Trabajo Práctico 1 - Señales y Sistemas

September 23, 2022

1 Consignas

Para realizar este trabajo práctico, cree una carpeta nueva con el siguiente formato de nombre: *Apellido_Nombre_TP1.* Dentro de la misma carpeta, crear los siguientes archivos: - **main.ipynb** - **functions.py** - **compare_mics.py**

Luego, dentro de la misma carpeta crear otras dos carpetas llamadas - **images/** - **files/**

dentro de la carpeta *archivos* ubicar los archivos *loudspeaker_Genelec.npy* y *loudspeaker_JBL.npy* que se encuentran en el drive.

Finalmente, la estructura del directorio de trabajo debería ser la siguiente:

```
Apellido_Nombre_TP1/
    main.ipynb
    functions.py
    compare_mics.py

  images/

  files/
       loudspeaker_Genelec.npy
       loudspeaker_JBL.npy
```

1.1 PUNTO 1

Dentro del script *functions.py* definir una función que sea capaz de recibir una variable e imprimir información descriptiva de la misma. Los tipos de variables que debe poder aceptar son: enteros, flotantes, strings, arrays, tuplas, diccionarios. Para todos los casos, se debe imprimir un mensaje en donde se informe de que tipo de variable se trata. Adicionalmente: - Si es un array, imprimir: - cantidad de dimensiones - tamaño de cada dimensión - cantidad de elementos - Si es una lista o tupla, imprimir: - cantidad de elementos - Si es un diccionario, imprimir: - pares de key:values - cantidad de elementos que componen el diccionario

Importe esta función desde el notebook principal *main.ipynb* y realice pruebas.

1.2 PUNTO 2

Dentro del script *functions.py* defina una función llamada *gen_discrete_signals.* La misma debe ser capaz de generar y graficar el siguiente conjunto de señales con sus respectivos argumentos de entrada:

- Impulso unitario
  - Inicio de la señal (en número de muestra)
  - Fin de la señal (en número de muestra)
  - Número de muestra en la que se ubica el impulso
- Escalón unitario
  - Inicio de la señal (en número de muestra)
  - Fin de la señal (en número de muestra)
  - Número de muestra en la que inicia el escalón
- Tren de impulsos
  - Inicio de la señal (en número de muestra)
  - Fin de la señal (en número de muestra)
  - Inicio del tren de pulsos
  - Fin del tren de pulsos
- Triangular*
  - Inicio de la señal (en número de muestra)
  - Fin de la señal (en número de muestra)
  - Factor de base
- Señal aleatoria normal:
  - Inicio de la señal (en número de muestra)
  - Fin de la señal (en número de muestra)
  - Media
  - Desvío estándar

Además, la función debe contar con la opción de poder guardar tanto los gráficos realizados como los arrays generados.

Importe esta función desde el notebook principal *main.ipynb* y realice pruebas.

1.3 PUNTO 3

Dentro del script *functions.py* defina una función llamada *gen_n_sin* que sea capaz de generar un número arbitrario de señales senoidales. Se debe poder indicar la frecuencia de cada señal senoidal a generar. El largo de la señal y la frecuencia de muestreo pueden permanecer fijos para todas las señales generadas.

La salida debe ser una lista de tuplas que contengan 3 variables `(t, y, label)` donde t es un array correspondiente al vector temporal, y es un array correspondiente al vector de amplitud, y *label* es una string de formato 'sin_N_freq_F' donde **N** es el número de curva y **F** es la frecuencia asociada a dicha curva.

1.4 PUNTO 4

Dentro del script *functions.py* defina una función llamada *plot_sin_list* que sea capaz de plotear un número arbitrario de señales senoidales dentro de una misma figura. La misma debe poder recibir cualquier salida de la función definida en el punto 3, y graficar las senoidales generadas. Además, se debe poder pasarle en forma de kwargs variables de configuración para la función `plot()` del módulo matplotlib.pyplot.

Importe esta función y la función del punto 3 desde el notebook principal *main.ipynb* y realice pruebas.

1.5 PUNTO 5

Los archivos *parlante_Genelec.npy* y *parlante_JBL.npy* contienen información de respuesta en frecuencia y fase a lo largo del espectro frecuencial de dos parlantes distintos. Cada archivo contiene una matriz de 3 filas conformada de la siguiente manera:

- Primera fila: Frecuencia
- Segunda fila: Magnitud
- Tercer fila: Fase

Crear un script llamado comparacion_parlantes.py que sea capaz de leer ambos archivos y realizar el o los gráficos que sean necesarios para visualizar correctamente las diferencias entre ambos parlantes. La cantidad de gráficos, y el diseño de los mismos son aspectos que deben decidir en base a los criterios charlados en clase. Por último, el script debe poder guardar todos los gráficos realizados en formato png.

2 Criterios de Presentación y Aprobación

Se debe comprimir la carpeta **Apellido_Nombre_TP1** en formato **.zip**. La misma debe contener, como mínimo, los archivos mencionados al inicio de este documento. **Cualquier** otro archivo que resulte necesario para el correcto funcionamiento del código, o para el cumplimiento de alguna de las consignas debe incluirse en la entrega. Si por algún faltante el código presentado no se puede ejecutar a la hora de la corrección, el trabajo será descartado **sin excepción**.

Cada estudiante deberá adjuntar y enviar el archivo *Apellido_Nombre_TP1.zip* en un email dirigido a las direcciones *martinbernardomeza@gmail.com* y *ugemassolo@gmail.com* colocando en el asunto: *Apellido TP1_SYS*.

La fecha límite de entrega es el miércoles 28 de septiembre a las 23:59 h.

Cualquier entrega que no cumpla con alguno de los puntos mencionados no será tenida en cuenta, sin excepción.

3 Recomendaciones

3.1 Mantener las buenas prácticas durante el desarrollo de código

Cuando estamos incorporando nuevos usos y costumbres a la hora de formatear el código que escribimos, suele suceder que lo dejamos como un aspecto estético más, y lo revisamos al final. Es decir, primero desarrollo todo el código, y antes de entregarlo o guardarlo, me fijo que no esté incumpliendo ninguna de las buenas prácticas o recomendaciones. Si bien esta modalidad de trabajo puede funcionar para cosas pequeñas, no escala a grandes trabajos. Además, las buenas prácticas no son solo cuestiones estéticas, sino que también son de gran ayuda para evitar cometer errores **durante** el desarrollo de los scripts, ayudando a tener una imagen más clara de que es lo que hice, que es lo que estoy haciendo y que cosas necesito hacer. Por eso, nuestra recomendación es intentar aplicar las buenas prácticas desde un principio, y tenerlas presentes durante todo el desarrollo de este trabajo.

3.2 Trabajar de formada ordenada y prolija

En estas instancias, el orden se vuelve fundamental para no perder tiempo en bugs que provengan de variables mal nombradas, identaciones o espaciados no respetados, errores de sintaxis, etc. El orden se vuelve un requisito fundamental cuando necesitamos desarrollar scripts complejos.

3.3 *Divide et impera*

El concepto de divide et impera (divide y vencerás) es un paradigma de programación que resulta muy útil para el diseño de ciertos algoritmos. En nuestro caso, para este trabajo, la complejidad mayor reside en el diseño de funciones. Es importante tratar de aplicar este concepto a la hora de desarrollar estas funciones, tratando siempre de descomponer el problema principal (cada inciso) en problemas más pequeños, y crear funciones que resuelvan esos problemas. Alinearse a esta manera de trabajar hace que terminemos diseñando funciones cortas, concisas, fáciles de interpretar y de debuggear, que luego se concatenan unas con otras para realizar el procesamiento general buscado.

3.4 Aprovechar el entorno

Muchas veces, la mejor forma de entender un concepto o de encontrar un error es consultarlo con alguien que comparta nuestra misma perspectiva, pero que tenga una manera de pensar diferente. También, encarar una problemática puede ser mucho más dinámico si se da en forma de conversación entre dos o más personas. Es por eso que consultar o pedir ayuda entre compañeros es una herramienta con un potencial muy alto, que no deben desaprovechar. De igual manera, la constante búsqueda en internet, en libros o en las fuentes que recomendamos en clase también es un mecanismo al cual deben acudir con comodidad durante el desarrollo de este trabajo. Recuerden que lo importante es que comprendan los conceptos con los que están trabajando para que luego los puedan aplicar de la manera correcta, o justificar su utilización.

3.5 No subestimar las consignas

El desarrollo de un script que resuelva un problema que se enuncie en unas pocas oraciones por lo general puede tomar unos minutos, unos días, o varios meses. Entrenar la capacidad de estimar el tiempo que me tomaría desarrollar una solución en código es algo complejo, y se adquiere con la experiencia. Este trabajo práctico, si bien se compone de puntos sencillos, presenta complicaciones que van a poner a prueba un gran número de conceptos complejos con los que trabajamos durante la práctica. Es por eso que comúnmente los estudiantes tienden a subestimar el tiempo que necesitan dedicarle a la resolución, lo que produce dos efectos no deseados: por un lado la falta de tiempo hace que dejen de lado cuestiones de forma y prolijidad en el desarrollo de su código, lo que deviene en trabajos crípticos y complejos de corregir. Por otro lado, dejar las cosas para los ultimos dias/horas produce que saturen los canales de consulta y las respuestas demoren más de lo usual, lo cual enlentece la resolución de conflicto. Con todo esto pretendemos advertirlos de la usual subestimación del tiempo, y les recomendamos no dejarse estar y aprovechar el tiempo de resolución que les ofrecemos.