	cp35	2017-10-01	10MB
labugr-1.0.5-cp36-cp36m-macosx_10_12_x86_64.whl (md5)	Python Wheel	cp36	2017-10-01	3MB
labugr-1.0.5-cp36-cp36m-manylinux1_i686.whl (md5)	Python Wheel	cp36	2017-10-01	13MB
labugr-1.0.5-cp36-cp36m-manylinux1_x86_64.whl (md5)	Python Wheel	cp36	2017-10-01	15MB
labugr-1.0.5-cp36-cp36m-win32.whl (md5)	Python Wheel	cp36	2017-10-01	8MB
labugr-1.0.5-cp36-cp36m-win_amd64.whl (md5)	Python Wheel	cp36	2017-10-01	10MB

Figura 20: Versiones de Wheels de labUGR 1.0.5 alojadas en PyPI

Por ejemplo la Wheel [labugr-1.0.5-cp34-cp34m-win32.whl](#) es para la versión 1.0.5 de labUGR compatible con Python 3.4.x en Windows 32 Bits. Al instalar la librería con pip, este selecciona automáticamente la versión correspondiente para el sistema.

9.2 Mac

En Mac OS no es necesario crear versiones de la Wheel para 64 bits y 32 bits ya que la misma versión de Python es compatible con ambos sistemas.

9.2.1 Pyenv

Para poder tener diferentes versiones de Python instaladas en mi sistema y facilitar así diferentes pruebas y poder crear fácilmente las diferentes Wheels he utilizado el módulo pyenv (<https://github.com/pyenv/pyenv>). Esta es una herramienta que te permite cambiar fácilmente entre las diferentes versiones de Python instaladas a través de pyenv.

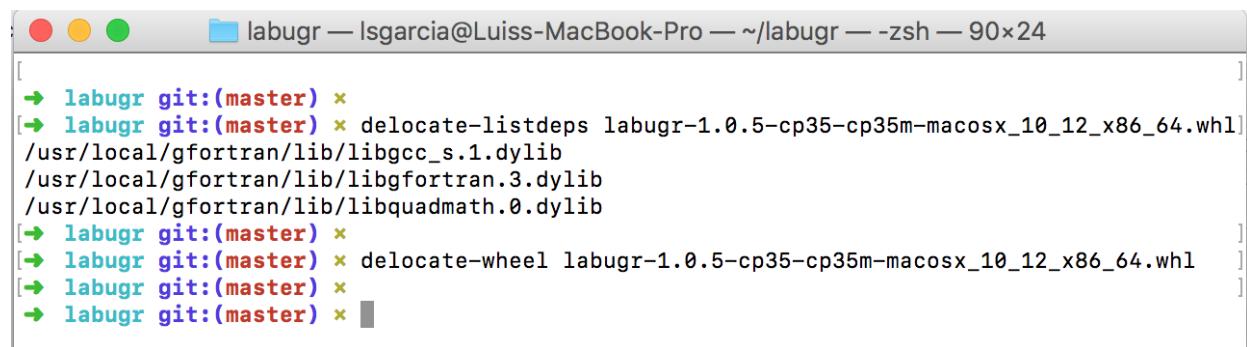


```
[→ labugr git:(master)
[→ labugr git:(master) python --version
Python 2.7.10
[→ labugr git:(master) pyenv versions
* system (set by /Users/lsgarcia/.pyenv/version)
  3.4.7
  3.5.4
  3.6.2
[→ labugr git:(master) pyenv global 3.5.4
[→ labugr git:(master)
[→ labugr git:(master) python --version
Python 3.5.4
```

Figura 21: Captura de la utilización de pyenv para el control de versiones de Python en Mac OS

9.2.2 Delocate

En Mac existe una herramienta para localizar todas las dependencias externas de un módulo de Python empaquetado y añadir estas a la Wheel. Este módulo se llama delocate (<https://github.com/matthew-brett/delocate>) y tiene dos funciones principales: **delocate-listdeps** para mostrar las dependencias y **delocate-wheel** para copiar estas dependencias dentro de la Wheel y ajustar los vínculos de estas librerías con el código dentro de Python.



```
[→ labugr git:(master) ×
[→ labugr git:(master) × delocate-listdeps labugr-1.0.5-cp35-cp35m-macosx_10_12_x86_64.whl
/usr/local/gfortran/lib/libgcc_s.1.dylib
/usr/local/gfortran/lib/libgfortran.3.dylib
/usr/local/gfortran/lib/libquadmath.0.dylib
[→ labugr git:(master) ×
[→ labugr git:(master) × delocate-wheel labugr-1.0.5-cp35-cp35m-macosx_10_12_x86_64.whl
[→ labugr git:(master) ×
[→ labugr git:(master) ×
```

Figura 22: Dependencias externas de labUGR en Mac OS utilizando 'delocate-listdeps'

En la figura 22 vemos las dependencias externas de labUGR. EL compilador de c (libgcc), el compilador de fortran(libgfortran) y la librería quadmath necesaria para las funciones de integración.

9.2.3 Script

El siguiente script automatiza la construcción de las Wheel en cada una de las versiones de Python, las instala para correr los tests, añade las dependencias externas y las sube al repositorio PyPi.

```
# Por si se llama desde otra carpeta cambiar el directorio
# a labugr
DIR=$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
cd $DIR/..

# Para que pyenv funcione correctamente
source ~/.bashrc

# Versiones de Python
py_versions=('3.4.7' '3.5.4' '3.6.2')
aux=4

for py in ${py_versions[@]};do
    # Activando la versión de python correspondiente
    pyenv global $py

    # Instalando requisitos y generando la wheel
    pip install wheel numpy cython
    pip wheel --no-deps . -w wheelhouse/

    # Instalando la librería y corriendo los tests
    pip install pytest nose
    wheel=$(find . -name "*cp3$aux*")
    pip install $wheel
    python -c 'import labugr; labugr.test_all()'
    let "aux+=1"
done

# Añadiendo las dependencias externas a la wheel
pyenv global system
pip3 install delocate
for whl in wheelhouse/*.whl
do
    delocate-wheel $whl
done

# Cargamos la contraseña de pipy desde un archivo txt
TWINE_PASSWORD=$(cat scripts/pipyPass.txt)
pip3 install twine
# Subiendo las wheels a pypi
twine upload wheelhouse/*.whl -u lserraga -p "${TWINE_PASSWORD}"
```

9.3 Windows

En el caso de Windows hay dos posibilidades, versiones para Windows de 32 bits y para Windows de 64 bits. Debido a que en Windows no existen herramientas como delocate o auditwheel, las dependencias externas se tienen que añadir manualmente al archivo '.whl' (se puede utilizar cualquier herramienta para la compresión de archivos como WinRAR). Las bibliotecas de enlace dinámico (.dll) que este proyecto utiliza son las siguientes:

- Los .dll del compilador mingw-64.
- El dll de la librería atlas incluida en el repositorio de GitHub de labUGR.

Estos archivos dll dependen del tipo de arquitectura del sistema y tienen diferentes versiones y localizaciones dependiendo si Windows es de 32 o 64 bits.

9.4 Linux

La distribución de paquetes que incluyen archivos binarios compilados en Windows y Mac OS es simple ya que, como hemos visto en los apartados anteriores, basta con añadir la etiqueta correspondiente a la plataforma en la Wheel. A diferencia de estos sistemas operativos, Linux no está estandarizado y hay cientos de distribuciones diferentes ya sean basadas en RPM como Red Hat Linux o CentOS; basadas en Debian como Ubuntu y sus variantes; o basadas en Gentoo como Chrome OS, el sistema operativo de los portátiles Chromebooks.

El gran número de configuraciones posibles hace que distribuir código compilado en Linux sea muy complicado. El código compilado en una de estas distribuciones tendrá problemas para ejecutarse en otras distribuciones de Linux, e incluso puede tener problemas cuando diferentes librerías del sistema en la misma plataforma. Debido a esto, PyPi no permite subir paquetes con etiquetas como las utilizadas en Windows y Mac OS. Etiquetas como linux_i686 o linux_x86_64 son demasiado ambiguas y no aseguran que el paquete funcione. Esto es lo que sucede al intentar subir una Wheel a PyPi con las etiquetas estándares:

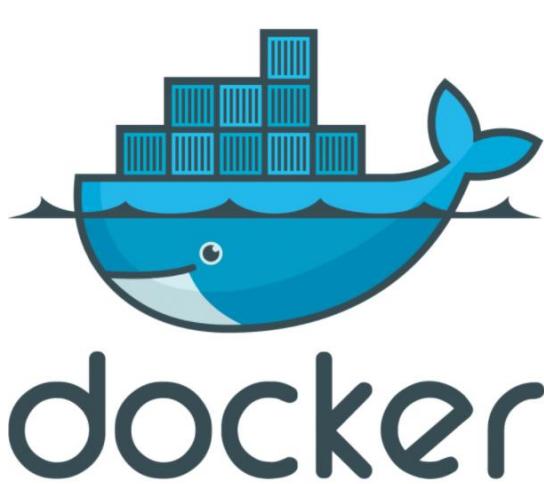
```
osboxes@osboxes:~/Labugr$ twine upload labugr-0.1.2.1-cp35-cp35m-linux_x86_64.whl
Uploading distributions to https://upload.pypi.org/legacy/
Enter your username: lserraga
Enter your password:
Uploading labugr-0.1.2.1-cp35-cp35m-linux_x86_64.whl
[=====] 1268356/1268356 - 00:00:20
HTTPError: 400 Client Error: Binary wheel 'labugr-0.1.2.1-cp35-cp35m-linux_x86_64.whl' has an unsupported platform tag 'linux_x86_64'. for url: https://upload.pypi.org/legacy/
osboxes@osboxes:~/Labugr$
```

Figura 23: Captura del error obtenido al intentar alojar una Wheel de Linux en PyPi sin las etiquetas adecuadas

En el “Python Enhancement Proposal” 513 ([PEP 513](#)) se define la solución a este problema. En este, se define una nueva etiqueta, manylinux1, que asegura que el paquete funcionará en un gran número de distribuciones de Linux. Los estándares para que una Wheel pueda ser elegible para llevar esta etiqueta son las siguientes:

- Depender solo de un limitado número de librerías externas listadas en el siguiente enlace: [The-manylinux1-policy](#).
- De las librerías externas, solo depender de versiones antiguas.
- Depender solo de ABI kernels comunes.

En base a PEP 513 y para facilitar en trabajo a los desarrolladores de Python, la organización PyPA ha creado el proyecto manylinux (<https://github.com/pypa/manylinux>). En se incluyen dos imágenes de Docker que cumplen con los estándares para utilizar la etiqueta manylinux, una para Linux de 64-bits (quay.io/pypa/manylinux1_x86_64) y otra para Linux de 32-bits (quay.io/pypa/manylinux1_i686).



9.4.1 Docker

Docker es una herramienta que permite crear contenedores de Linux para obtener entornos de programación aislados y lo más importante, replicables en cualquier plataforma donde puedas instalar Docker (Mac OS, Linux y Windows).

Docker, virtualiza el sistema operativo y genera los contenedores virtuales aislados encima de ese OS. Esta herramienta trae grandes ventajas al mundo del desarrollo de software y, en los últimos años, se ha incrementado su uso de manera exponencial.

Gracias a Docker y las imágenes de manylinux podemos crear contenedores para crear las Wheels para Linux. Dentro de estas imágenes, las diferentes versiones de Python están incluidas, así como la herramienta Wheel y Auditwheel.

9.4.1.1 Auditwheel

El módulo auditwheel (<https://github.com/pypa/auditwheel>) es equivalente al módulo delocate en Mac OS. Aparte de añadir las dependencias externas a la Wheel, este módulo también asigna las etiquetas correspondientes a la versión de manylinux correspondiente.

Contenidos del archivo docker-compose.yml:

```
#Docker-compose para la creación de manylinux wheels
#en Linux 32 bits y 64 bits
version: '3'

services:
    #Creando el container para linux 32 bits
    linux32:
        container_name: linux32
        #Especificamos la imagen a utilizar
        image: quay.io/pypa/manylinux1_i686
        #Link para que el script se pueda utilizar desde
        #dentro del contenedor
        volumes:
            -./scripts:/scripts
            #Script para la creación de las wheels
            command: bash scripts/wheel.sh <contraseña PyPi>

    #Mismo proceso para linux 64 bits
    linux64:
        container_name: linux64
        image: quay.io/pypa/manylinux1_x86_64
        volumes:
            -./scripts:./scripts
            command: bash scripts/wheel.sh <contraseña PyPi>
```

Con este archivo y usando el comando ***docker-compose up*** creamos 2 contenedores, uno con manylinux de 32 bits y otro con manylinux de 64 bits y ejecuta el siguiente script dentro de ambos contenedores (*wheel.sh*):

```
#Versiones de Python para las que queremos crear las wheels
pythonV='cp34-cp34m cp35-cp35m cp36-cp36m'

#Descargamos el paquete de github
git clone https://github.com/lserraga/labugr.git

#Instalamos la librería atlas para la compilación de labugr
yum install -y atlas-devel

#Para cada versión de python instalamos numpy, cython y creamos una wheel
for version in $pythonV
do
    ENV=opt/python/$version/bin
    "${ENV}/pip" install numpy cython
    "${ENV}/pip" wheel --no-deps labugr/ -w wheelhouse/
done

#Para cada wheel creada utilizamos auditwheel para comprobar que esta
#cumple con los estándares de manylinux y establecer las etiquetas
#adecuadas para que pip pueda utilizar la wheel
for whl in wheelhouse/*.whl
do
    auditwheel repair $whl -w wheelhouseOK/
done

#La contraseña de PyPi es el primer parámetro al llamar al script
TWINE_PASSWORD=$1
#Instalamos twine y subimos las wheels creadas a pipy
"${ENV}/pip" install twine
"${ENV}/twine" upload wheelhouseOK/*.whl -u lserraga -p "${TWINE_PASSWORD}"
```

Con un único script generamos las Wheels para todas las versiones de Python que nos interesan (3.4, 3.5 y 3.6) y las subimos a la plataforma PyPi.

10 Aplicación práctica: laboratorio de señales analógicas y digitales

En este apartado se incluyen una serie de prácticas propuestas para las asignaturas de sistemas lineales y señales digitales impartidas en la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada. También se incluye una ‘práctica 0’ como introducción a Python y la librería labUGR.

10.1 Práctica 0: Introducción a Python

A diferencia de lenguajes como C++ o Java la sintaxis de Python es bastante similar al lenguaje natural y por lo tanto mucho más fácil de entender. En Python, no necesitamos declarar variables ni gestionar memoria como en C. En esta práctica podemos ejecutar los comandos directamente en la terminal de Windows (abrir terminal y ejecutar el comando Python, en cuanto aparezca >>> podemos escribir comandos y ejecutarlos pulsando enter). En las próximas prácticas habrá que crear scripts y ejecutarlos mediante un IDE (PyCharm recomendado).

Cuando creamos una variable, Python asigna automáticamente su tipo:

- **Números enteros**: conocidos como int.
- **Números reales**: conocidos como floats. Si queremos definir un numero entero como float solo basta con añadir un ‘.’ al final del número (a=2.)
- **Strings**: son cadenas de caracteres. Para definir una variable como string basta con poner el texto dentro de unas comillas, simples o dobles (a='Esto es una string'). Podemos concatenar strings simplemente con el operador suma, también podemos concatenar otros tipos de variables convirtiéndolos a string con str().

```
>>> a = 'Esto es una string'
>>>b = 'concatenada '
>>>c = a + b + str(1)
>>> print(c)
Esto es una string concatenada 1
```

- **Boolean**: esta variable solo puede ser verdadero (True) o falso (False). True y False se conocen como palabras reservadas que no pueden ser utilizadas como nombres de variables ya que tienen una función por sí mismas.

Para utilizar la librería labUGR debemos importarla en cada programa que creamos:

```
from labugr import *
```

Esta línea le dice al compilador que importe todas las funciones de la librería labUGR. Ahora podemos llamar a las funciones de esta librería por ejemplo, array([]).

MOSTRANDO VALORES

Con Python hay dos formas de mostrar valores por pantalla: Llamando directamente el nombre de la variable lo que mostrará el valor tal y como Python lo guarda en memoria.

```
>>>a = 'Esto es una string'

>>>a

'Esto es una string'

>>>b = array([(1,2), (3,4)])

>>>b

array([[1, 2],
       [3, 4]])
```

O utilizando print(), la manera correcta de representar valores en pantalla.

- Texto: poner el texto entre comillas (pueden ser dobles o simples).

```
>>>print("Texto aqui")

Texto aqui
```

- Variables: directamente poner el nombre de la variable

```
>>>print(b)

[[1 2]
 [3 4]]
```

- Texto y variables: utilizando el siguiente formato podemos añadir variables a un texto utilizando lo que se llaman formatters: %d para números enteros (si la variable es real no mostrará los decimales), %f para números reales (con %.2f podemos especificar el número de decimales) y %s para texto (lo que hace este último es convertir la variable a string con str()). Para mostrar una matriz utilizar %s. **print("Texto %s mas variables %d" % (a, b))** siendo b un número.

```
>>>print("Texto %s mas variables \n %s" % (a, b))

Texto Esto es una string mas variables

[[1 2]
 [3 4]]
```

- Caracteres especiales: para utilizar alguno caracteres tenemos que utilizar “\”, los más importantes son:

\\\	Barra invertida (\)
\'	Comilla simple (')
\"	Comilla ("")
\n	Nueva línea (print siempre acaba con nueva línea)
\t	Tabulación

CREACION DE MATRICES Y VECTORES

Python no incluye funciones para trabajar con matrices, por ello debemos utilizar las funciones dentro de la librería labugr. Para crear matrices utilizamos la función array():

```
>>>a = array([(2,3,4),(3,3,3),(6,6,6)], dtype=float)
```

Dentro de los [] cada fila es definida por los elementos dentro de los paréntesis. A su vez cada elemento de las columnas está separado por una coma. Esto crea una matriz 3x3 de números reales. Podemos forzar el tipo de los elementos de la matriz con dtype, aunque no hace falta ponerlo y por defecto será enteros.

```
>>>print(a)
[[ 2.,  3.,  4.],
 [ 3.,  3.,  3.],
 [ 6.,  6.,  6.]]
```

Para obtener las dimensiones de un array podemos utilizar la función shape que devuelve dos valores: el primero es el número de filas (matriz.shape[0]) y el segundo el número de columnas(matriz.shape[1]). Si utilizamos matriz.shape[:] el resultado son los dos numeros.

```
>>>print('La matriz tiene %i filas y %i columnas'%(a.shape[0],a.shape[1]))
La matriz tiene 3 filas y 3 columnas
```

Otra forma de crear matrices es con las funciones arange(), reshape() y linspace():

- Arange(x,y,z): nos permite crear una matriz de una fila empezando en x hasta y (y no se incluye) con una distancia de z entre los números.

- `Reshape(x,y)`: nos permite cambiar las dimensiones de la matriz. Dará error si el número de elementos no es el mismo (por ejemplo, una matriz (1,15) no se puede transformar en una (3,3)).

```
>>>a = arange(0,15,2)

>>>print(a)

[ 0,  2,  4,  6,  8, 10, 12, 14]

>>>a.reshape(2,4)

array([[ 0,  2,  4,  6],
       [ 8, 10, 12, 14]])
```

- `Linspace(x, y, z, dtype)`: tiene la misma función que arange pero con z establecemos el número de elementos que queremos entre x e y (ambos incluidos), no la distancia entre valores. Dtype no es necesario ponerlo y por defecto será float.

```
>>> linspace(0,120,16,dtype=int).reshape(4,4)

array([[ 0,   8,  16,  24],
       [ 32,  40,  48,  56],
       [ 64,  72,  80,  88],
       [ 96, 104, 112, 120]])
```

MANIPULACIÓN DE MATRICES

Para acceder a los elementos dentro de una matriz basta con conocer el índice del elemento. Si tenemos una matriz 3x3 llamada a, podemos acceder individualmente cada elemento con `a[x,y]` siendo x el índice de la fila e y el índice de la columna. IMPORTANTE, hay que tener en cuenta que la primera fila y columna tienen índice 0 y no 1 como en Matlab, por lo que para coger el elemento en la tercera fila y segunda columna necesitamos hacer `a[2,1]`. Si intentamos acceder a un elemento fuera del rango de la matriz obtendremos un error.

```
>>> a

array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

>>>a[2,1]
8
```

También podemos seleccionar determinadas filas y columnas de una matriz $a[x:y,z:t]$. Con este comando seleccionaremos las filas de índice x hasta y(no incluida) y las columnas z a t(no incluida).

```
>>>a[0:2,1:3]
array([[2, 3],
       [5, 6]])
```

Si uno de estos valores no se incluye se considera como índice 0 si está a la izquierda de ":" o el último índice si está a la derecha.

```
>>>a[:2,1:]
array([[2, 3],
       [5, 6]])
```

Si en vez de poner un rango separado por ":", ponemos solo un [número,] cogerá la columna/fila con ese índice.

```
>>>a[[2],:]
array([7, 8, 9])
```

También podemos coger los elementos que cumplen una condición con $a[\text{condición}]$, esto nos devuelve una matriz de una fila con esos elementos:

```
>>> a[a>5]
array([6, 7, 8, 9])
```

TRAZANDO FUNCIONES

Para trazar funciones labUGR incluye una serie de funciones importadas del módulo Matplotlib.

```
# Esto es un comentario (no se ejecuta)

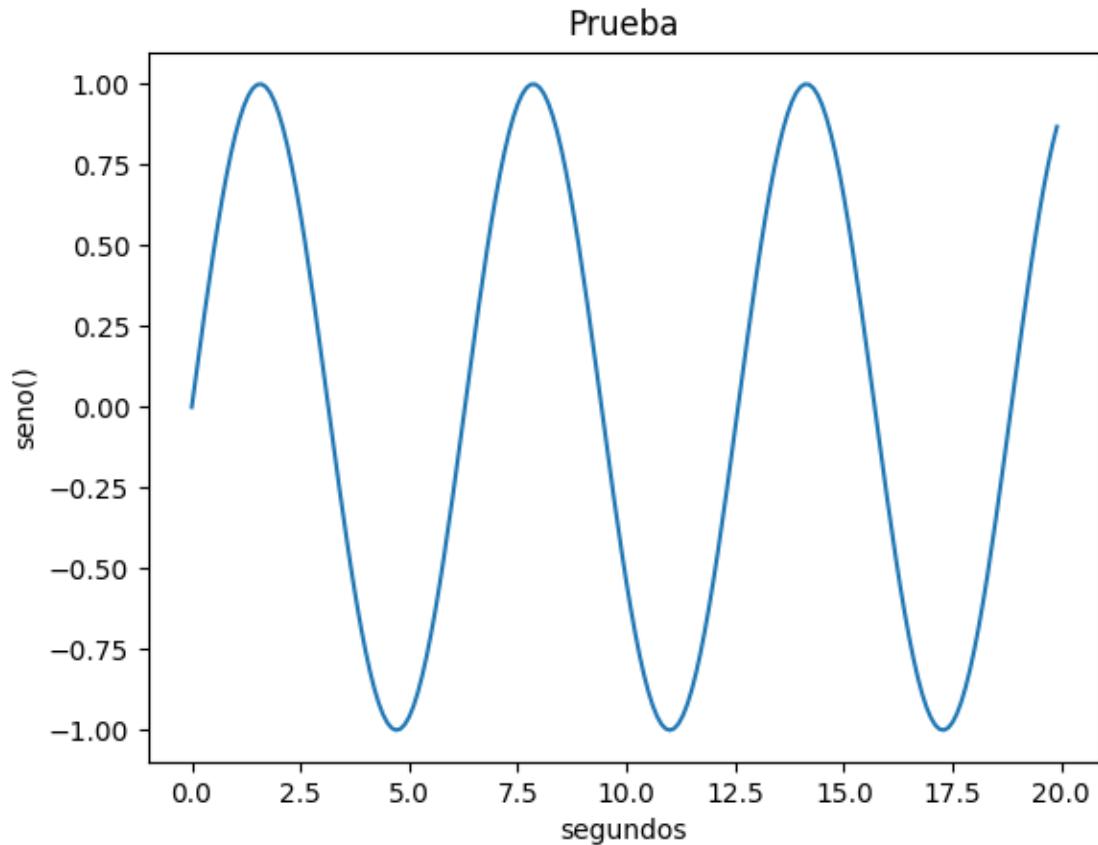
x = arange(0, 20, 0.1)

y = sin(x)

plot(x, y)
ylabel('seno()')
xlabel('segundos')

title('Prueba')

show()
```



Es recomendable poner comentarios en el código para que otras personas lo puedan entender fácilmente. Todo lo que se escriba detrás de % no será ejecutado.

OPERACIONES CON MATRICES Y VECTORES

Los operadores suma “+” y resta “-“tienen un funcionamiento básico, pero hay que asegurarse de que las matrices tienen las mismas dimensiones, sino obtendremos un error. Si sumamos un único elemento a una matriz la suma se hace con todos los elementos.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,3)
```

Los operadores multiplicación “*”, división “/”, potencia “**”, división entera “//” y resto “%” utilizan los valores de la matriz elemento a elemento.

```

>>> a*a
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])

>>>a**2
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])

>>>a%4
array([[1, 2, 3],
       [0, 1, 2],
       [3, 0, 1]])

>>>a/2
array([[ 0.5,  1. ,  1.5],
       [ 2. ,  2.5,  3. ],
       [ 3.5,  4. ,  4.5]])

>>>a//2
array([[0, 1, 1],
       [2, 2, 3],
       [3, 4, 4]], dtype=int32)

```

Para hacer las operaciones matriciales necesitamos utilizar los comandos `dot(a,b)` para la multiplicación y `linalg.matrix_power(x, a)` para la potencia.

```

>>>dot(a,a)
array([[ 30,  36,  42],
       [ 66,  81,  96],
       [102, 126, 150]])

>>>np.linalg.matrix_power(a,2)
array([[ 30,  36,  42],
       [ 66,  81,  96],
       [102, 126, 150]])

```

Para que el valor de la variable se actualice debemos modificar el valor de esta (`a = a/2`). También se puede hacer añadiendo el operador `=` a nuestro operador : (`a/=2`).

Traspuesta

Para hacer la traspuesta utilizamos la función transpose().

```
>>>b=array([(1,2,3),(4,5,6)])  
  
>>>b  
array([[ 1.,  2.,  3.],  
       [ 4.,  5.,  6.]])  
  
>>>b.transpose()  
  
array([[ 1.,  4.],  
       [ 2.,  5.],  
       [ 3.,  6.]])
```

Determinante

Para calcular el determinante de una matriz (si la matriz es cuadrada) utilizamos la función linalg.det(a).

Nota: si pruebas a hacer un determinante de una matriz singular dará como resultado un número muy pequeño diferente de 0. Funciona bien si lo ejecutas dentro de un script

```
>>>c=array([(2,0),(-1,3)])  
  
>>>np.linalg.det(c)  
  
6.0
```

Inversa

Para calcular la inversa utilizamos la función linalg.inv(). La matriz debe ser cuadrada y tener un determinante diferente de 0 (no singular).

Nota: si pruebas a hacer un determinante de una matriz singular dará como resultado un número muy pequeño diferente de 0. Funciona bien si lo ejecutas dentro de un script

→**numpy.linalg.linalg.LinAlgError: Singular matrix**

```
>>>a=array([(2,0,1),(3,0,0),(5,1,1)])  
  
>>>np.linalg.inv(a)  
  
array([[ 0. ,  0.33333333,  0.        ],  
       [-1. , -1. ,  1.        ],  
       [ 1. , -0.66666667,  0.        ]])
```

Operadores relacionales

En Python contamos con los siguientes operadores relacionales:

- < menor que
- > mayor que
- <= menor o igual que
- >= mayor o igual que
- == igual que
- != distinto de

Si una comparación se cumple el resultado es 1 (True), mientras que si no se cumple es 0 (False). En Python, estos operadores se pueden aplicar a multitud de variables sin tener que hacer conversiones ni utilizar otras funciones. Por ejemplo, podemos comparar dos strings para ver si son iguales o comparar los valores de dos matrices (en el caso de las matrices, devuelve una matriz de tipo bool en la que se comparan elemento a elemento):

```
>>>a=array([(1,8,50),(8,8,8),(3,3,3)])
>>>b=array([(2,8,1),(3,8,8),(3,1,3)])
>>>a==b
array([[False,  True, False],
       [False,  True,  True],
       [ True, False,  True]], dtype=bool)

>>>a>b
array([[False, False,  True],
       [ True, False, False],
       [False,  True, False]], dtype=bool)

>>>a>7
array([[False,  True,  True],
       [ True,  True,  True],
       [False, False, False]], dtype=bool)
```

Si queremos utilizar estas condiciones en sentencias for, while o if estas condiciones tienen que devolver un solo valor: True o False. Debemos utilizar el operador array_equal(a,b). También podemos utilizar any() o all() para comprobar si hay algún valor igual o todos los valores son iguales (las matrices tienen que tener las mismas dimensiones).

```
>>>np.array_equal(a,b)
False
>>>(a==b).all()
False
>>>(a==b).any()
True
```

Operadores lógicos

Los operadores lógicos de Python son los siguientes:

- and
- or
- not

Los operadores lógicos se combinan con los relacionales para poder comprobar el cumplimiento de condiciones múltiples. Con “and” ambas condiciones tienen que cumplirse; con “or” solo una de ellas y “not” devuelve True cuando la condición no se cumple. Con esto podemos crear condiciones muy complejas.

```
>>>5>4 and 7!=8 or 6<=3 and not 5==5
True
```

Estos operadores son secuenciales, en el ejemplo anterior ($5>4$ and $7\neq 8$) devuelve True por lo que el resultado total es true ya que el operador or devuelve True si una de las condiciones es True. Es recomendable señalar claramente con paréntesis el orden de las condiciones ya que esto puede variar mucho el resultado:

```
>>>((5>4 and 7!=8) or 6<=3) and (not 5==5)
False
```

En este caso la segunda condición del último and es falsa por lo que el resultado total es False.

BIFURCACIONES Y BUCLES

En Python, para diferenciar el código que está dentro de un bucle (for o while) o una bifurcación (if, elif y else) utilizamos la indentación (separación de 4 espacios o tabulación) en las líneas de código dentro de la bifurcación o bucle. PyCharm automáticamente añade la indentación por lo que el usuario solo tiene que estar pendiente de que el código está dentro del if correcto.

Sentencia if

La sentencia if nos permite ejecutar una parte del código solo si se cumple una condición (o varias). También puede ir acompañada por las sentencias elif y else:

if condicion1:

código a ejecutar si se cumple condicion1

elif condicion2:

código a ejecutar si no se cumple condicion1 y se cumple condicion2

elif condicion3:

código a ejecutar si no se cumple condicion1 ni condicion2 y se cumple condicion3

else:

código a ejecutar si no se cumple ninguna de las condiciones anteriores

Ejemplo:

```
from labugr as *
a = array([(1, 2, 3), (4, 5, 6), (7, 8, 9)])
b = a + 1
if np.array_equal(a, b):
```

```

if 3 <2:

    print('Esto no se muestra')

    print('Las matrices son iguales')

else:

    print('Las matrices son diferentes')

```

Por pantalla se mostrará “Las matrices son diferentes”

Sentencia for

La sentencia for repite un bucle un número predeterminado de veces. La sintaxis básica es la siguiente:

```

for i in range(x, y, z):
    código

    print(i)

```

‘X’ representa el primer número por el que se empieza a contar, ‘y’ el último número (no se incluye) y ‘z’ representa la distancia entre los números. El valor de z puede ser negativo, pero tendremos que asegurarnos de que y<x. Por ejemplo, si x=4, y=10 y z=2 el “código” se ejecutará tres veces (siendo i=4 en la primera iteración, i=6 en la segunda e i=8 en la tercera).

```

>>>a=array([(1,2),(3,4)])

>>>dimensiones=shape[:]

>>>for i in range(0,dimensiones[0]):

...     for j in range(0,dimensiones[1]):

...         print ('a[%i,%i]=%i'%(i,j,a[i,j]))

...

a[0,0]=1

a[0,1]=2

a[1,0]=3

a[1,1]=4

```

En el ejemplo anterior iteramos a través de dos bucles anidados todos los elementos de una matriz. Pero una gran ventaja de Python es que nos permite iterar a través de diferentes tipos de elementos

directamente en el bucle for. Si por ejemplo, en vez de range utilizamos un string, el valor de I será una letra del string cada iteración.

```
>>>for i in 'Hola':
...     print (i)
...
H
o
l
a
```

Lo que nos interesa a nosotros son las matrices, y en Python esto funciona de manera análoga. Si queremos acceder a los elementos de la matriz uno por uno podemos utilizar la función nditer(matriz, op_flags=['readwrite']):

```
>>>a=array([(1,2,3),(4,5,6),(7,8,9)])
>>>for valor in nditer(a, op_flags=['readwrite']):
...     valor[...] = 2 * valor
>>>a
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

No es necesario poner op_flags=['readwrite'] pero si no se pone la iteración será solo de lectura y no se podrán modificar los valores. Utilizando [...] podemos modificar el valor del elemento.

Sentencia while

El código se ejecuta mientras la condición se cumpla (True). Su sintaxis es la siguiente:

while condicion:

 código

Ejemplo:

```
>>>a=0
```

```
>>>while a<3:
...     print('a es menor que 3. Su valor es %d'%a)
...     a+=1
...
a es menor que 3. Su valor es 0
a es menor que 3. Su valor es 1
a es menor que 3. Su valor es 2
```

Definición de funciones

Si hay parte del código que tenemos que repetir muchas veces a lo largo del código es buena práctica crear una función que contiene el código y llamarla cada vez que necesitamos el código. Este es un ejemplo de la estructura de una función, al igual que los bucles y bifurcaciones, el código dentro de una función tiene que estar indentado (4 espacios):

```
x = 5
z = 1

def funcion_test(a, b):
    c = (a + b + z) / 3
    d = (a + b + z) // 3
    a += 1
    print(x)
    return c, d, z

parametro1 = 8
f = funcion_test(parametro1, 20)

print(f)
print("%.2f" % f[0])
print(f[1])
print(parametro1)
```

Lo primero que hay que tener en cuenta es que las funciones son “independientes del resto del código”:

- Variables globales: estas son las variables creadas fuera de una función, en el ejemplo x y z son variables globales. Las funciones pueden acceder a su valor y utilizarlo dentro de la función (se crea una copia de la variable para su uso dentro de la función) pero las modificaciones que se hagan a esa variable no se verán aplicadas a la variable global (a no ser que la variable sea una referencia, puntero, como los

arrays). Se recomienda no utilizar variables globales dentro de funciones: puede causar errores y limita la reusabilidad de la función.

- Parámetros La manera correcta de utilizar variables globales dentro de una función es pasarlo como parámetros. En el ejemplo a y b son parámetros. Los cambios que se hagan dentro de la función a estos parámetros no se aplicarán a las variables globales. Podemos ver que el valor de parametro1 no cambia fuera de la función y sigue siendo 8.
- Salida de la función: la función puede devolver valores. Si solo devuelve un valor no necesitamos utilizar corchetes. EN el ejemplo, c,d y 5 son la salida de la función.
- Si guardamos las funciones en un archivo .py (como funciones.py) en la misma carpeta que main.py podemos acceder a ellas dentro de main.py si las importamos.

```
from funciones import *
```

EJERCICIOS

1. Crear un script en python que cree las siguientes matrices:

$$A = \begin{pmatrix} 1 & 3 & 5 & 8 \\ 2 & 6 & 5 & 3 \\ 4 & 1 & 9 & 7 \\ 1 & 8 & 0 & 2 \end{pmatrix}; B = \begin{pmatrix} 1 & 9 & 5 & 8 \\ 12 & 5 & 5 & 9 \\ 4 & 2 & 9 & 74 \\ 0 & 6 & 0 & 3 \end{pmatrix}$$

El script debe mostrarlas en pantalla después de crearlas. Seleccionar la segunda fila de B, la cuarta columna de B y las submatriz formada por las dos primeras filas y las dos primeras columnas de la matriz B y muéstrela por pantalla. Realizar los siguientes cálculos básicos $3*A$, $A-7$, $A*B^T$, A^{-1} , B^{-1}

2. Obtener utilizando la función `linspace`, un vector de 20 elementos que recorra el intervalo $[0,\pi]$ (Nota, $\pi=np.pi$). Generar utilizando la función `arange()`, un vector de 1000 puntos que recorra el intervalo $[0,10]$.
3. Generar un vector de 250 puntos que sea una secuencia aleatoria uniforme (`random.rand`) de valores entre 0 y 1. Encontrar: • Aquellas componentes del vector cuyo valor sea mayor que 0.9. • Aquellas componentes del vector cuyo valor sea menor o igual que 0.15.
4. Implementar una función que calcule los cuadrados de los N primeros números naturales de 2 formas distintas: • Utilizando un bucle `for` • Calculando el cuadrado del vector correspondiente (operación componente a componente). Calcular, con la ayuda de la función `timeit.timeit()` (`import timeit`) el tiempo que se tarda en realizar el cálculo con cada uno de los dos métodos. La función a implementar debe tener como variable de entrada el número N, y como salida el tiempo que se tarda con cada uno de los dos métodos (`t1, t2`).

NOTA: Definir dos funciones dentro de la función principal (una para cada método) y luego utilizar `timeit.timeit(funcion1)`. Los resultados dependen de la máquina donde se ejecute y hay que tener en cuenta que `timeit` repite la función *1.000.000 de veces por defecto (si va muy lento reducir el número →`timeit.timeit(funcion1, number=100000)`)*.

10.2 Sistemas Lineales

10.2.1 Práctica 1: Representación de señales

Ejemplo 1.1

Considere una señal $x(t)$ como la mostrada en la Figura 1.

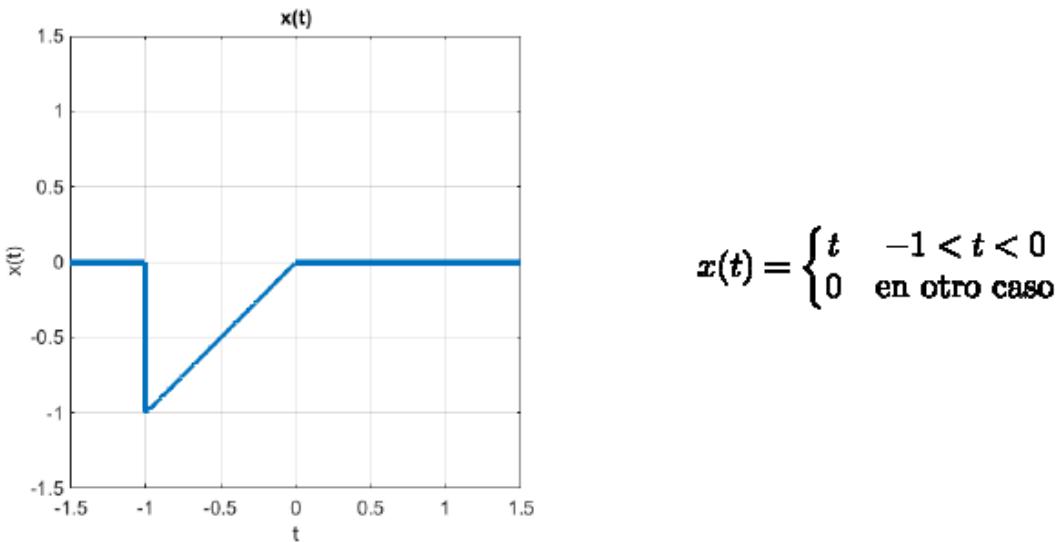


Figura 1. Señal $x(t)$

1.1.1 Escriba una función de Python de nombre `senal_x` que admita como entrada valores de la variable independiente t y como salida obtenga los valores de la señal $x(t)$ anterior para cada instante de tiempo evaluado. Recuerde que la primera línea de la función debe ser del siguiente tipo: `def senal_x(t):`

Como la función da distintos valores de salida para distintas condiciones de la variable independiente, es necesario implementar en la función `senal_x` lo siguiente

```
def senal_x():
    resultado = zeros(0)
    for i in range(0, t.size):
        if # Poner aquí la condición para que se cumpla la línea
        siguiente
            resultado.append(t[i])
        else:
            resultado.append(0)
    return resultado
```

En realidad, realizando el cálculo con el esquema anterior, el tamaño de la variable y va creciendo en cada iteración. Desde el punto de vista de la eficiencia y rapidez del código, sería más conveniente que el vector y tuviera definido su tamaño de antemano (antes de entrar en la línea de código con la condición `if`). Por lo tanto, análogamente, podríamos reescribir la función `senal_x()` del siguiente modo:

```
def senal_x(t):
    tamano = t.size
    resultado = zeros(tamano) # Definimos un vector de zeros con el
    mismo tamaño que t
    for i in range(0, tamano):
        if # Poner aquí la condición para que se cumpla la linea
        siguiente
            resultado[i] = t[i]
    return resultado
```

1.1.2 Escriba un script desde donde se haga una llamada a la función `senal_x` y represente sus valores entre $t = -3$ y $t=3$. Compruebe que se obtiene la señal de la Figura 1. Como Python trabaja con variables discretas, por ejemplo, puede crear el vector de tiempo del siguiente modo:

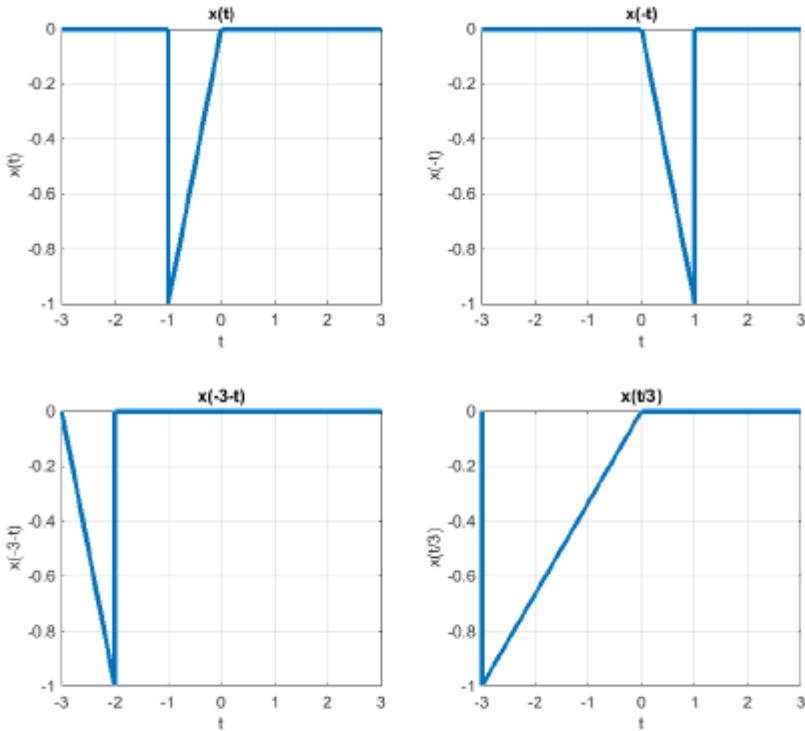
```
FS = 1e4 #Frecuencia de Muestreo
dt = 1/FS #Periodo de muestreo (tamaño de paso del vector discreto de
tiempos)
t = arange(-3, 3, dt)
```

Donde F_s es el número de muestras por segundo. Aumentando el valor de la frecuencia demuestreo F_s hacemos más denso el vector t de tiempos.

1.1.3 Represente, también para $-3 < t < 3$ las siguientes señales:

- $x(-t)$
- $x(-3-t)$
- $x(t/3)$

Compruebe que obtiene el resultado de la Figura 2.



1.1.4 Descomponga $x(t)$ en sus partes par e impar. Represente $x(t)$, su parte par y su parte impar.

Represente también la suma de las partes par e impar. Compruebe que se obtiene un resultado como el de la Figura 3.

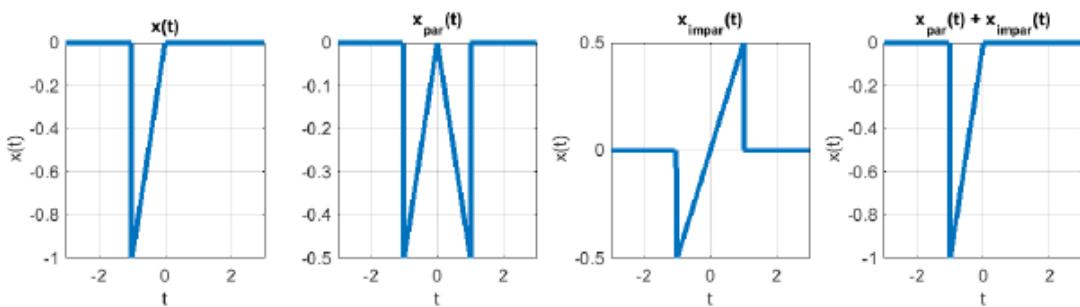


Figura 3. Representación gráfica de la señal $x(t)$ y la señal obtenida mediante la descomposición en parte par e impar y la suma de ambas.

Ejemplo 1.2

1.2.1 Considere las siguientes señales periódicas

$$x_1(t) = 0.5 \sin(t)$$

$$x_2(t) = 1.5 \cos\left(\frac{\pi}{2}t + 2\right)$$

$$x_3(t) = 2 \cos\left(\frac{3\pi}{2}t + 3\right)$$

$$x_4(t) = 3 \sin(\pi t) + 1.5 \sin\left(\frac{1}{4}\pi t\right)$$

1. ¿Cuál es el período de las tres primeras señales?
2. ¿Es periódica la suma de x_1 y x_2 ?
3. ¿Es periódica la suma de x_2 y x_3 ?
4. ¿Es periódica la suma de x_2 y x_4 ?
5. Represente las señales en cada uno de los apartados (señales x_1 , x_2 , x_3 , x_4 , x_1+x_2 , x_2+x_3 y x_2+x_4).

Para trazar las señales anteriores, se puede usar el siguiente código de Python.

```
# Apartado 1.2.1
Fs = 1000
dt = 1./Fs
t = arange(0, 20, dt)

figure(1)

# Señal 1
x1 = 0.5 * sin(t)
f1 = 1/(2*pi)
T1 = 1/f1

subplot(221)
plot(t, x1)
plot([T1, T1], [-2, 2], 'r-', lw=2)
grid()
```

```

xlabel('t')
ylabel('x_1(t)')
axis([0, 15, -2, 2])
title('x_1(t), T_1 = ' + str(round(T1, 4)), fontweight='bold')

# Señal 2
x2 = 1.5*cos(pi/2*t + 2)
f2 = 1/4.
T2 = 1/f2

subplot(222)
plot(t, x2)
plot([T2, T2], [-2, 2], 'r-', lw=2)
grid()
xlabel('t')
ylabel('x_2(t)')
axis([0, 15, -2, 2])
title('x_2(t), T_2 = ' + str(round(T2, 4)), fontweight='bold')

# Señal 3
x3 = 2*cos(3*pi/2*t + 2)
f3 = 3/4.
T3 = 1/f3

subplot(223)
plot(t, x3)
plot([T3, T3], [-3, 3], 'r-', lw=2)
grid()
xlabel('t')
ylabel('x_3(t)')
axis([0, 15, -3, 3])
title('x_3(t), T_3 = ' + str(round(T3, 4)), fontweight='bold')

# Señal 4
x4 = 3*sin(pi*t) + 1.5*sin(pi/4*t)
f4 = 1/8.
T4 = 1/f4

subplot(224)
plot(t, x4)
plot([T4, T4], [-5, 5], 'r-', lw=2)
grid()
xlabel('t')

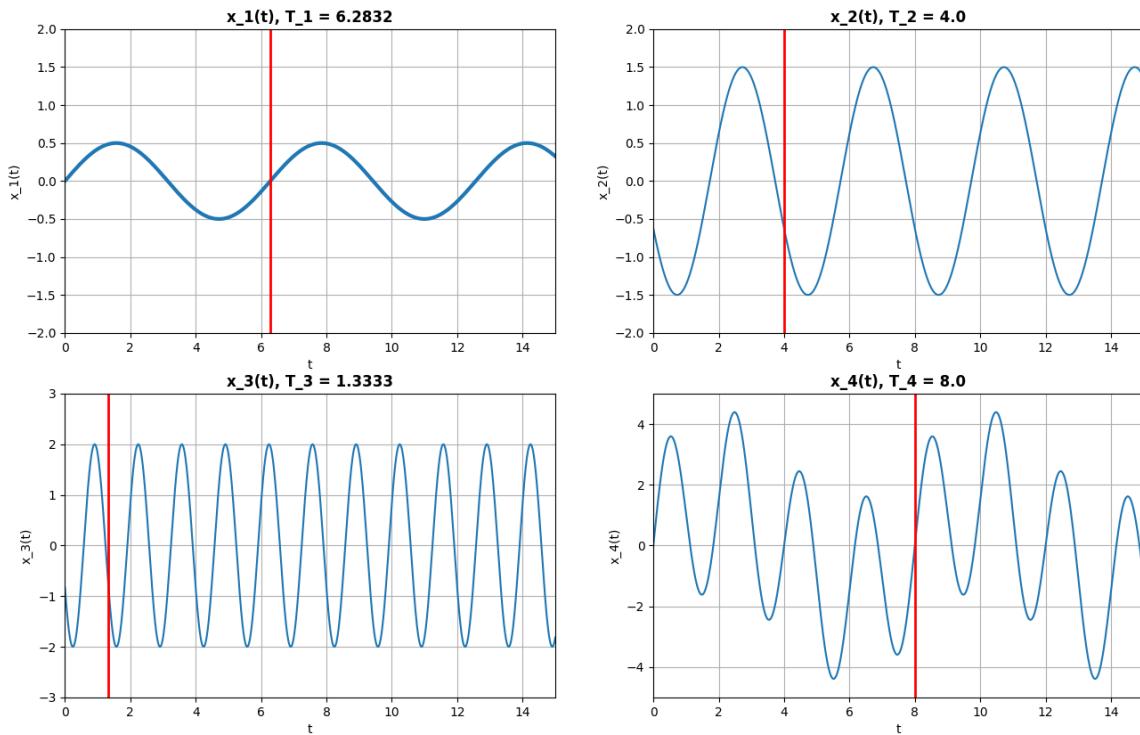
```

```

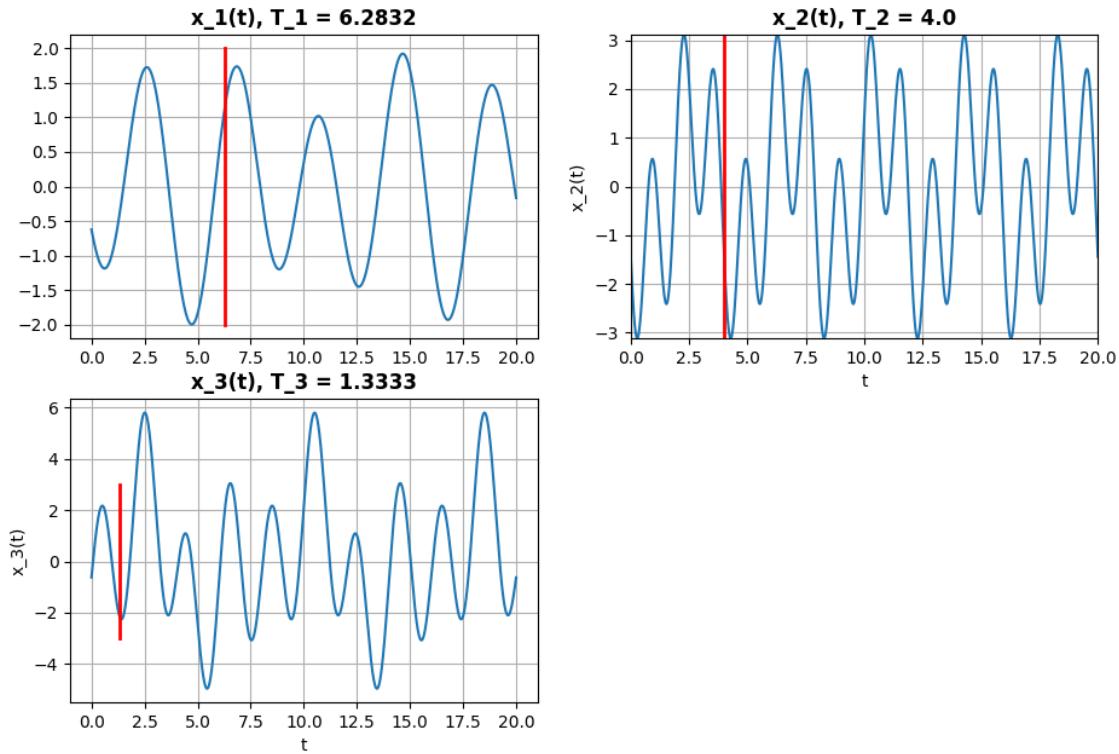
ylabel('x_4(t)')
axis([0, 15, -5, 5])
title('x_4(t), T_4 = ' + str(round(T4, 4)), fontweight='bold')
plt.tight_layout()
show()

```

Observe que mediante el código anterior se representa mediante una línea roja vertical el tiempo donde la señal vuelve a repetir valores, es decir, el periodo de la señal (Figura 4).



1.2.2 Análogamente, a lo que se ha realizado en el apartado realice una representación gráfica de las señales x_1+x_2 , x_2+x_3 y x_2+x_4 . En el caso en que la señal suma sea periódica, calcule su periodo y localícelo en la gráfica, tal y como se ha hecho en el apartado anterior.



Ejemplo 1.3

Use Python para simular las siguientes señales

1. La señal escalón unitario $u(t)$
2. El impulso unitario $\delta(t)$
3. La señal rampa $r(t)$
4. Señal rectangular.

1.3.1 Tal y como hizo cuando construyó la función `signal_x` en el apartado 1.1.1., construya ahora una función de Python de nombre `ustep` donde se implemente la señal escalón unitario. Es decir, la señal que es cero cuando la variable independiente t es negativa y 1 cuando t es positiva. Recuerde que la primera línea de código de la función debe ser como sigue:

```
def ustep(t):
```

Represente esta señal usando la función `plot(t,y)`.

1.3.2 Construya también una función de nombre `delta.m` que implemente la señal impulso.

Esta señal es igual a 0 para cualquier valor de t excepto para $t=0$ que vale 1.

```
def delta(t):
```

Represente esta señal usando la función `plot(t,y)` y también la función `stem(t,x)`. Compare ambos resultados.

1.3.3 Construya una función de nombre rampa que implemente la señal rampa. Esta señal es igual a 0 para cualquier valor de t negativo e igual a t para $t > 0$.

```
def rampa(t):
```

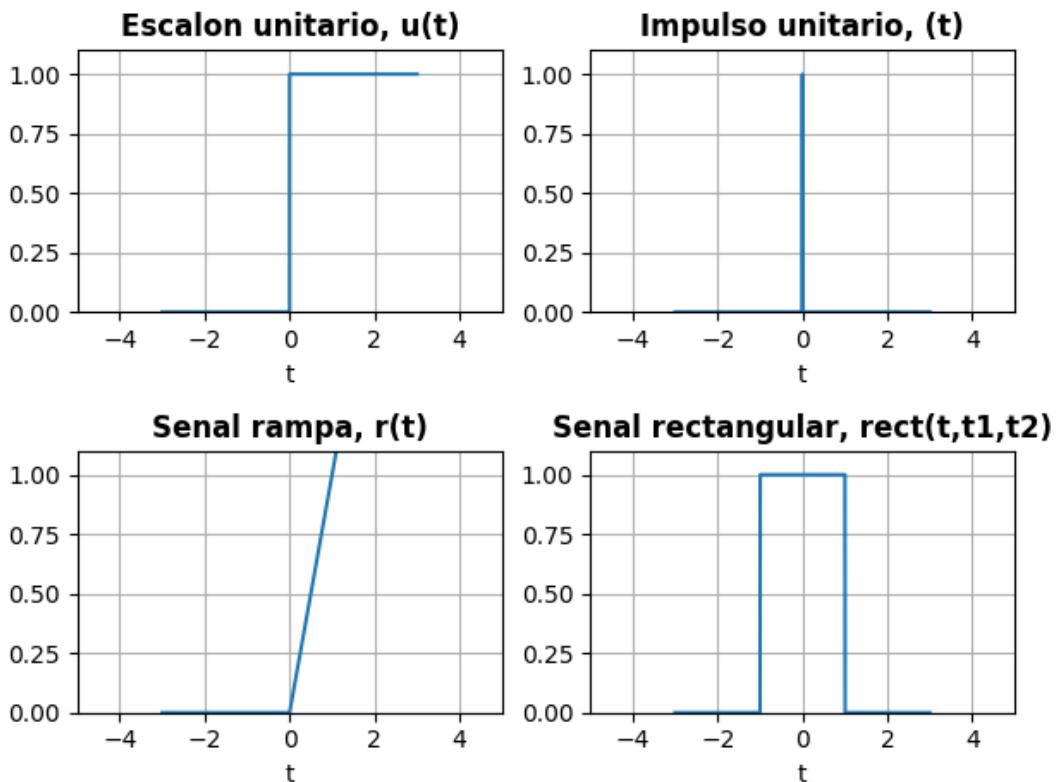
Represente esta señal usando la función plot(t,y).

1.3.4 Tal y como se vio en teoría, demuestre que una señal rectangular $\text{rect}(t, t_1, t_2)$, es decir, una señal que es cero para cualquier t excepto en el intervalo $[t_1, t_2]$ que vale 1; puede construirse mediante la diferencia entre dos señales escalón. Elija los valores $t_1 = -1$ y $t_2 = 1$ y represente la señal $y = \text{ustep}(t-t_1) - \text{ustep}(t-t_2)$.

```
# Apartado 1.3.
Fs = 1000. # Frecuencia de muestreo
dt = 1/Fs # Periodo de muestreo
t = arange(-3, 3, dt) # Vector de tiempos
t1, t2 = -1, 1
y = ustep(t-t1) - ustep(t-t2)
```

Compruebe que obtiene los resultados de la Figura 5.

Figura 5. Representación gráfica de las señales elementales escalón, impulso, rampa y señal rectangular.



10.2.2 Práctica 2: Números complejos

En Python, el numero imaginario $i = \sqrt{-1}$ se representa con el término ‘j’ que puede ser utilizado para generar números complejos. Por ejemplo, el numero complejo con parte real igual a 3 y parte imaginaria igual a 8 se representaría de la siguiente manera:

```
>>> a = 3 + 8j
>>> a
(3+8j)
>>> type(a)
<class 'complex'>
```

Existen una serie de métodos incluidos en la librería labUGR para tratar con números complejos:

- ***imag()* o *.imag*:** para obtener la parte imaginaria de un número complejo:

```
>>> imag(a)
8.0
>>> a.imag
8.0
```

- ***real()* o *.real*:** para obtener la parte real de un número complejo.

```
>>> real(a)
3.0
>>> a.real
3.0
```

- ***conj()*:** para obtener el conjugado de un número complejo.

```
>>> conj(a)
(3-8j)
```

- ***abs()*:** para obtener el módulo de un número complejo.

```
>>> abs(a)
8.54400374531753
```

- ***angle()*:** para obtener la fase de un numero complejo en radianes. Para grados decimales utilizar deg=True.

```
>>> angle(a)
1.2120256565243244
>>> angle(a, deg=True)
69.443954780416533
```

- ***deg2rad()* y *rad2deg()*:** para convertir grados decimales a radianes y viceversa.

```
>>> deg2rad(90)
1.5707963267948966
>>> rad2deg(pi/2)
90.0
```

Todas estas funciones se pueden aplicar tanto a números individuales como a matrices de números complejos (siendo el resultado una matriz con las mismas dimensiones). Por ejemplo:

```
>>> b = array([(1+2j, 3-0.5j), (2j, 3)])
>>> b
array([[ 1.+2.j,  3.-0.5j],
       [ 0.+2.j,  3.+0.j ]])
>>> real(b)
array([[ 1.,  3.],
       [ 0.,  3.]])
>>> imag(b)
array([[ 2., -0.5],
       [ 2.,  0. ]])
>>> conj(b)
array([[ 1.-2.j,  3.+0.5j],
       [ 0.-2.j,  3.-0.j ]])
>>> abs(b)
array([[ 2.23606798,  3.04138127],
       [ 2.          ,  3.          ]])
>>> np.angle(b, deg=True)
array([[ 63.43494882, -9.46232221],
       [ 90.          ,  0.          ]])
```

Ejemplo 2.1

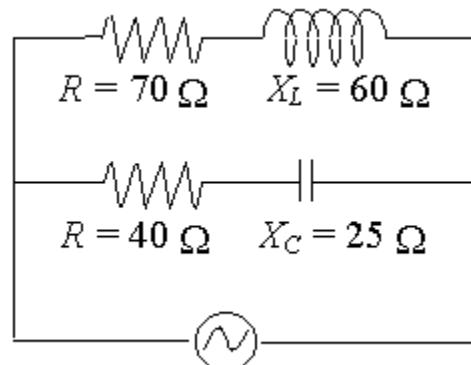
Una de las aplicaciones de los números complejos en el análisis de circuitos es el cálculo de la impedancia combinada de un conjunto de elementos. La impedancia total del circuito es la siguiente:

$$Z_1 = \frac{(70 + 60j)(40 - 25j)}{(70 + 60j) + (40 - 25j)}$$

Código en Python:

```
from labugr import *

#Generando la ecuacion de la impedancia combinada
z1 = ((70+60j)*(40-25j))/((70+60j)+(40-25j))
#Obteniendo el modulo
modulo = abs(z1)
#Obteniendo la fase
fase = angle(z1)
#Obteniendo la parte real y la parte imaginaria
preal = z1.real
pimag = z1.imag
#Obteniendo la representacion en forma polar
formaPolar = (abs(x), angle(x))
```



El circuito anterior está alimentado con una fuente de voltaje de 12V con una fase de 15 grados decimales ($12\angle 15^\circ$). Es importante tener en cuenta que Python trabaja con radianes por defecto y no en grados decimales. Con la impedancia y el voltaje de entrada podemos encontrar la corriente que circula por la fuente:

```
#Definiendo el voltaje
faseDec = deg2rad(15)
v1 = 12*cos(faseDec) + 12j*sin(faseDec)

#Obteniendo la corriente
i1 = v1/z1
print('Parte real: {} \nParte imaginaria: {} \nModulo: {} \nFase: {}'.
      format(i1.real,
             i1.imag, abs(i1), np.angle(i1)))
```

Anote los valores del módulo, la fase, la parte real e imaginaria de la impedancia y de la corriente.

Ejercicio 2.1

Encuentre, de manera análoga, el módulo, la fase, la parte real e imaginaria de las siguientes impedancias complejas:

$$Z_2 = \frac{100-30j}{40-25j} + 3 + 3j - \frac{17}{14+13j}$$

$$Z_3 = \frac{3j}{5-22j} + \frac{18\angle 30^\circ}{7-5j} - 24\angle 0.25$$

Ejemplo 2.2

La respuesta en frecuencia de un sistema viene dada por la siguiente ecuación:

$$H(\omega) = \frac{600\pi}{j\omega + 600\pi}$$

Si creamos un vector de frecuencias con la función linspace podemos obtener la respuesta en frecuencia del sistema para los diferentes valores de la frecuencia, así como la parte real, la parte imaginaria, el módulo y la fase. Para convertir el módulo en dB basta realizar $20\log_{10}(\text{módulo})$.

```
#Vector de frecuencias de 200 elementos entre 0 y 2000 Hz
f = linspace(0, 2000, 200)
#Ecuacion de la respuesta
H = (2*pi*300) / (2j*pi*f + 2*pi*300)

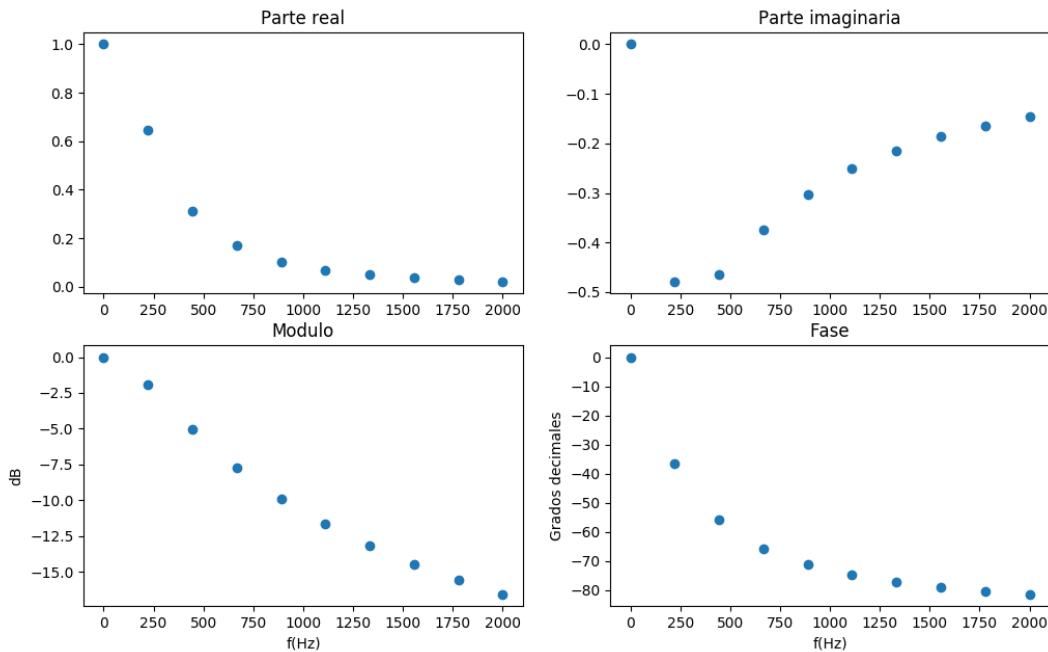
figure()
#Parte real
subplot(2, 2, 1)
title('Parte real')
plot(f, H.real, 'o')
#Parte imaginaria
subplot(2, 2, 2)
```

```

title('Parte imaginaria')
plot(f, H.imag, 'o')
#Modulo
subplot(2, 2, 3)
title('Modulo')
plot(f, 20*log10(abs(H)), 'o')
xlabel('f(Hz)')
ylabel('dB')
#Fase
subplot(2, 2, 4)
title('Fase')
plot(f, angle(H, deg=True), 'o')
xlabel('f(Hz)')
ylabel('Grados decimales')

show()

```



Como podemos observar, este sistema es un filtro paso baja con una frecuencia de corte alrededor de los 300Hz.

Ejercicio 2.2

De manera análoga al ejemplo anterior, obtenga el módulo, la fase, la parte imaginaria y la parte real de los siguientes sistemas. Anote el valor de la respuesta en frecuencias para las frecuencias 0, 200, 400, 500, 600, 1000 y 2000 Hz y razoné que tipo de filtros podrían ser:

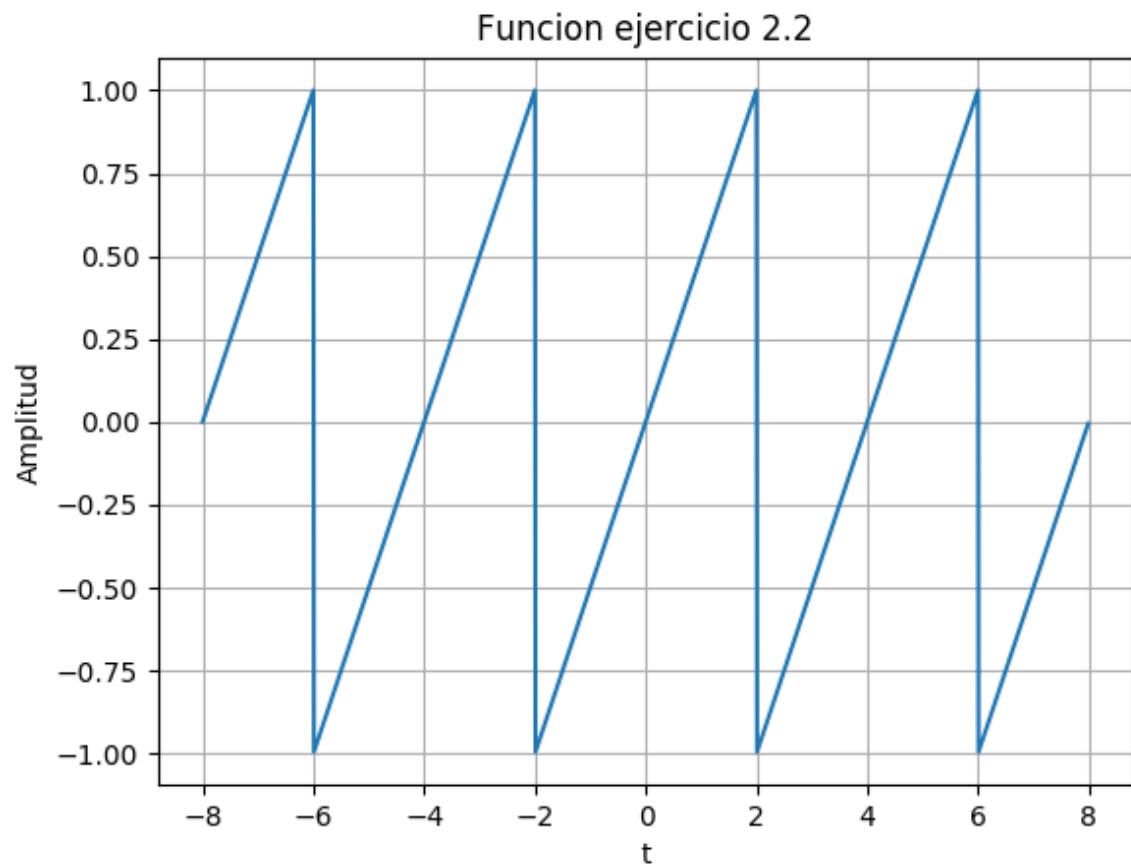
$$H_2 = \frac{j\omega}{j\omega + 800\pi}$$

$$H_3 = \frac{j\omega}{\frac{\omega^2}{1000\pi} + j\omega + 1000\pi}$$

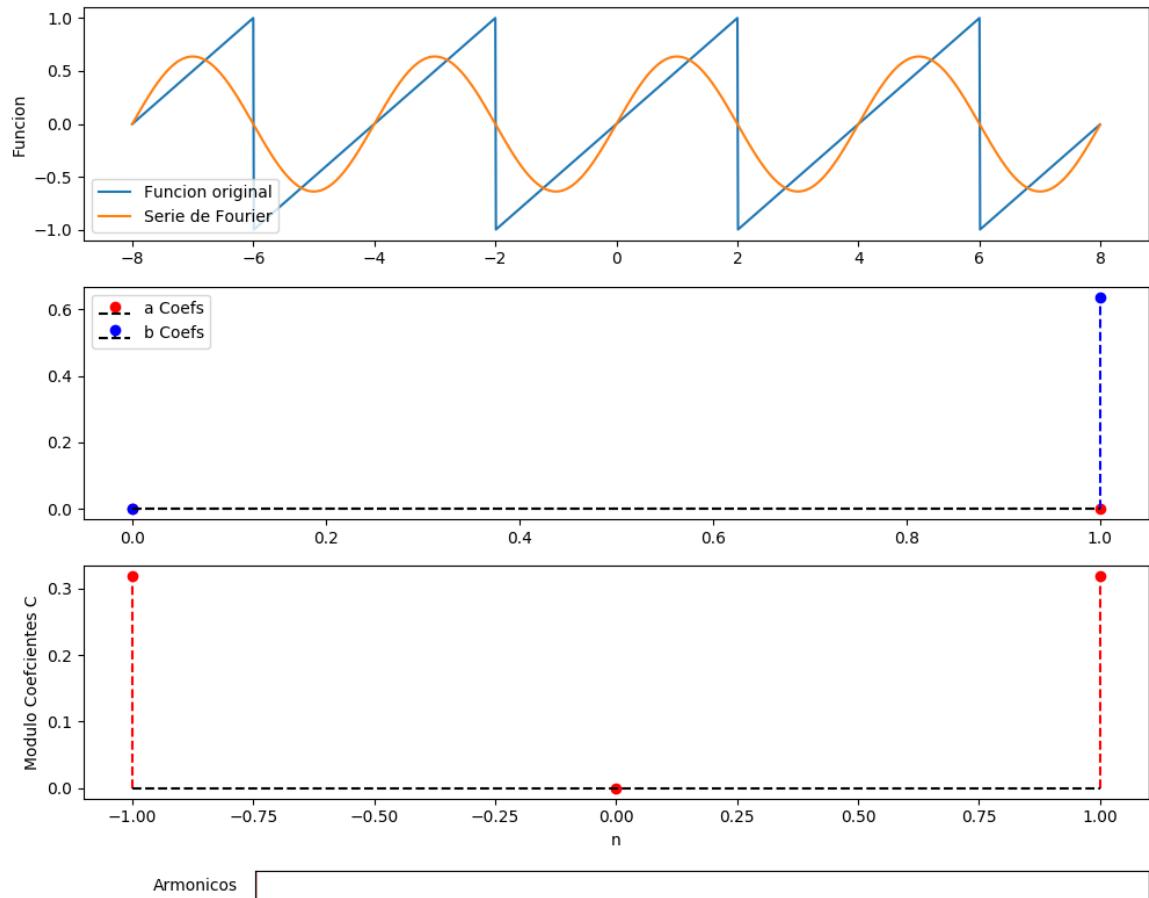
10.2.3 Práctica 3: Desarrollo en serie de Fourier

Desarrollo en serie de Fourier:

Para la señal periódica $f(t)$ de la figura (amplitud 2 y periodo 4), obtenga los coeficientes del desarrollo exponencial de Fourier y posteriormente exprese la misma señal en términos de un desarrollo en funciones seno y coseno.



Descargue el archivo `practica3.py` y ejecútelo.



En la primera grafica podemos observar la función original en azul y la aproximación por desarrollo en serie de Fourier en naranja. El desarrollo en serie de Fourier de una señal periódica con frecuencia $f_0=1/T$, siendo T el periodo de la función, se calcula con la siguiente función.

$$y(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos(2\pi n f_0 t) + b_n \sin(2\pi n f_0 t)]$$

Función para obtener el desarrollo en serie de Fourier de x:

```
#Devuelve 3 valores: los coeficientes de a, de b y el desarrollo en serie

def serieFourier(x,T,n,t):
    resultado = zeros(len(t))

    #Calculamos los coeficientes a y b
    an = aCof(x,T,n)
```

```

ab = bCof(x, T, n)

#Para cada elemento del vector de tiempos calcular la suma de los
coeficientes

for i in range(0, len(t)):

    resultado[i] += an[0] / 2

    for i2 in range(1,n+1):

        resultado[i] += an[i2]*cos(2*pi*i2/T*t[i]) +
                        ab[i2]*sin(2*pi*i2/T*t[i])

return (resultado,an,ab)

```

En la segunda grafica se muestran los coeficientes a y b del desarrollo. Estos se calculan de la siguiente manera:

$$a_n = \frac{2}{T_0} \int_{-T_0/2}^{T_0/2} y(t) \cos(2\pi n f_0 t) dt, \quad n = 0, 1, 2, \dots$$

$$b_n = \frac{2}{T_0} \int_{-T_0/2}^{T_0/2} y(t) \sin(2\pi n f_0 t) dt, \quad n = 1, 2, \dots$$

$$a_0 = \frac{2}{T_0} \int_{-T_0/2}^{T_0/2} y(t) dt$$

Función para obtener los primeros $n+1$ coeficientes a de x:

```

def aCof(y,T0,n):

    resultado=zeros(n+1)

    #Valor de a0

    resultado[0]=(quad(x,-T0/2, T0/2)[0])*(2/T0)

    #Definimos la funcion que deseamos integrar

    def z(t):

        return y(t) * cos(2 * pi * i * (1 / T0) * t)

    #Calculo del resto de coeficientes

    for i in range(1,n+1):

        resultado[i]=(quad(z,-T0/2, T0/2)[0])*(2/T0)

    return resultado

```

La función que definimos para los coeficientes de b es análoga:

```
#Funcion para obtener los coeficientes b de x

def bCof(y,T0,n):
    resultado = zeros(n+1)
    #Valor de b0 es 0
    # Definimos la funcion que deseamos integrar
    def z(t):
        return y(t) * sin(2 * pi * i * (1 / T0) * t)
    # Calculo del resto de coeficientes
    for i in range(1,n+1):
        resultado[i]=(quad(z,-T0/2, T0/2)[0])*(2/T0)
    return resultado
```

En la tercera grafica se muestran el módulo de los coeficientes de la serie compleja de Fourier. Estos coeficientes se calculan de la siguiente manera:

$$\alpha_n = \frac{1}{T_0} \int_{-T_0/2}^{T_0/2} y(t) e^{-j2\pi n f_0 t} dt$$

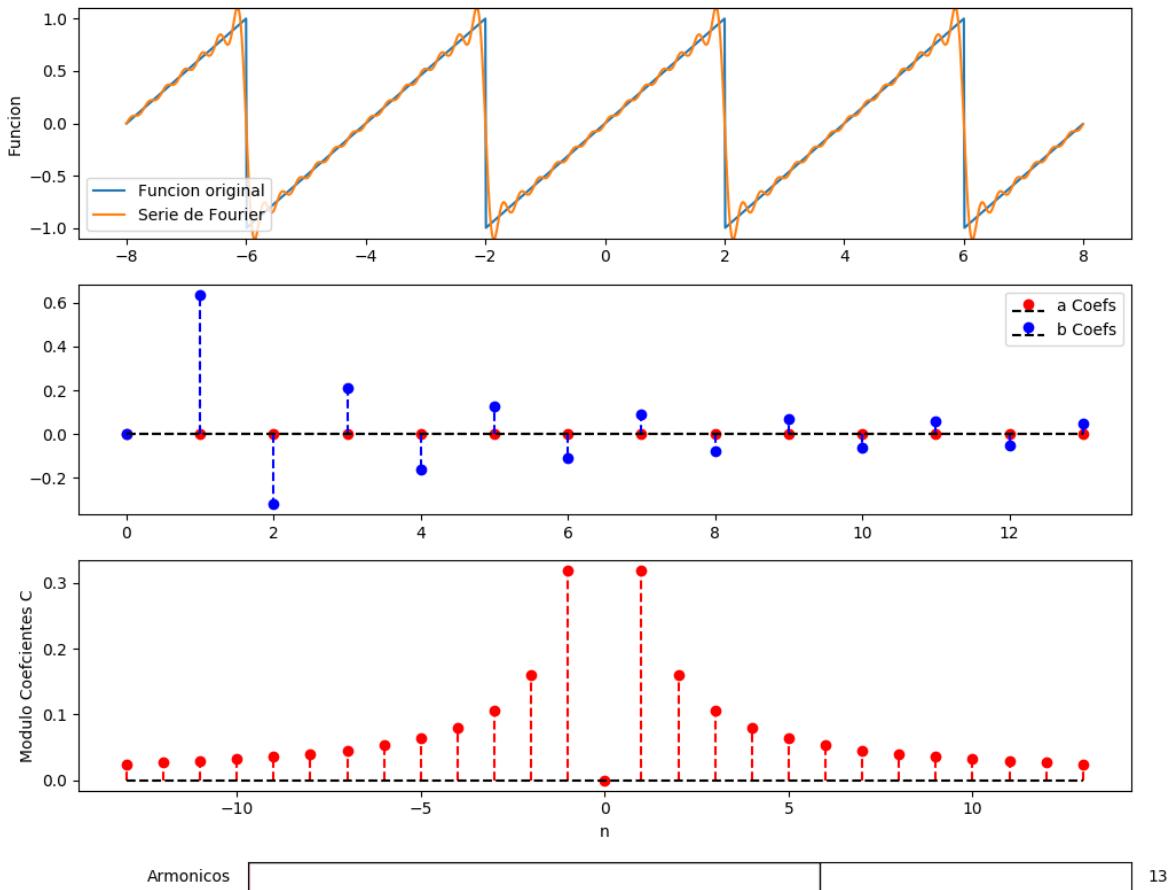
Aunque, como ya hemos calculado antes los coeficientes a y b, podemos obtener los coeficientes α_n a partir de a_n y b_n :

$$\alpha_0 = \frac{1}{2} a_0 \quad \alpha_n = \frac{1}{2}(a_n - jb_n) \quad \alpha_n = \frac{1}{2}(a_n + jb_n)$$

Función para obtener los coeficientes complejos de x a partir de los coeficientes a y b:

```
def cCoef(a,b):
    #Positive part
    c=(a-1j*b)/2
    #Adding the negative part
    for i in range(1,len(a)):
        c=np.insert(c,0,(a[i]+1j*b[i])/2)
    return c
```

En la parte inferior de la figura encontramos un elemento para seleccionar el número de armónicos que queremos representar en las gráficas.

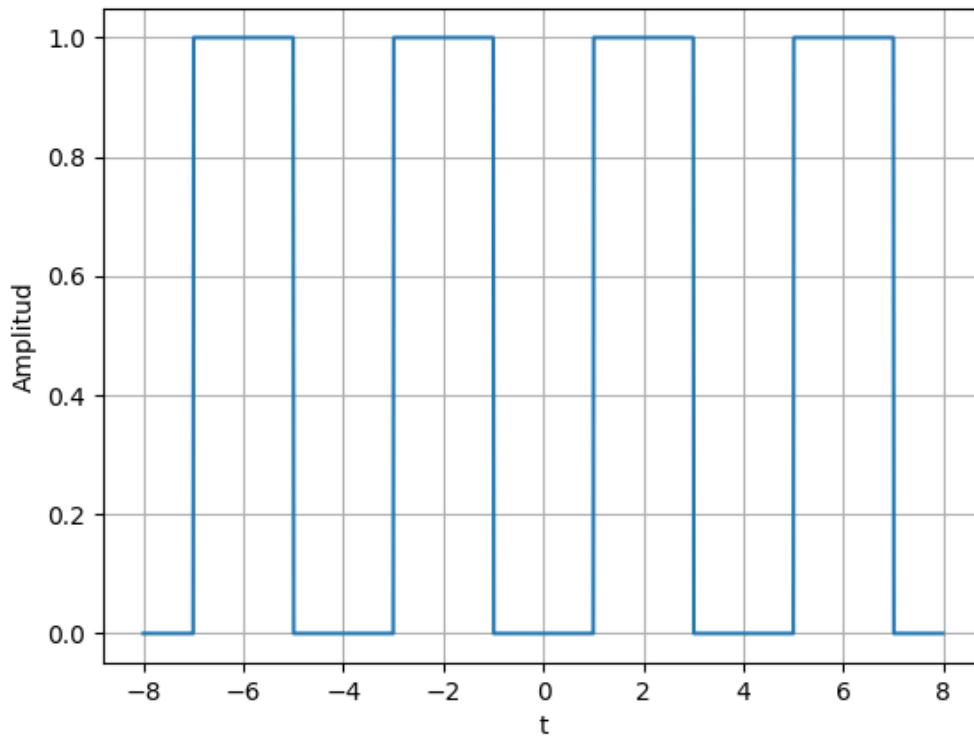


13

Ejercicio 3.1

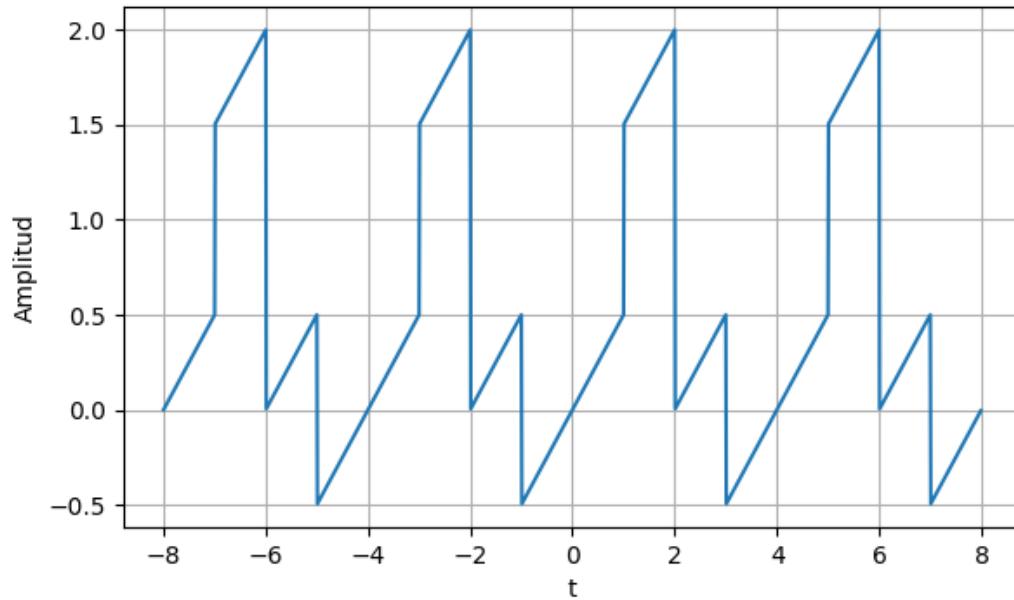
Repita el ejemplo anterior para las señales de las siguientes figuras:

Funcion ejercicio 2.3.1



Tip: Para generar la función podéis utilizar la función `square()` de manera análoga a como creamos la primera señal. El periodo de la señal es 4 y la amplitud 1.

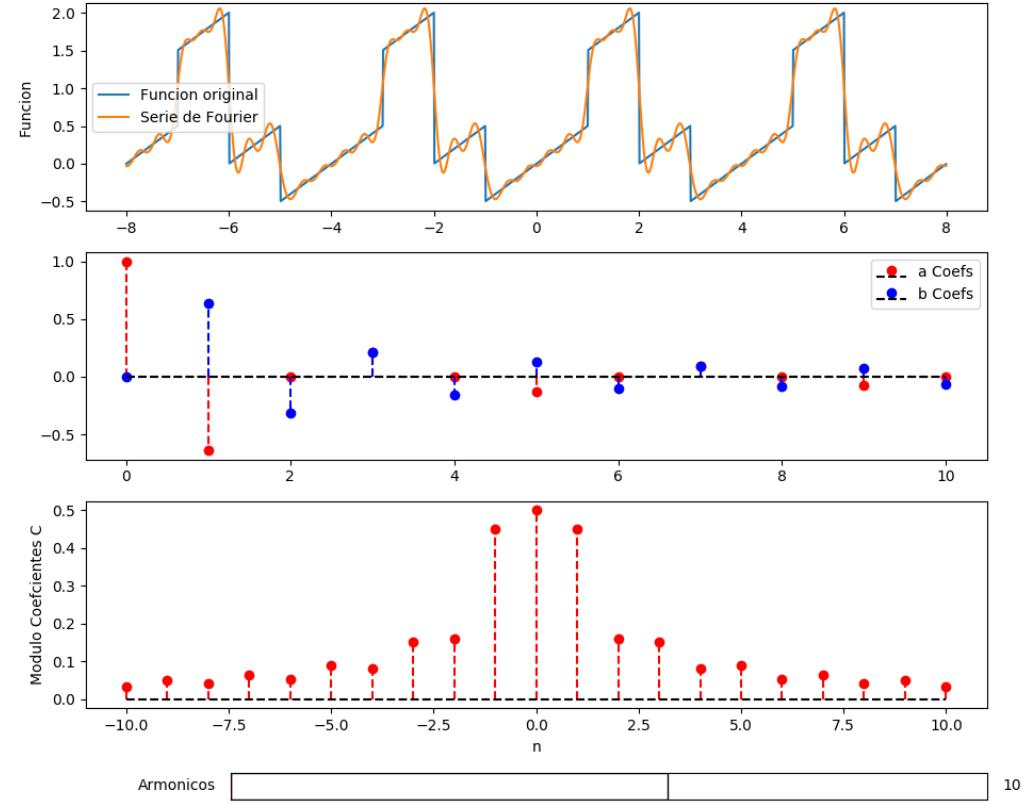
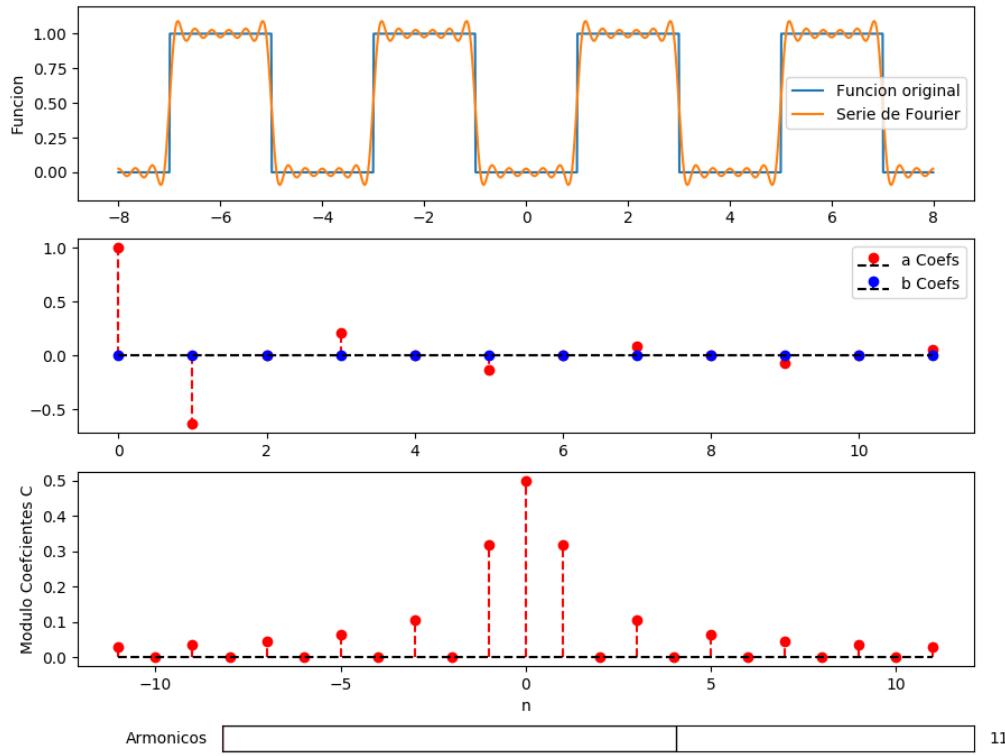
Funcion ejercicio 2.3.2



TIP: Para generar esta señal basta con sumar las dos señales anteriores (el periodo sigue siendo 4s)

1. Realice el desarrollo en serie de Fourier y muestre el resultado con 3 valores diferentes de n.
2. Analice los valores de los componentes, tanto a y b como los componentes complejos.

3. Describa la forma de esta señal y de la anterior (par, impar o ninguna) y explique la relación entre la forma de la señal y los valores de los coeficientes



10.2.4 Práctica 4: Sistemas Lineales

Los sistemas lineales se pueden representar mediante su función de trasferencia:

$$G(s) = \frac{Y(s)}{X(s)} = \frac{\sum_{k=0}^M b_k s^k}{\sum_{k=0}^N a_k s^k}$$

Siendo $X(s)$ la entrada de un sistema LTI, $Y(s)$ la salida en el dominio de Laplace ($S=j\omega$), b los coeficientes del numerador y a los coeficientes del denominador.

Ejercicio 4.1

La función de transferencia de un filtro paso banda de segundo orden con una frecuencia de corte de 1kHz, una ganancia de banda de 6 (15.56dB) y un factor calidad Q de 4 es la siguiente:

$$H(s) = K \frac{s}{\left(\frac{Q}{\omega_c}\right)s^2 + s + Q\omega_c}$$

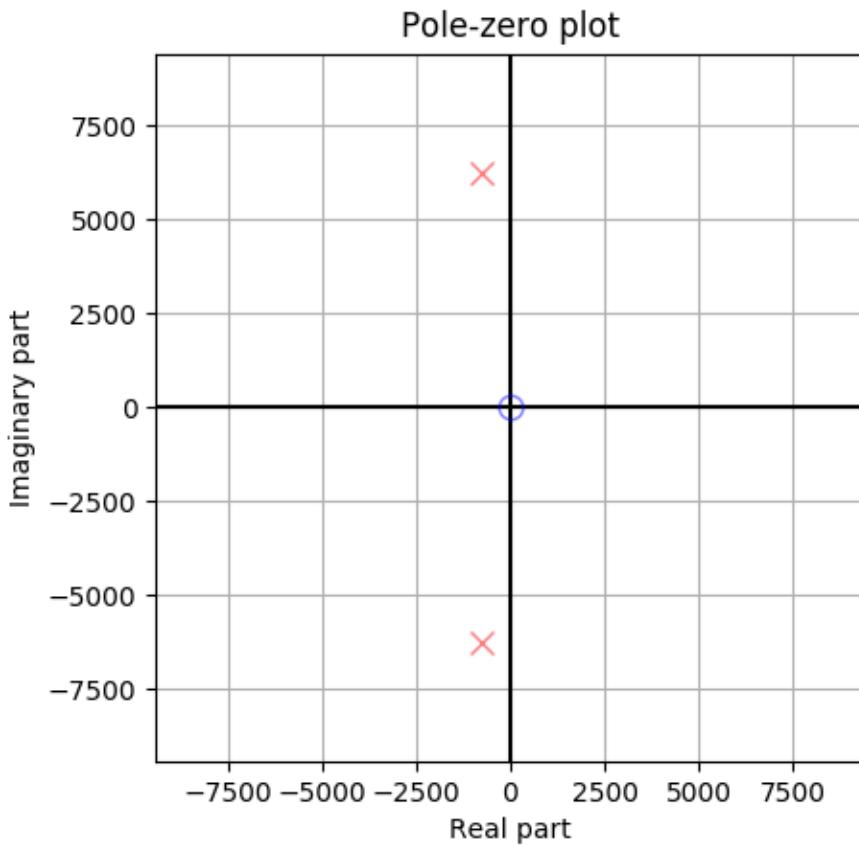
Código para generar el sistema en Python:

```
from labugr import *

# Parametros del filtro
K = 6
fc = 1e3
wc = 2*pi*fc
Q = 4

# Creacion del sistema
numerador = [K*1, 0]
denominador = [Q/wc, 1, Q*wc]
sistema = TransferFunction(numerador, denominador)

# Representacion en el plano Z
zplane(sistema)
```



Como podemos observar en la gráfica, todos los polos del sistema se encuentran a la izquierda del plano z y, al tratarse de un sistema continuo, sabemos que este será probablemente estable. La estabilidad del sistema también se puede comprobar a través la representación de su respuesta impulsiva. El código para generar y representar las respuestas impulsivas y al escalón es el siguiente:

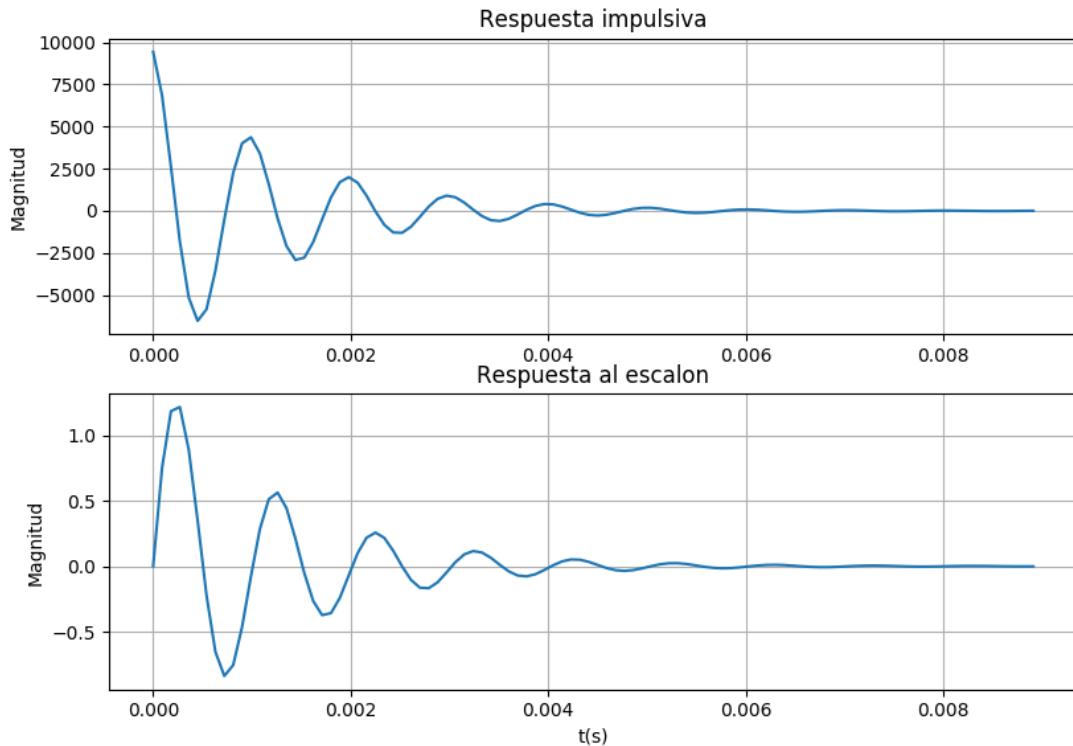
```
# Respuesta impulsiva
t_imp, y_imp = impulse(sistema)

# Respuesta al escalon
t_esc, y_esc = step(sistema)

# Figura para la respuesta impulsiva y al escalon
figure(1)
subplot(2, 1, 1)
title('Respuesta impulsiva')
ylabel('Magnitud')
grid()
plot(t_imp, y_imp)
subplot(2, 1, 2)
title('Respuesta al escalon')
ylabel('Magnitud')
```

```
plot(t_esc, y_esc)
grid()
xlabel('t(s)')
```

Estos son los gráficos obtenidos:

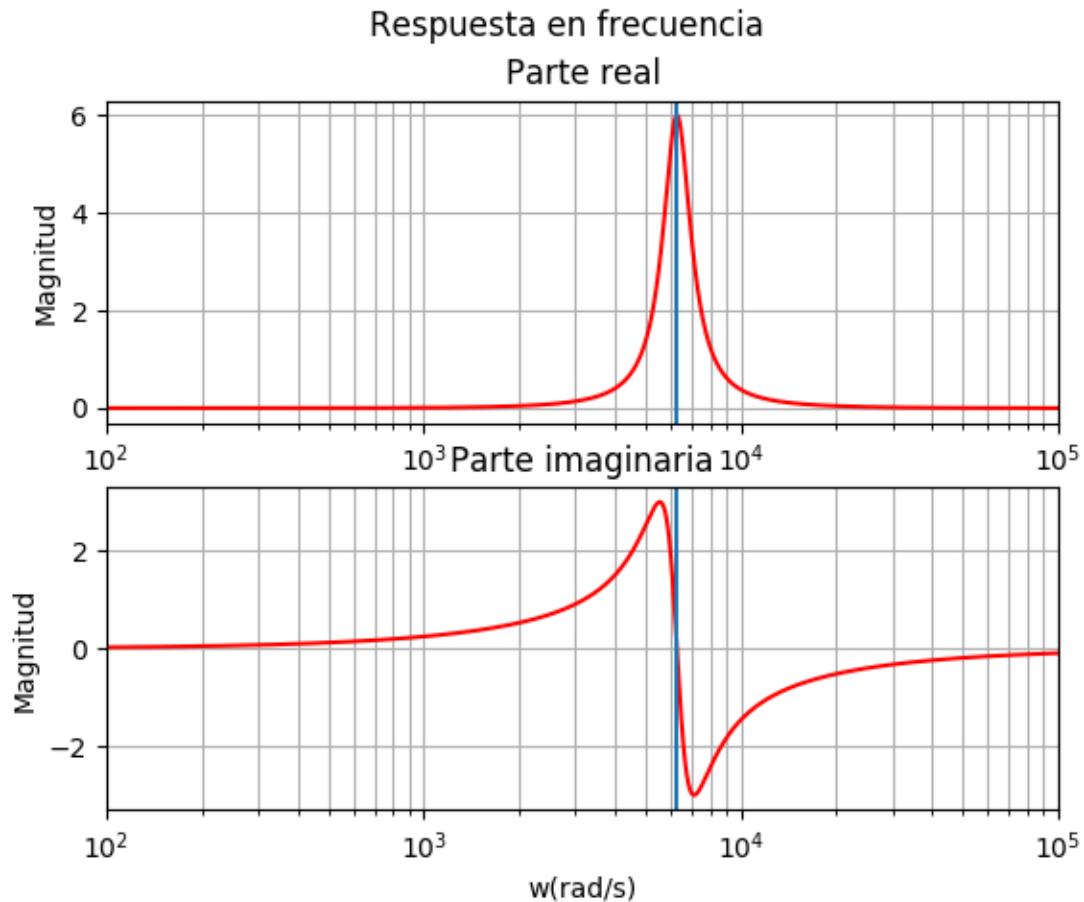


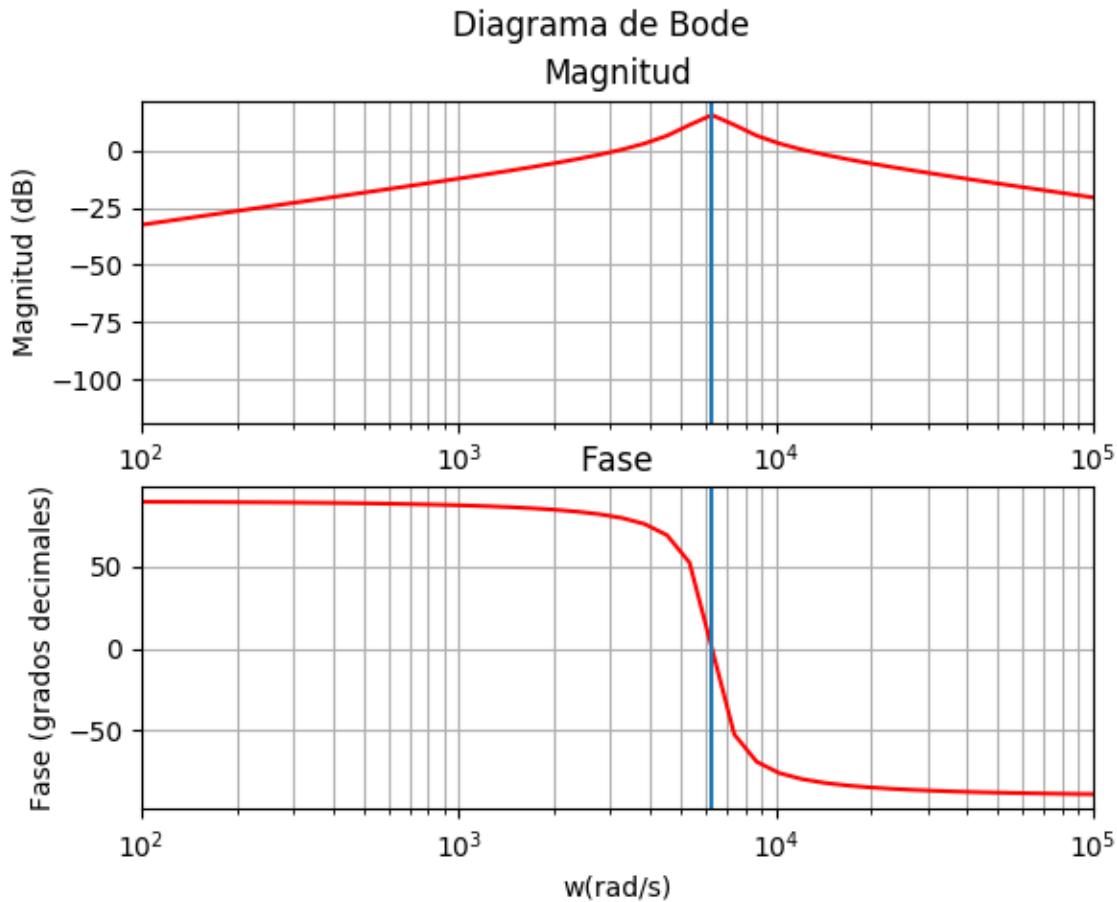
Como podemos observar, el sistema es estable. Para una entrada finita, en este caso un impulso o una señal escalón, la respuesta del sistema es finita y esta disminuye en magnitud hasta alcanzar 0. El código para obtener y representar la respuesta en frecuencia del sistema es el siguiente:

```
# Representacion en frecuencia
w_frecs, magn_frecs = freqresp(sistema)

# Figura para la respuesta en frecuencia
figure(2)
plt.suptitle("Respuesta en frecuencia")
subplot(2, 1, 1)
xlim([1e2, 1e5]) # Los valores interesantes se encuentran en este intervalo
title('Parte real')
ylabel('Magnitud')
plt.semilogx(w_frecs, magn_frecs.real, 'r') # Eje horizontal en escala logaritmica
grid(True, which="both")
plt.axvline(wc) # Linea vertical representando la frecuencia de corte
subplot(2, 1, 2)
```

```
 xlim([1e2, 1e5])
 title('Parte imaginaria')
 ylabel('Magnitud')
 plt.semilogx(w_frecs, magn_frecs.imag, 'r')
 plt.axvline(wc)
 xlabel('w(rad/s)')
 grid(True, which="both")
```





Ejercicio 5.2

Estudie la estabilidad, analice la respuesta en frecuencia, el diagrama de bode y razoné si se asemejan a algún tipo de filtro los siguientes sistemas:

$$1) \quad H_1(s) = \frac{s+1}{s^2 + 4s + 4}$$

$$2) \quad H_2(s) = \frac{s(s+2)}{(s-1/2)(s+4)}$$

$$3) \quad y(t) = 0.5x(t) + 0.2x(t-1) + 0.5y(t-1) - 0.1y(t-2)$$

10.3 Señales digitales

10.3.1 Práctica 1: Señales básicas

El objetivo de esta práctica es de servir como introducción a la generación de señales discretas básicas en Python. Es importante recordar que al inicio de cualquier práctica se debe utilizar el comando ‘from labugr import *’ para importar todas las funciones del módulo labUGR al entorno de trabajo.

1 Funciones básicas

En la librería labUGR se encuentran una serie de funciones para la creación de vectores, principalmente: ‘array’, ‘arange’, ‘zeros’, ‘ones’ y ‘full’. Para obtener la documentación de cualquiera de estas funciones, y de las utilizadas más adelante, se puede utilizar los métodos help() o ayuda(), para la documentación traducida al español (e.j. ayuda(arange)).

Por ejemplo, para crear un vector de 30 muestras con 5 como valor se utiliza el siguiente código en Python:

```
>>> L = 30
>>> vector = full(30, 5)
```

Hay que tener en cuenta que, en Python, a diferencia de Matlab, los índices de los valores dentro de un vector empiezan en 0 no 1. Al representar los vectores, estos siempre estarán definidos entre 0 y longitud-1. Para especificar el rango de los índices de las muestras es necesario crear un vector con los índices de las muestras utilizando la función arange:

```
>>> nn = arange(-15, 15)
```

Esta función crea un vector de números enteros con todos los valores de números enteros entre el extremo izquierdo y el extremo derecho -1. Es importante que la longitud de este vector sea la misma que la del de la señal o su representación producirá errores.

2 Señal impulso unitario

La señal impulso unitario es la señal digital más básica:

$$\delta(n - n_0) = \begin{cases} 1 & \text{si } n == n_0 \\ 0 & \text{si } n \neq n_0 \end{cases}$$

Esta se puede generar con el siguiente código de Python:

```
>>> L = 20
>>> impulso = zeros(20)
>>> impulso[0] = 1
>>> stem(impulso)
```

```
>>> show()
```

La función stem se utiliza, en vez de plot, para que los valores no se grafiquen unidos. Lea la documentación de la función unit_impulse() y utilícela en vez del código anterior. Grafique también las siguientes señales:

$$x_1(n) = 0.5\delta(n - 1) - 2\delta(n + 3) \text{ para } -10 < n \leq 20$$

$$x_2(n) = x_1 + 0.5\delta(n + 8) \text{ para } -15 < n < 4$$

Preste especial atención a los rangos de los vectores de muestras y utilice arange correctamente para generarlos.

3 Señal escalón

Otra de las funciones esenciales para el procesado de señales es la función escalón unitario. Esta se define de la siguiente manera:

$$u(n) = \begin{cases} 1 & \text{si } n \geq 0 \\ 0 & \text{si } n < 0 \end{cases}$$

En Python se pueden modificar múltiples valores dentro de un vector utilizando una condición de verdadero o falso como índice. El siguiente código es utilizado para generar un vector de 30 muestras (entre -15 y 14) que representa el escalón unitario:

```
>>> escalon = zeros(30)
>>> nn = arange(-15, 15)
>>> escalon[nn>=0] = 1
>>> stem(nn, escalon)
>>> show()
```

Utilice las propiedades del desplazado temporal de funciones para crear la siguiente señal rectangular a partir de dos señales escalón:

$$x(n) = \begin{cases} 1 & \text{si } 3 \leq n < 8 \\ 0 & \text{else} \end{cases} \text{ para } -5 \leq n \leq 10$$

Lea la documentación de la función square () y utilícela en vez del código anterior para generar la misma señal rectangular.

4 Otras señales

Gracias a las funciones incluidas en la librería labUGR, que adaptan las operaciones matemáticas básicas incluidas en Python para que puedan trabajar con vectores, podemos generar vectores con una gran variedad de funciones matemáticas. Se puede utilizar cualquier función trigonométrica, hiperbólica, exponencial, logarítmica, ... e incluso utilizar funciones propias con lambda. Hay que tener en cuenta que Python siempre utiliza radianes por defecto.

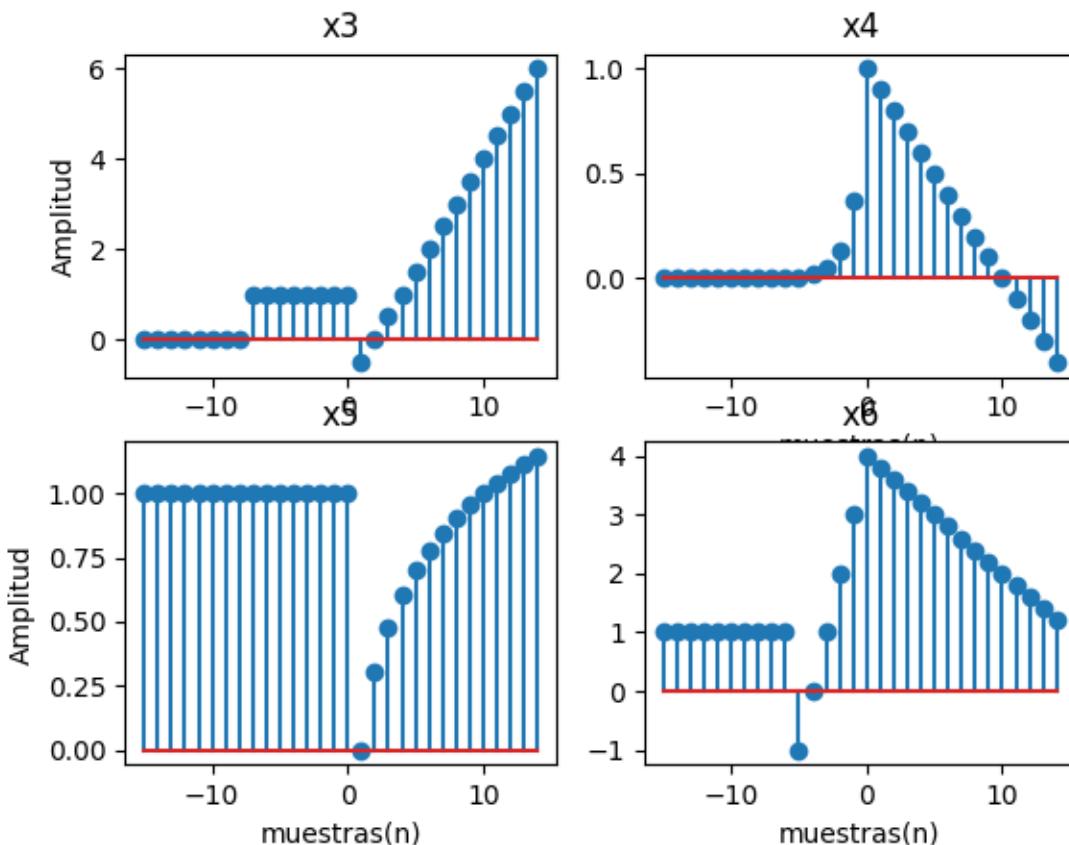
Por ejemplo, el código para generar una señal sinusoidal discreta entre 0 y 20 con una frecuencia de 0.5 radianes por muestra y una fase de 90 grados:

```
>>> nn = arange(21)
>>> x1 = sin(0.5*nn + pi/2)
>>> stem(nn, x1)
>>> show()
```

Represente la siguiente señal y compárela con la anterior:

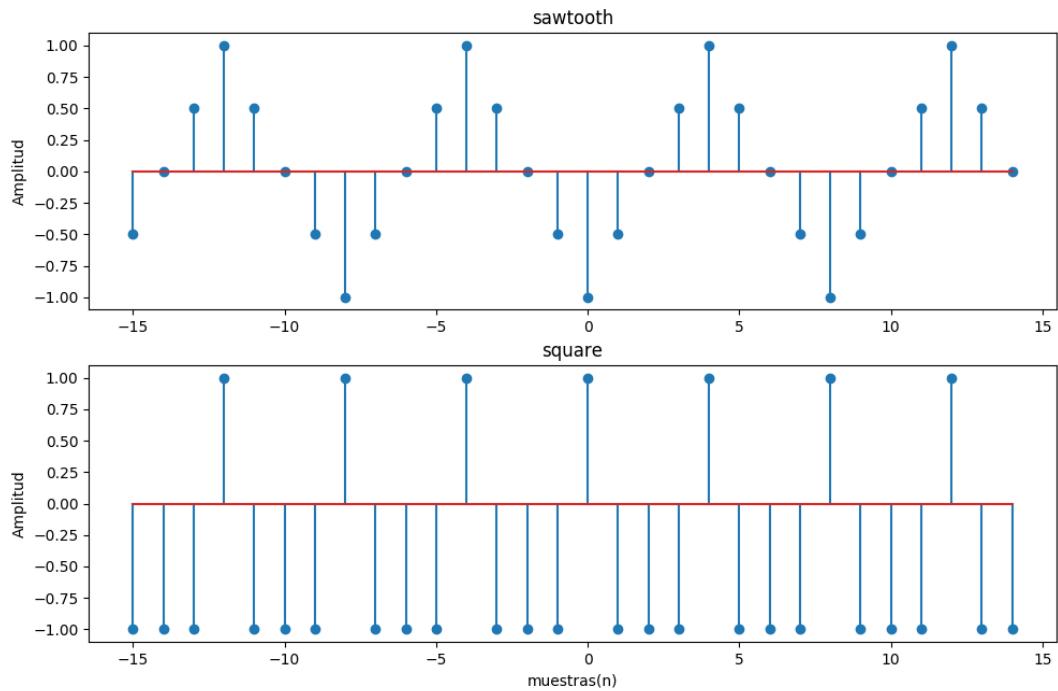
$$x_2(n) = \cos(0.5n) \text{ para } 0 \leq n \leq 20$$

Genere el código en Python que representa las siguientes señales:



Nota: en x5 se utiliza un logaritmo decimal y en x4 una función exponencial.

Lea la documentación de `sawtooth` y `square` y obtenga el código que representa las siguientes señales:



10.3.2 Práctica 2: Transformada discreta de Fourier

Sabemos que la transformada discreta de Fourier (DFT) es una versión muestreada de la transformada de Fourier, donde su valor es:

$$X(k) \equiv X(\omega)|_{\omega=\omega_0 k} \quad \text{siendo} \quad \omega_0 = \frac{2\pi}{N} \quad y \quad k = 0, 1, 2, \dots, N-1$$

Para obtener la DFT de una señal digital de N muestras se utiliza la siguiente ecuación:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j\frac{2\pi k}{N}n} \quad k = 0, 1, 2, \dots, N-1$$

Si analizamos esta ecuación podemos determinar que se necesitan dos bucles `for` para encontrar los valores del resultado: uno que itere entre $k=0$ y $k=N-1$ y otro que represente la sumatoria ($n=0$ a $n=N-1$). El código para implementar la anterior ecuación en Python es el siguiente:

```
def dft(senal):
    """
    Funcion para calcular la transformada discreta de una secuencia de valores
    """
    N = len(senal) # numero de muestras
    resultado = zeros(N, dtype=np.complex_) # Inicializando el resultado como
    # un vector de numeros complejos (sin dtype la parte imaginaria se perderia)

    # Ecuacion principal
```

```

for k in range(0, N):
    for n in range(0, N):
        resultado[k] += senal[n]*exp(-2j*pi*k*n/N)

# Estableciendo la precision del resultado (8 decimales)
return round(resultado, 8)

```

Esta ecuación se puede implementar utilizando el producto exterior de dos vectores que representen los instantes de tiempo en los que se muestrea la señal (n) y los índices de los valores en frecuencia (k). Para ello se utiliza la función ‘outer’ de labUGR, que produce todas las combinaciones de n-k que necesitamos para obtener la DFT. Por ejemplo:

```

>>> a = [1, 2, 3, 4]
>>> b = [0, 1, 0, 1]
>>> outer(a,b)
array([[0, 1, 0, 1],
       [0, 2, 0, 2],
       [0, 3, 0, 3],
       [0, 4, 0, 4]])

```

El código para implementar esta función en Python es el siguiente:

```

def dft2(senal):
    """
    Funcion para calcular la transformada discreta de una secuencia de valores a
    traves del calculo matricial
    """
    N = len(senal) # numero de muestras
    k = arange(N) # vector de k
    n = arange(N) # vector de n

    expon = exp(-2j*pi * np.outer(n, k) / N) # parte exponencial
    # outer es equivalente a los dos bucles for

    # producto vectorial de la senal de entrada y la exponencial
    resultado = dot(senal, expon)

    # Estableciendo la precision del resultado (8 decimales)
    return round(resultado, 8)

```

Ejercicio 1

Calcule manualmente la transformada discreta de Fourier de las siguientes secuencias y compruebe que el resultado es correcto mediante las funciones anteriores:

1. $x_1(n) = [0, 0, 0, 2, 3, 4, 0, 0, 0]$
2. $x_2(n) = [1 - 2j, 3, -2, 8 + 3j]$
3. $x_2(n) = [e^{-2j}, 0, e^{-1+j}, 2, 0]$

Ejercicio 2

Lea la documentación de la función ‘fft’ y utilícela para obtener las transformada de Fourier de las secuencias anteriores.

Ejercicio 3

La transformada inversa de Fourier discreta se puede obtener mediante la siguiente ecuación:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j \frac{2\pi k}{N} n} \quad n = 0, 1, 2, \dots, N - 1$$

Escriba las funciones que implementarían esta ecuación en Python, utilizando la forma matricial y los bucles for. Utilice la función de labUGR ‘allclose’ para comprobar que la transformada inversa de Fourier discreta implementada recupera la señal entrada con una tolerancia de 8 cifras decimales.

10.3.3 Práctica 3: Diseño de filtros por el método de las ventanas

Una de las formas de diseñar un filtro digital FIR es mediante el método de las ventanas. Este método requiere de 4 etapas:

- Especificar las propiedades optimas deseadas en frecuencia del filtro.
- Realizar una transformada inversa de Fourier discreta de la respuesta en frecuencia deseada para obtener los coeficientes de un filtro. Este filtro tendrá una respuesta infinita al impulso.
- Utilizar una ventana para acotar la respuesta impulsiva del filtro.
- Aplicar un retardo para obtener un filtro causal (igual a ceros para valores menores que 0).

En este ejemplo diseñaremos un filtro digital paso baja. La respuesta en frecuencia $|H(\omega)|$ ideal de un filtro paso baja con una frecuencia de corte de ω_c es la siguiente:

$$|H(\omega)| = \begin{cases} 1 & \text{para } -\omega_c < \omega < \omega_c \\ 0 & \text{else} \end{cases}$$

Sabemos que la transformada inversa de Fourier de una función rectangular es una función sinc por lo que la respuesta impulsiva del filtro ideal es la siguiente:

$$h(n) = \frac{\omega_c}{\pi} \operatorname{sinc}\left(\frac{\omega_c n}{\pi}\right)$$

Conociendo las propiedades de la función sinc sabemos que esta respuesta es claramente no causal e infinitamente larga. Para obtener un filtro con respuesta impulsiva finita debemos utilizar una ventana para acotar esta respuesta. Ya que en Python no podemos representar vectores infinitos, generaremos un vector acotado de muestras (esto es análogo a utilizar una ventana rectangular o ‘boxcar’). El código de Python utilizado es el siguiente:

```
from labugr import *

wc = pi/4 #frecuencia de corte normalizada
M=20 #numero de muestras
n = arange(-M,M) #vector de muestras
h = wc/pi * sinc(wc*(n)/pi) #respuesta impulsiva ideal del filtro acotada
#entre -M y M
n = n + M #Retardo para tener un filtro causal

w, H = freqz(h, 1, whole=True) # dominio en frecuencia del filtro
H = np.fft.fftshift(H) # centrar la respuesta en frecuencia en 0
w = w-pi #Centramos el vector de frecuencias en 0

figure()
#Representamos la respuesta impulsiva del filtro
subplot(3,1,1)
stem(n, h)
xlabel("n(muestras)")
ylabel("h(n)")
grid()
```

```

#Representamos el espectro en frecuencia
subplot(3,1,2)
plot(w,abs(H))
xlim([-pi/2, pi/2])
plt.vlines([-wc,wc],0,1.2,color='g',lw=2.,linestyle='--')
xlabel("w(muestras/s)")
ylabel("|H(w)|")
grid()

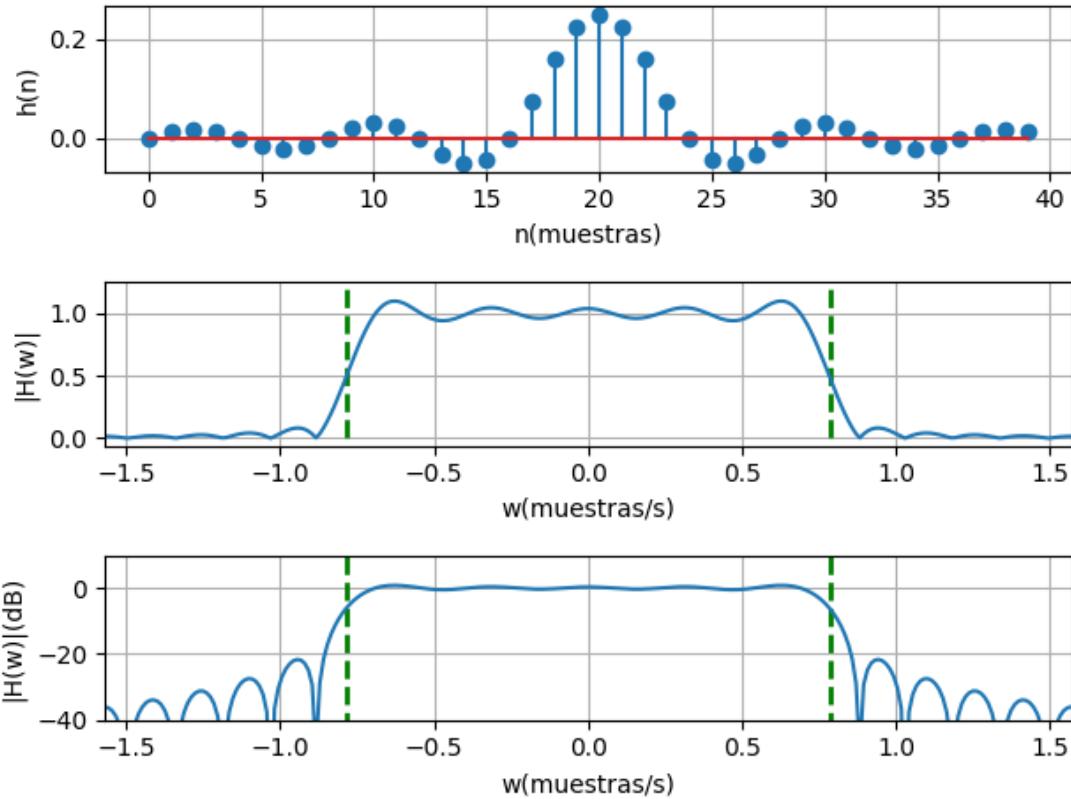
#Representamos el espectro en frecuencia end dB
subplot(3,1,3)
plot(w,20*np.log10(abs(H)))
xlim([-pi/2,pi/2])
ylim([-40, 10])
plt.vlines([-wc,wc],10,-40,color='g',lw=2.,linestyle='--')
xlabel("w(muestras/s)")
ylabel("|H(w)| (dB)")
grid()

plt.tight_layout()

show()

```

Las gráficas obtenidas se pueden observar en la siguiente figura (las líneas verticales representan la frecuencia de corte del filtro):



Como se puede observar, la respuesta en frecuencia del filtro obtenido no es exactamente igual a la respuesta en frecuencia ideal y se puede observar un rizado en los extremos de la banda de paso. Esto es debido a la utilización del método de las ventanas.

Ejercicio 1

Repita el ejemplo anterior, pero utilizando las siguientes ventanas y comente las diferencias entre cada caso:

- 1.Hamming
2. Kaiser con $\beta=6$
3. Blackman

Para ello, lea la documentación de la función `get_window()` y utilícela para crear las ventanas.

Multiplique la respuesta impulsiva del filtro por la ventana para aplicarla.

Ejercicio 2

Este mismo proceso se puede seguir para generar los filtros paso alta, paso banda y elimina banda. Las respuestas impulsivas de estos filtros son las siguientes:

1. Paso alta:

$$h(n) = \begin{cases} -\frac{\omega_c}{\pi} \text{sinc}\left(\frac{\omega_c n}{\pi}\right) & \text{para } n \neq 0 \\ 1 - \frac{\omega_c}{\pi} & \text{para } n = 0 \end{cases} \quad \text{siendo } \omega_c \text{ la frecuencia de corte normalizada.}$$

2. Paso banda:

$$h(n) = \frac{\omega_2}{\pi} \text{sinc}\left(\frac{\omega_2 n}{\pi}\right) - \frac{\omega_1}{\pi} \text{sinc}\left(\frac{\omega_1 n}{\pi}\right) \quad \text{siendo } [\omega_1: \omega_2] \text{ las frecuencias normalizadas que representan la banda de paso.}$$

3. Elimina banda:

$$h(n) = \begin{cases} \frac{\omega_1}{\pi} \text{sinc}\left(\frac{\omega_1 n}{\pi}\right) - \frac{\omega_2}{\pi} \text{sinc}\left(\frac{\omega_2 n}{\pi}\right) & \text{para } n \neq 0 \\ 1 - \frac{\omega_2 - \omega_1}{\pi} & \text{para } n = 0 \end{cases}$$

Siendo $[\omega_1: \omega_2]$ las frecuencias normalizadas que representan la banda eliminada.

Repita los pasos seguidos en el ejemplo del filtro paso baja con el resto de filtros.

Ejercicio 3

En el módulo labUGR se encuentra la función `firwin()`, que permite generar un filtro digital a partir del número de coeficientes del filtro deseado y la frecuencia de corte. Esta función utiliza la ventana Hamming por defecto. Lea atentamente la documentación de la función (utilizando `help()` o `ayuda()`), cree y represente el filtro paso baja del ejemplo inicial utilizando esta función para un filtro de orden 4, 7 y 10 y compare los resultados con el ejemplo inicial.

Nota: el parámetro pasado a la función es el número de coeficientes del filtro (orden del filtro + 1).

Ejercicio 4

Utilice de nuevo la función firwin() para generar un filtro paso alta, un filtro paso banda y un filtro elimina banda; todos ellos con orden del filtro igual a 5. Compare las gráficas con ejercicio anterior.

Nota: para generar los diferentes filtros se utilizan combinaciones de los parámetros 'pass_zero' y las frecuencias de corte 'cutoff'.

11 Conclusiones y líneas futuras

11.1 Conclusiones

Con el desarrollo de este proyecto, se ha logrado alcanzar el objetivo principal de este Trabajo Fin de Grado, la creación de una alternativa Open Source y gratuita a Matlab para aquellas personas que deseen entrar en el mundo del procesado de señales a través de Python.

Durante el proceso de elaboración de esta librería hemos podido comprobar que el diseño de módulos de Python que incluyen código en otros lenguajes no es algo trivial y necesita de una serie de pasos para la correcta compilación y distribución de estos. A pesar de ello, Python provee una amplia variedad de herramientas con extensas documentaciones para facilitar la tarea a los desarrolladores.

También hemos podido comprobar que, debido a que Python y el resto de herramientas y módulos utilizados son Open Source y mantenidos por colaboradores voluntarios, no es inusual encontrar mucha información y tutoriales desactualizados.

Con la aplicación práctica y los ejemplos creados para los diferentes apartados hemos podido constatar que Python es una opción completamente viable como software para el procesado y representación de señales digitales y analógicas.

11.2 Líneas futuras

Una vez finalizado este proyecto se pueden plantear varias posibilidades a la hora de avanzar el proyecto:

- **Aplicación práctica.** si se decide utilizar este módulo para realizar un laboratorio de señales para alguna de las asignaturas del grado sería necesario instalar este junto a Python en las imágenes de sistema utilizadas en los laboratorios. Gracias al hincapié que se ha hecho en facilitar la instalación de este módulo tanto en Windows como Linux, este proceso no debería ser complicado. Se debería analizar si la mejor opción es la inclusión de este módulo en la imagen o requerir la instalación manual del módulo en cada sesión.

- **Ampliación de la librería.** Gracias a la modularización de esta librería, esta se podría fácilmente extender más adelante con otras funcionalidades como, por ejemplo, para el procesado de imágenes. Siguiendo los pasos descritos en los apartados de compilación y distribución se puede crear un entorno para continuar con el desarrollo de esta librería y la distribución de nuevas versiones de esta.

El código fuente del módulo LabUGR se encuentra en un repositorio de GitHub público lo que facilita el acceso a este a cualquier persona interesada en el desarrollo posterior de esta librería.

12 Bibliografía

Referencias utilizadas durante la creación de este proyecto, la mayoría de ellas en línea (ultimo acceso en octubre-noviembre de 2017, los contenidos de estas pueden variar en consultas posteriores).

- Web oficial Python: <https://www.python.org/>
- Web oficial Matlab: <https://es.mathworks.com/products/matlab.html>
- Módulo NumPy:
 - Página oficial: <http://www.numpy.org/>
 - Código fuente(GitHub): <https://github.com/numpy/numpy>
 - Guía de usuario 1.13.0(PDF): <https://docs.scipy.org/doc/numpy/numpy-user-1.13.0.pdf>
 - Documentación 1.13.0(PDF): <https://docs.scipy.org/doc/numpy-1.11.0/numpy-ref-1.13.0.pdf>
 - Submódulo distutils(creación de paquetes): <https://docs.scipy.org/doc/numpy-1.13.0/reference/distutils.html>
 - Python – C/FORTRAN: <https://docs.scipy.org/doc/numpy-1.10.0/user/c-info.python-as-glue.html>
- Módulo Matplotlib:
 - Página oficial: <https://matplotlib.org/index.html>
 - Código fuente(GitHub): <https://github.com/matplotlib/matplotlib>
 - Documentación 2.1.0(PDF): <https://matplotlib.org/Matplotlib.pdf>
- Módulo SciPy:
 - Página oficial: <https://www.scipy.org/>
 - Código fuente(GitHub): <https://github.com/scipy/scipy>
 - Documentación 0.19.1(PDF): <https://docs.scipy.org/doc/scipy-0.19.1/scipy-ref-0.19.1.pdf>
- Modulo PyAudio:
 - Página oficial y documentación: <https://people.csail.mit.edu/hubert/pyaudio/>
 - Código fuente(GitHub): <https://github.com/jleb/pyaudio>
- Modulo AudioRead:
 - Código fuente(GitHub) y documentación: <https://github.com/beetbox/audioread>
- Módulo mpmath:
 - Página oficial y documentación: <http://mpmath.org/>
 - Código fuente(GitHub): <https://github.com/fredrik-johansson/mpmath>
- Documentación de otros módulos:
 - [Wheel](#): Python compressed packaging.
 - [Twine](#): utility for interacting with PyPI.
 - [Delocate](#): tool for Python wheels in Mac OS.
 - [AuditWheel](#): tool for Python wheels in Linux.
 - [Pytest](#): testing framework for Python.

➤ Índice de propuestas de mejora de Python, PEP (<https://www.python.org/dev/peps/>):

- [PEP 8](#) -- Style Guide for Python Code
- [PEP 20](#) -- The Zen of Python
- [PEP 257](#) -- Docstring Conventions
- [PEP 373](#) -- Python 2.7 Release Schedule
- [PEP 425](#) -- Compatibility Tags for Built Distributions
- [PEP 427](#) -- The Wheel Binary Package Format 1.0
- [PEP 513](#) -- A Platform Tag for Portable Linux Built Distributions

➤ Wikipedia de Python (<https://wiki.python.org/moin/FrontPage>):

- Python 2.7x vs 3.x: <https://wiki.python.org/moin/Python2orPython3>
- Documentación: <https://wiki.python.org/moin/Documentation>

➤ Libros sobre Python:

- Kenneth Reitz, [*The Hitchhiker's Guide to Python*](#)
- Zed Shaw, [*Learn Python the Hard Way*](#)
- Unpingco, José (2013), [*Python for Signal Processing*](#), Springer

➤ Distribución de paquetes de Python (<https://packaging.python.org/>):

- Guía: <https://packaging.python.org/tutorials/distributing-packages/>
- Especificaciones de PyPA: <https://packaging.python.org/specifications/>

➤ The Hitchhiker's Guide to Packaging: <https://the-hitchhikers-guide-to-packaging.readthedocs.io/en/latest/index.html>

➤ Python Packaging Authority: <https://www.pypa.io/en/latest/>

➤ Sitios web de preguntas/resuestas:

- Stack Overflow: <https://stackoverflow.com/>
- Quora: <https://quora.com/>
- Experts Exchange: <https://www.experts-exchange.com/>
- Reddit: <https://www.reddit.com>
- Stack Exchange: <https://stackexchange.com/>
- Signal Processing Stack Exchange: <https://dsp.stackexchange.com/>

➤ Web sobre Python (Dev): <https://dev.to/t/python>

➤ Guía de Docker: <https://docs.docker.com/engine/userguide/>

➤ Adventures in Signal Processing with Python: <https://www.embeddedrelated.com/showarticle/197.php>

➤ Proyecto manylinux: <https://github.com/pypa/manylinux>

Apéndice A: Planificación y costes del proyecto

En este apartado se describen con detalle las diferentes fases en las que se ha dividido el desarrollo de este proyecto, así como una estimación de los costes de este proyecto.

A.1 Planificación

El presente trabajo se ha desarrollado en diferentes fases entre marzo y noviembre de 2017, como se muestra en el desarrollo temporal recogido en el diagrama de Gantt de la figura 21. Las labores desarrolladas en cada una de las partes quedan descritas a continuación:

- **Fase 1: Búsqueda de información.** Se ha realizado una extensa investigación acerca de las diferentes opciones de software para el procesado de señales disponibles en la actualidad. Una vez escogido Python como lenguaje de programación a utilizar para la creación de la librería, se ha investigado qué versión específica de Python es la más adecuada para el proyecto.
- **Fase 2: Familiarización con Python.** Esta parte del trabajo abarca todo lo relativo a la adquisición de un grado alto de soltura en la distribución básica de Python, es decir, sin tener en cuenta ningún módulo externo no incluido por defecto en su instalación. Para ello se han seguido una serie de libros y tutoriales, así como la realización de proyectos para practicar.
- **Fase 3: Recopilación de módulos y funciones a implementar.** En esta fase se realiza un trabajo de investigación para crear una lista con todas las funciones y módulos considerados indispensables para el procesado de señales en Python. Se realiza un proceso de familiarización con estos módulos utilizando la documentación oficial de estos y se empiezan a explorar las diferentes formas de incluir estas funciones en un módulo único (labUGR).
- **Fase 4: Creación de la librería.** Siguiendo los estándares de empaquetado y desarrollo de módulos en Python; en esta fase se produce el desarrollo de la librería en sí.
- **Fase 5: Distribución de la librería.** Esta fase se podría dividir en tres subfases: (1) Mejor forma de distribuir la librería para facilitar su instalación por parte del usuario final. Herramientas blablablá; (2) familiarización con las herramientas a utilizar para llevar a cabo la distribución; (3) creación de los scripts para automatizar la distribución de labUGR.

- **Fase 6: Documentación de las funciones.** Documentación de las funciones principales que componen el módulo labUGR, así como la creación de pequeños ejemplos que muestran estas funciones en uso. Para completar esta fase ha sido necesario obtener un conocimiento en detalle del funcionamiento de cada una de estas funciones.
- **Fase 7: Diseño de la aplicación práctica.** Diseño de una serie de prácticas de laboratorio propuestas para las asignaturas de Señales Digitales y Sistemas Lineales de la Universidad de Granada.
- **Fase 8: Redacción de la memoria del proyecto.** En esta fase se ha estructurado y documentado todo el trabajo realizado.
- **Fase 9: Exposición del proyecto.**

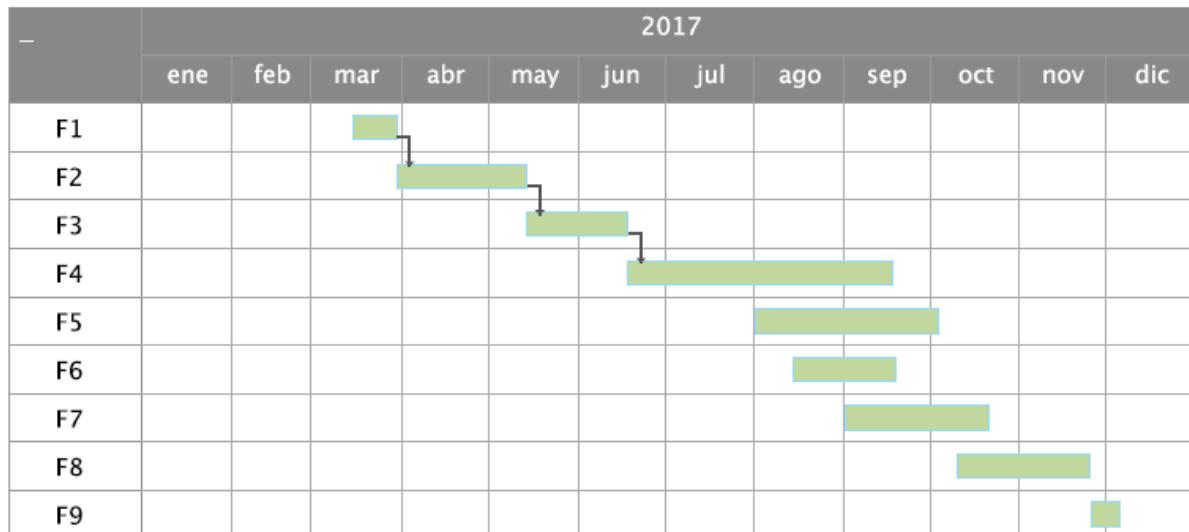


Figura 24: Diagrama de Gantt del proyecto.

A.2 Costes

Debido a que este proyecto se ha realizado en un ámbito académico y utilizando únicamente software libre, el coste real de este proyecto ha sido nulo. No obstante, se puede realizar una estimación de lo que hubiese costado este proyecto si se hubiese desarrollado en un ámbito privado. A continuación, se detallan cada uno de los conceptos que conformarían el precio final.

- **Horas de trabajo.** El trabajo fin de grado está valorado en 12 créditos ECTS o 300 horas de trabajo realizado por el alumno. Si se establece la hora de trabajo en 15 €, se obtiene un coste de 4.500 €. También se pueden considerar las horas de trabajo realizado por los tutores como un total de 50 horas a 40 € la hora, aumentando el coste de las horas de trabajo a un total de 6.500 €.
- **Software utilizado.** La mayor parte del software utilizado es completamente libre y gratuito, pero algunos de ellos están limitados a un ámbito académico, requiriendo la obtención de licencias para un uso comercial.
 - **Python:** El uso de Python, así como todos los diferentes módulos utilizados (SciPy, Matplotlib, NumPy, PyAudio, AudioRead, mpmath, Delocate, Auditwheel, Twine y GoogleTrans), es completamente gratuito tanto en un amito académico como comercial.
 - **PyCharm y Sublime Text:** los entornos de desarrollo integrado escogidos para ser utilizados a la hora de desarrollar la librería han sido PyCharm (a través de una licencia educativa) y Sublime Text (a través de una licencia de evaluación). El coste de Sublime Text es de 80 € para una licencia indefinida; el coste de PyCharm es de 7'8 €/mes acumulando un total de 70'2 € en los 9 meses de trabajo. EL coste total de estos IDEs asciende a 150'2 €.
 - **Docker:** con la versión ‘community’ de Docker (gratuita) se satisfacen todas las necesidades de este proyecto.
 - **MinGW:** este compilador de GCC para Windows es completamente gratuito.
 - **GitHub:** al utilizar un repositorio público, el precio de este es 0 €.
 - **Parallels Desktop:** el software utilizado para empaquetar la librería en Linux ha sido Parallels Desktop. El precio de una licencia nueva asciende a 63 €.
- **Equipo de trabajo.** Para el desarrollo de este proyecto se han utilizado dos ordenadores diferentes, uno con Mac OS para el desarrollo de la librería en Mac y otro con Windows para el empaquetado en Windows, valorados en 1300 € y 1000 € respectivamente. Estimando las vidas útiles de estos en 5 años y teniendo en cuenta que la mayor parte del proyecto se ha desarrollado en el ordenador Mac, se puede considerar que el coste por los equipos de trabajo asciende a 228'3 € (9 meses de utilización del ordenador Mac y 2 del ordenador Windows).

Sumando los diferentes conceptos, se tiene que el coste del proyecto asciende a un total de 6.713'2 €.



ugr | Universidad
de **Granada**

TRABAJO FIN DE GRADO

INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Laboratorio de Señales en Python

Autor

Luis Serra García

Directores

Sonia Mota Fernández
Pablo Padilla de la Torre



*ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN*

Granada, noviembre de 2017