[MN] Apunte 7 - 2021

November 23, 2021

Métodos numéricos

Segundo cuatrimestre 2021

Apunte 7: Métodos espectrales

Cátedra: Pablo Dmitruk

1 Métodos espectrales

El objeto de esta práctica es el estudio e implementación de métodos espectrales para la resolución de Ecuaciones en Derivadas Parciales (EDP). En los métodos espectrales la idea es obtener aproximantes globales, i.e. que usen los N puntos del dominio, a las funciones de interés y sus derivadas. Si bien, en algunos casos, estos métodos pueden pensarse como el límite de algún esquema de diferencias finitas para orden N, esta interpretación suele resultar poco valiosa en términos prácticos.

Una manera alternativa de pensar a los métodos espectrales es como la proyección de la función de interés sobre un conjunto de funciones ortogonales $\{\phi_0, \phi_1, \ldots, \phi_{M-1}\}$. Luego, si disponemos de la función de interés f evaluada sobre nuestra grilla, es decir, $f_0, f_1, \ldots, f_{N-1}$, podemos escribir el par de transformaciones

$$\hat{f}_m = \alpha_{MN}^m \sum_{n=0}^{N-1} f_n \phi_m^*(x_n), \tag{1}$$

$$f_n = \beta_{MN}^n \sum_{m=0}^{M-1} \hat{f}_m \phi_m(x_n) \tag{2}$$

con α_{MN}^m y β_{MN}^n constantes de normalización apropiadas de forma que la composición de (1) y (2) resulte en la operación identidad y * denota la operación de conjugación. Noten que este par de transformaciones es simplemente una reexpresión de nuestra función mediante una combinación lineal sobre una base ortogonal de $\mathcal{R} \subset \mathbb{R}^{\mathbb{R}}$. Los coeficientes de esta combinación lineal se obtienen sencillamente mediante la proyección sobre cada elemento de la base, i.e. mediante un producto interno.

En principio estas transformaciones que propusimos no parecen representar una ventaja con respecto a una descomposición en series de Taylor que usamos, por ejemplo, para diferencias finitas. Sin embargo, podemos elegir el conjunto $\cup_m \{\phi_m\}$ de forma que: 1. Las ϕ_m sean fácilmente diferenciables/integrables, por ejemplo seleccionando las mismas para que sean autofunciones de algún operador integrodiferencial o bien que sus derivadas puedan obtenerse a partir de alguna relación

de recurrencia. 2. Que la transformación dada por (1) y (2) presente buenas propiedades de convergencia. Es decir, que alcance con pocas autofunciones ϕ_m e información sobre pocos puntos de grilla x_n para obtener buenas aproximaciones a f, f', etc. 3. Que las transformaciones dadas por (1) y (2) puedan ser estimadas de forma computacionalmente eficiente.

Es usual denominar a las cantidades f_n como la representación la función en el *espacio físico*, mientras que para $\hat{f}_m$ se considera a la misma en el *dominio espectral*. Vale resaltar que, en general, $\hat{f}_m$ puede ser complejo, aún cuando f_n sea real, si ϕ_m es una función de $\mathbb{C} \to \mathbb{C}$.

Durante esta práctica nos concentraremos solamente, por simplicidad, en el uso de técnicas espectrales para el cálculo de derivadas en el espacio físico. Como vieron en clases teóricas, estas ideas pueden utilizarse también para evolucionar directamente los $\hat{f}_m$ en la EDP de interés en caso que la misma dependa del tiempo, dando lugar al método de Galerkin. Esto generalmente permite reducir levemente la cantidad de operaciones necesarias.

Empecemos ahora a tratar con conceptos menos abstractos. El método espectral que usaremos en la mayor parte de esta práctica es aquel que surge de escoger $\phi_m(x) = e^{ik_m x}$, con i la unidad imaginaria, i.e. el **método de Fourier**. Hacia el final veremos también el uso de polinomios de Chebyshev $\phi_m(x) = T_m(x)$, dando lugar al **método de Chebyshev**. Estos dos métodos representan el conjunto de técnicas espectrales más populares.

1.1 Método de Fourier

1.1.1 Transformada discreta y FFT

Dada una función con periódo L, cuyos valores conocemos sobre N puntos, vamos a buscar representarla media una cantidad M de funciones trigonométricas (y sus respectivos coeficientes), es decir

$$\hat{f}_m = \alpha^m_{MN} \sum_{n=0}^{N-1} f_n e^{-ik_m x_n}, \tag{3}$$

$$f_n = \beta^n_{MN} \sum_{m=0}^{M-1} \hat{f}_m e^{ik_m x_n}, \tag{4}$$

con $k_m = m2\pi/L$. Esta última elección es para garantizar que todos los elementos de la base tengan período L.

Definidos α^m_{MN} y β^n_{MN}, las expresiones anteriores no son ni más ni menos que una aproximación a la serie de Fourier, y otra para los coeficientes de la misma. Adicionalmente, por motivos computacionales, vamos a considerar el caso donde los x_n son equiespaciados.

Finalmente, dado que la memoria de una computadora es limitada, la cantidad de puntos de grilla x_i y de números de onda k_m deberá ser necesariamente finita. En particular, resulta computacionalmente atractivo el escenario en que $M = N$, ya que permite usar el uso de Transformadas Rápidas de Fourier (o *Fast Fourier Transforms* — **FFT**— en inglés). La FFT es un algoritmo que permite

calcular

$$\hat{f}_m = \sum_{n=0}^{N-1} f_n e^{-i\frac{2\pi}{N}mn}, \tag{TDF}$$

$$f_n = \frac{1}{N}\sum_{m=0}^{N-1} \hat{f}_m e^{i\frac{2\pi}{n}mn}, \tag{ITDF}$$

para todos los $\hat{f}_m$ o todos los f_n en $\mathcal{O}(N\log(N))$ operaciones, en contraposición a las N^2 operaciones que surgen de computar N veces (TDF) o (ITDF) de manera inocente. TDF en este contexto significa Transformada Discreta de Fourier, mientras que ITDF es la transformada discreta inversa de Fourier.

Noten que la equivalencia entre las expresiones (3) (respect. (4)) y la TDF (respect. TDIF) surgen de las condiciones propuestas. En particular, por como definimos el grillado espacial y el espectral tenemos

$$\left.\begin{aligned} x_n &= n\Delta x = n\frac{L}{N} \\ k_m &= m\Delta k = m\frac{2\pi}{L} \end{aligned}\right\} \qquad k_m x_n = \frac{2\pi}{N}mn.$$

Noten que se cancelan las dependencias sobre el espaciamiento de puntos Δx (o el período L) en el cálculo.

La FFT surge de utilizar estrategias de dividir y conquistar al problema de la TDF. Un ejemplo de la familia de técnicas de FFT es el método introducido por Cooley y Tukey (similares a lo hallado en unas notas de Gauss públicadas póstumamente).

1.1.2 Convergencia de la FFT y fenómeno de Gibbs

Mencionamos al inicio que de los métodos espectrales nos va a interesar fuertemente su tasa de convergencia. Dado que ya nos restringimos al estudio de funciones periódicas, sabemos que las mismas siempre pueden representarse[†] mediante una serie de Fourier

$$f(x) = \sum_{m=-\infty}^{\infty} \hat{a}_m e^{i\frac{2\pi}{L}mx}, \qquad \hat{a}_m = \frac{1}{L}\int_0^L f(x)e^{-i\frac{2\pi}{L}mx}\mathrm{d}x.$$

Noten entonces que la ecuación (3) no es ni más ni menos que una aproximación, con una regla trapezoidal, a la expresión para $\hat{a}_m$. No lo mostraremos, pero en general las aproximaciones $\hat{f}_m$ a los coeficientes $\hat{a}_m$ obtenidas mediante la regla tradicional, poseen un muy buen orden de convergencia. Sin embargo, lo que resta analizar es qué tan bien aproxima la serie de Fourier truncada

$$S_M(x) = \sum_{m=-M}^{M} \hat{a}_m e^{i\frac{2\pi}{L}mx}$$

a $f(x)$. Una respuesta rigurosa a esta pregunta excede los contenidos de la materia, sin embargo vamos a establecer y mostrar algunos resultados.

1. $|S_M(x) - f(x)|$ tiende a cero como M^{-P}, con P la primer derivada discontinua de f. En caso que todas las derivadas de f sean continuas, la convergencia irá como M^{-M} (convergencia geométrica).

2. Si $f(x)$ tiene discontinuidades puntuales sobre el conjunto de puntos $\{x_j\}$, entonces S_M converge *lentamente* a $f(x)$ en todos lados, excepto en $\{x_j\}$ donde el error permanece constante para una cantidad finita de modos. Dicho de otra manera, para reproducir discontinuidades se precisan infinitos modos. Puede pensarse en términos de 1. como el caso con $P = 0$ (es decir, ni siquiera la función es continua).
3. Una función f no periódica es equivalente a una función con una discontinuidad en $x = L$. Esto surge naturalmente de considerar la extensión periódica de f. En términos del punto 1, esto significa que si la p-ésima derivada de f no es periódica en $[0, L)$, entonces podemos pensar a dicha derivada como discontinua y la convergencia estará dada por M^{-p}.

Veamos con tres ejemplos las propiedades enunciadas. Para ello, estudiaremos numéricamente la convergencia de series de Fourier que podemos obtener analíticamente. Por simplicidad, trabajaremos en el intervalo $(-\pi, \pi)$, teniendo las funciones un periódo 2π.

- La función $f(x) = \text{sen}(x)/(5 - 3\cos(x))$ es infinitamente periódica y continua. Tiene como serie de Fourier a

$$f(x) = \sum_{m=-\infty m\neq 0}^{\infty} \frac{-i\text{sign}(m)}{3^{|m|+1}} e^{imx},$$

  con sign la función signo.

- La serie de Fourier de $f(x) = x^2$, cuya primer derivada es no periódica, está dada por

$$f(x) = \frac{2\pi^2}{3} + \sum_{m=-\infty m\neq 0}^{\infty} \frac{(-1)^m}{m^2} e^{imx}.$$

- La función rectangular dada por $f(x) = \Theta(x - \pi/2) - \Theta(x + \pi/2)$, con Θ la función de Heaviside, que es discontinua, admite una representación de Fourier a

$$f(x) = \frac{1}{2} + \sum_{m=-\infty m\neq 0}^{\infty} \frac{\text{sen}\left(m\frac{\pi}{2}\right)}{m\pi} e^{imx}.$$

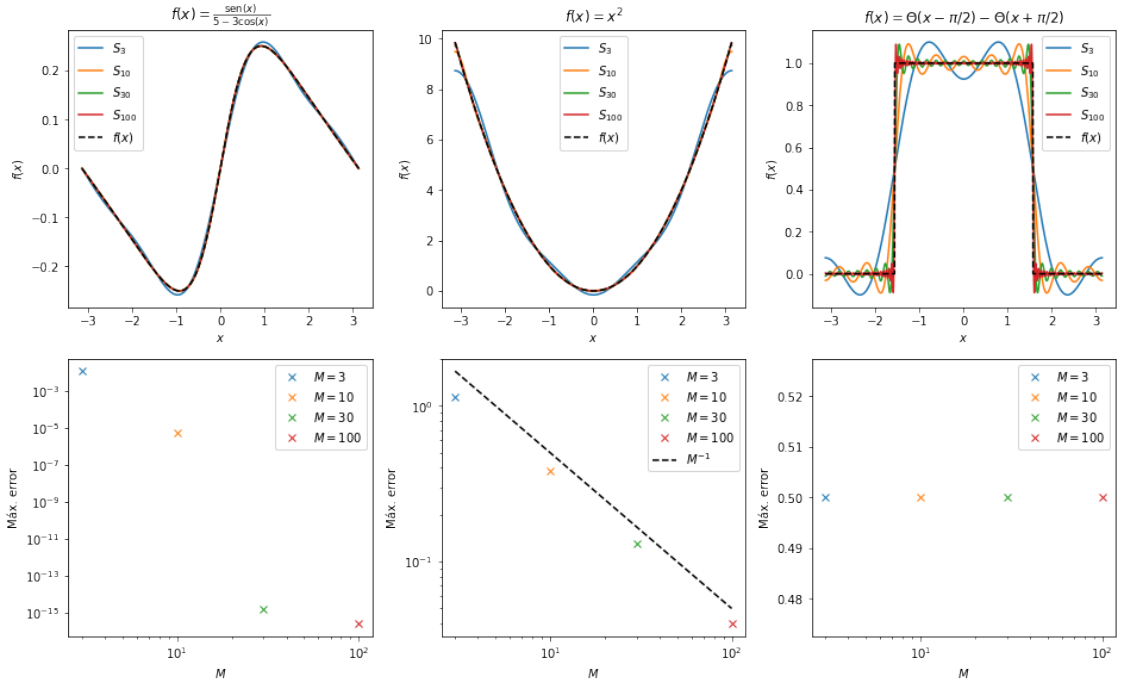
  mediante los coeficientes $\hat{a}_m = \text{sen}(m\pi/2)/(\pi m)$ para $m > 0$ y $\hat{a}_0 = 1/2$.

†: En rigor de verdad hay algunas funciones periódicas que no admiten esta representación. Para el tipo de funciones comunes en problemas físicos estas excepciones no nos resultarán de interés.

[ ]:
```
#@title Cálculo de sumas parciales de Fourier
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter

MS = np.array([3, 10, 30, 100])

x = np.linspace(-np.pi, np.pi, 10000, endpoint=False)

def modo_trig(m):
    if m == 0:
        return 0
    else:
```

```
        return -np.sign(m)*1j/(3**(np.abs(m)+1))

f_trig = np.sin(x)/(5-3*np.cos(x))


def modo_cuad(m):
    if m == 0:
        return np.pi**2/3
    else:
        return 2*(-1)**m/m**2
f_cuad = x**2


def modo_rect(m):
    if m == 0:
        return 1/2
    else:
        return np.sin(m*np.pi/2)/(m*np.pi)

f_rect = np.zeros_like(x)
f_rect[np.abs(x) < np.pi/2] = 1
# Ajusto el valor sobre la discontinuidad para que sea 1/2
mascara_punto_medio = np.not_equal(np.roll(f_rect,-1), f_rect)
f_rect[mascara_punto_medio] = 1/2

fig, axs = plt.subplots(2,3, figsize=(13,8), constrained_layout=True)
for M in MS:
    F_trig = np.zeros_like(x, dtype=np.complex128)
    F_cuad = np.zeros_like(x, dtype=np.complex128)
    F_rect = np.zeros_like(x, dtype=np.complex128)
    for m in range(-M, M+1):
        F_trig = F_trig + modo_trig(m)*np.exp(1j*m*x)
        F_cuad = F_cuad + modo_cuad(m)*np.exp(1j*m*x)
        F_rect = F_rect + modo_rect(m)*np.exp(1j*m*x)

    F_trig = np.real(F_trig)
    error_trig = np.max(np.abs(F_trig - f_trig))
    axs[0,0].plot(x, F_trig, label=f"$S_{{{M}}}$")
    axs[1,0].loglog(M, error_trig, 'x', label=f"$M={M}$")

    F_cuad = np.real(F_cuad)
    error_cuad = np.max(np.abs(F_cuad - f_cuad))
    axs[0,1].plot(x, F_cuad, label=f"$S_{{{M}}}$")
    axs[1,1].loglog(M, error_cuad, 'x', label=f"$M={M}$")

    F_rect = np.real(F_rect)
    error_rect = np.max(np.abs(F_rect - f_rect))
```

```
    axs[0,2].plot(x, F_rect, label=f"$S_{{{M}}}$")
    axs[1,2].semilogx(M, error_rect, 'x', label=f"$M={M}$")

axs[0,0].plot(x, f_trig, "--k", label="$f(x)$")

axs[0,1].plot(x, f_cuad, "--k", label="$f(x)$")
axs[1,1].plot(MS, 5*MS**-1.0, "--k", label="$M^{-1}$")

axs[0,2].plot(x, f_rect, "--k", label="$f(x)$")

[ax.set_xlabel("$x$") for ax in axs[0,:]]
[ax.set_ylabel("$f(x)$") for ax in axs[0,:]]

[ax.set_xlabel("$M$") for ax in axs[1,:]]
[ax.set_ylabel("Máx. error") for ax in axs[1,:]]

[ax.legend() for ax in axs.flatten()]
axs[1,2].yaxis.set_major_formatter(FormatStrFormatter('%.2f'))

axs[0,0].set_title(r"$f(x)=\frac{\mathrm{sen}(x)}{5-3\cos(x)}$")
axs[0,1].set_title(r"$f(x)=x^2$")
axs[0,2].set_title(r"$f(x)=\Theta(x-\pi/2) - \Theta(x+\pi/2)$");
```



Si bien la convergencia (es decir que tan rápido decrecen los $\hat{a}_m$) puede verse directamente de las expresiones, vemos que efectivamente la tasa de convergencia en cada caso obedece las reglas

generales que establecimos arriba.

Para la primer función, vemos una convergencia extremadamente rápida, al punto que para solo 30 modos ya nos encontramos con las limitaciones de la aritmética de punto flotante. Por otro lado, para el caso de la segunda función, que presenta una derivada no periódica, vemos que la convergencia es de primer orden. Finalmente, para el caso discontinuo vemos una convergencia lenta lejos de los puntos de discontinuidad, mientras que en un entorno de los mismos no hay convergencia, como muestra el error constante. Adicionalmente, vemos para este caso la presencia de oscilaciones alrededor de la discontinuidad. Esto es conocido como *fenómeno de Gibbs* o *ringing*. Este tipo de oscilaciones pueden resultarnos de interés porque cuando las encontramos en una solución numérica pueden estar indicando que la solución buscada es discontinua, o bien que precisamos una mayor resolución a efectos de resolver un gradiente de gran magnitud.

Un detalle interesante que podemos ver gráficamente, es que en presencia del fenómeno de Gibbs el salto alrededor de la discontinuidad que obtenemos por utilizar una cantidad finita de modos es $\approx 18\%$ mayor que el que presenta la función real, independientemente de la cantidad de modos utilizados. Noten que todas las curvas coloreadas en la tercer columna presentan escencialmente la misma amplitud cerca de la discontinuidad.

Para quienes quieran un bosquejo de demostración, la tasa de convergencia para funciones *comunes* surge naturalmente de considerar que si $f(x)$ es C^P y $P-1$ veces periódica, entonces integrando por partes tenemos

$$\hat{a}_m = \frac{1}{L}\int_0^L f(x)e^{-i\frac{2\pi}{L}mx}\mathrm{d}x = \frac{1}{L}\left(\frac{L}{2\pi m}\right)^P \int_0^L f^{(P)}(x)e^{-i\frac{2\pi}{L}mx}\mathrm{d}x,$$

donde los términos de borde se anulan P veces por la periodicidad. Luego si $f^{(P)}$ es no periódica la integral resultante irá como $1/m$, dando un orden de convergencia M^{-P} para las sumas parciales.

1.1.3 Cálculo de derivadas con FFT

Luego de establecer qué tipo de funciones vamos a poder representar y qué tan bien funcionarán dichas aproximaciones en la práctica, podemos concentrarnos en nuestro principal interés para el uso de expansiones trigonométricas: el cálculo de derivadas.

Esto va a resultar muy sencillo puesto que las funciones de tipo e^{ikx} son autofunciones del operador de derivación. En otras palabras

$$\frac{\mathrm{d}}{\mathrm{d}x}e^{ikx} = (ik)e^{ikx}.$$

Luego, en caso que nuestra suma de Fourier presente buenas propiedades de convergencia, tendremos que

$$\begin{aligned} f'(x) = \frac{\mathrm{d}}{\mathrm{d}x}\left[f(x)\right] &\approx \frac{\mathrm{d}}{\mathrm{d}x}\left[\sum_{m=-M}^{M}\hat{f}_m e^{ik_m x}\right] \\ &= \sum_{m=-M}^{M}(ik_m\hat{f}_m)e^{ik_m x} = \sum_{m=-M}^{M}\hat{f}'_m e^{ik_m x}. \end{aligned}$$

Es decir, tenemos una serie de Fourier para f', cuyos coeficientes pueden obtenerse a partir de aquellos para f sencillamente como $\hat{f}'_m = ik_m\hat{f}_m$.

La forma práctica de usar esto para calcular derivadas a partir de un conjunto de valores $\mathbf{f} = (f_0, f_1, \ldots, f_{N-1})$ pasando por el dominio espectral es la siguiente: 1. Se calcula la FFT de $\mathbf{f}$, obteniendo el conjunto de coeficientes $\hat{\mathbf{f}} = (\hat{f}_0, \hat{f}_1, \ldots, \hat{f}_{N-1})$. 2. Se arma la grilla espectral $\mathbf{k} = (k_0, k_1, \ldots, k_{N-1})$. 3. Se obtienen los coeficientes para la derivada q-ésima de $\mathbf{f}$ mediante $(i\mathbf{k})^{\{q\}} \circ \hat{\mathbf{f}}$, donde $^{\{\}}$ denota la potenciación elemento a elemento y $\circ$ la multiplicación elemento a elemento. 4. Se calcula la IFFT de $\hat{\mathbf{f}}^{(q)}$, obteniendo la derivada q-ésima en el espacio físico: $\mathbf{f}^{(q)} = (f_0^{(q)}, f_1^{(q)}, \ldots, f_{N-1}^{(q)})$.

Con esto podemos obtener derivadas muy precisas (para funciones cuya serie de Fourier presente convergencia rápida) en $\mathcal{O}(N \log(N))$ operaciones.

Sobre los números de onda

Si bien es posible considerar a los números de onda enteramente positivos, de forma de que discreticen el dominio angular $[0, 2\pi/\Delta x)$ es más común, tanto para la presentación de resultados como para los cálculos, considerar números de onda definidos en $[-\pi/\Delta x, \pi/\Delta x)$. Sin bien esta última forma resulta más simétrica (además que permite visualizar más rápidamente qué modos están correctamente resueltos), deben tener en cuenta que en caso que N sea par se tendrán $N/2$ números de onda negativos y $N/2 - 1$ números de onda positivos. Para el caso de N impar se tienen la misma cantidad de modos negativos y positivos.

1.1.4 FFT en Python

(Esta subsección está adaptada de aquella presentada oportunamente en la Práctica 3)

Resulta extremadamente sencillo calcular la FFT de una cierta señal en Python. Existen numerosos paquetes que realizan esta tarea, sin embargo nos concentraremos en la interfaz brindada por `numpy.fft`.

Para esto podemos usar la función `numpy.fft.fft` que calcula la descomposición de Fourier en una dimensión de un arreglo. En caso que dicho arreglo posea más de una dimensión, podemos seleccionar sobre cuál queremos realizar la transformada mediante el argumento `axis`. Por ejemplo, dado un arreglo `f[i,j]` donde el índice `i` corresponde a distintos tiempos y `j` a distintas posiciones en el espacio, podemos calcular la transformada de Fourier en espacio como `numpy.fft.fft(f, axis=1)`.

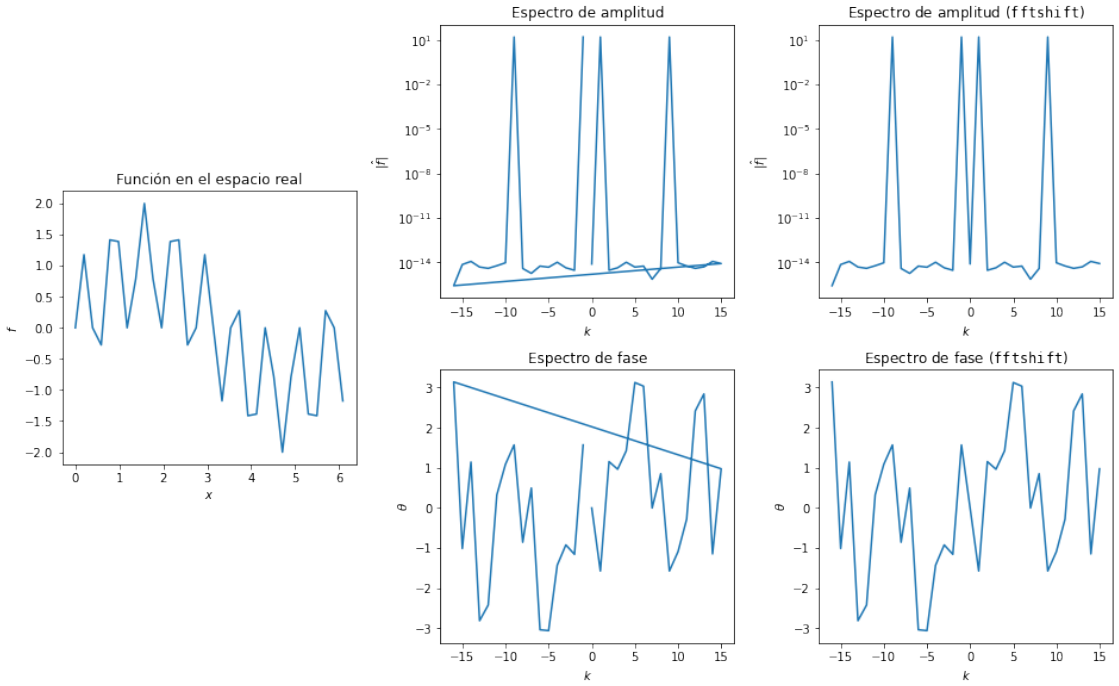
Como notarán, y conforme a lo que mencionamos anteriormente, no es necesario contar con información sobre el espaciamiento entre puntos para calcular la FFT, basta con que sean equiespaciados. Sin embargo, está información sí resulta necesaria para interpretar los valores $\hat{f}_m$ como el contenido espectral asociado al número de onda k_m. Numpy nos ofrece la función `numpy.fft.fftfreq` (donde freq denota frecuencia, el equivalente temporal al número de onda). Por ejemplo, si la señal original (en el espacio x) tenía `N` puntos y un espaciamiento `dx`, podemos obtener el vector $\mathbf{k}$ como `2*numpy.pi*numpy.fft.fftfreq(N, d=dx)`. El factor de 2π interviene para expresar a los números de onda en unidades de rad/m en lugar de 1/m.

Dado un arreglo de N cantidades (reales o complejas), `numpy.fft.fft` devuelve un arreglo de N **cantidades complejas**. Adicionalmente, adopta la convención de que devuelve primero el espectro correspondiente al número de onda $k_m = 0$, seguido de los elementos asociados a $k_m > 0$ y finalmente aquellos asociados a $k_m < 0$. Este ordenamiento, por consistencia, es también seguido por `numpy.ffy.fftfreq`.

Veamos un código que calcula el espectro de amplitud $|\hat{f}(k_m)|$ y de fase $\theta_m = \arctan[\mathrm{Im}\{\hat{f}(k_m)\}/\mathrm{Re}\{\hat{f}(k_m)\}]$ de la función $f(x) = \mathrm{sen}(x) + \mathrm{sen}(9x)$, que es 2π-periódica. Esperamos que la amplitud presente picos para $|k| = 1$ y $|k| = 9$. Veamos que efectivamente es así:

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

N = 32                                                    # Número de puntos
L = 2*np.pi                                               # Tamaño del dominio
x, dx = np.linspace(0, L, N, endpoint=False, retstep=True)  # Grilla espacial
f = np.sin(x) + np.sin(9*x)                               # Función objetivo

k = 2*np.pi*np.fft.fftfreq(N, d=dx)       # Grilla espectral
F = np.fft.fft(f)                         # Transformada de Fourier

amp = np.abs(F)                           # Amplitud
fas = np.arctan2(np.imag(F), np.real(F))  # Fase

# Figura (con un par de ejes abarcando dos filas)
fig = plt.figure(figsize=(13,8), constrained_layout=True)
gs  = gridspec.GridSpec(4, 3, figure=fig)
ax0 = plt.subplot(gs[1:3, 0])
ax1 = plt.subplot(gs[ :2, 1])
ax2 = plt.subplot(gs[ :2, 2])
ax3 = plt.subplot(gs[2: , 1])
ax4 = plt.subplot(gs[2: , 2])

ax0.plot(x, f)
ax0.set_title("Función en el espacio real")
ax0.set_xlabel("$x$")
ax0.set_ylabel("$f$")

ax1.semilogy(k, amp)
ax1.set_title("Espectro de amplitud")
ax1.set_xlabel("$k$")
ax1.set_ylabel("$|\hat f|$")

ax2.semilogy(np.fft.fftshift(k), np.fft.fftshift(amp))
ax2.set_title("Espectro de amplitud ($\mathtt{fftshift}$)")
ax2.set_xlabel("$k$")
ax2.set_ylabel("$|\hat f|$")

ax3.plot(k, fas)
ax3.set_title("Espectro de fase")
ax3.set_xlabel("$k$")
ax3.set_ylabel(r"$\theta$")
```

```
ax4.plot(np.fft.fftshift(k), np.fft.fftshift(fas))
ax4.set_title("Espectro de fase ($\mathtt{fftshift}$)")
ax4.set_xlabel("$k$")
ax4.set_ylabel(r"$\theta$");
```



Noten que en la columna del medio los gráficos presentan una discontinuidad cerca del 0 y una línea un tanto extraña que conecta los valores en los extremos de los intervalos. Esto se debe a que, como mencionabamos antes, Numpy organiza las cantidades de manera que primero aparecen los números de onda positivos y luego los negativos, generando estas discontinuidades al graficar. Para subsanar esto, podemos utilizar la función `numpy.fft.fftshift` que reacomoda los valores para que aparezcan primero aquellos asociados a $k < 0$ y luego aquellos con $k > 0$, quedando $k = 0$ en el medio. Esta operación debemos realizarla tanto para el vector `k` como para el vector `F` (o para cualquier cantidad derivada del mismo), de manera que su orden siga siendo consistente.

Reparen además en que $|f(k_m)|$ es simétrico con respecto al origen, mientras que θ_m es antisimétrico. Esto obedece a una simetría presente en $\hat{f}$ cuando f es real. Existen funciones que aprovechan esta simetría para disminuir el costo de cómputo y de almacenamiento de las FFTs de funciones reales. Estas funcionalidades pueden accederse mediante las interfaces `numpy.fft.rfft` y `numpy.fft.rfftfreq`. Sin embargo, dado que los ahorros que proporcionan no serán determinantes para la realización de esta práctica, no será necesario que las empleen.

Para realizar la transformación inversa, Numpy prove `numpy.fft.ifft`, que se usa de la misma manera que la transformada directa. Noten, sin embargo, que `ifft` devuelve un arreglo de números complejos, por lo que si quieren la parte real del mismo, deben utilizar `numpy.real()`. Veamos un ejemplo:

[ ]:
```
import numpy as np

f  = np.arange(0,15)**2
F  = np.fft. fft(f)
fb = np.fft.ifft(F)

print("Tipo de arreglo después de la IFFT:", fb.dtype)
print("Igualdad entre las partes reales antes y después de transformar:",
      np.all(np.isclose(f, np.real(fb))))
```

```
Tipo de arreglo después de la IFFT: complex128
Igualdad entre las partes reales antes y después de transformar: True
```

1.1.5 Solapamiento (*aliasing*)

Un tópico que no mencionamos hasta el momento es aquel relacionado con la resolución necesaria para poder reproducir fidedignamente una función mediante una serie de Fourier truncada, i.e. mediante una transformada discreta de Fourier.

En cierta manera ya hemos respondido implícitamente a este interrogante. Afirmamos más arriba que los números de onda que podemos resolver con nuestra grilla espectral están dados por $k \in (-\pi/\Delta x, \pi/\Delta x)$. Si pensamos en términos de longitudes de onda, la longitud de onda más corta que podemos resolver λ está dada por

$$\lambda_{\text{min}} = \frac{2\pi}{\max(|k|)} = 2\Delta x.$$

Dicho de otra manera, para poder representar adecuadamente una onda $\text{sen}(x/(2\pi\lambda))$, precisamos contar más de 2 muestras por longitud de onda. Este resultado es conocido como Teorema de muestreo[†].

Veamos computacionalmente que sucede si no cumplimos con este requisito:

[†]: Este problema fue estudiado por Claude Shannon, quien mostró matemáticamente resultados experimentales de telegrafía comunicados previamente por Harry Nyquist. Por ese motivo a veces se nombra a este teorema como Teorema de muestreo de Nyquist-Shannon.

[ ]:
```
#@title Verificación computacional del aliasing
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, FloatSlider, Layout
from IPython.display import display

lamda = 1/8

x, dx = np.linspace(0, 1, 10000, endpoint=False, retstep=True)

f = np.sin(2*np.pi*x/lamda) + 1
F = np.fft.fftshift(np.abs(np.fft.fft(f)))**2/10000**2
k = np.fft.fftshift(2*np.pi*np.fft.fftfreq(x.size, dx))
```

```
def anim(ppw):
    dxm = lamda/ppw
    nxm = int(round(1/dxm))

    xm = np.arange(0, nxm+1)*dxm
    fm = np.sin(xm*2*np.pi/lamda) + 1
    FM = np.fft.fftshift(np.abs(np.fft.fft(fm)))**2/nxm**2
    km = np.fft.fftshift(2*np.pi*np.fft.fftfreq(xm.size, dxm))

    fig, axs = plt.subplots(2, 1, figsize=(12,6), constrained_layout=True)
    fig.suptitle(f"$\lambda = {lamda/dxm:.3f} \Delta x$")
    axs[0].plot(x , f , 'k')
    axs[0].plot(xm, fm, '-x', c="C1")
    axs[0].set_xlabel("$x$")
    axs[0].set_ylabel("$f$")

    axs[1].plot(k , F , 'ok')
    axs[1].plot(km, FM, 'o', c="C1")
    axs[1].set_xlim(-3*np.pi/lamda, 3*np.pi/lamda)
    axs[1].set_xlabel("$k$")
    axs[1].set_ylabel("$|\hat f|^2$")

    plt.show()

slider = FloatSlider(
    value=4,
    min=1,
    max=4,
    step=0.1,
    description="Puntos por longitud de onda",
    continuous_update=True,
    orientation='horizontal',
    readout=True,
    readout_format='.3f',
    layout=Layout(width='auto'),
    style= {'description_width': 'initial'}
)

interact(anim, ppw=slider);
```

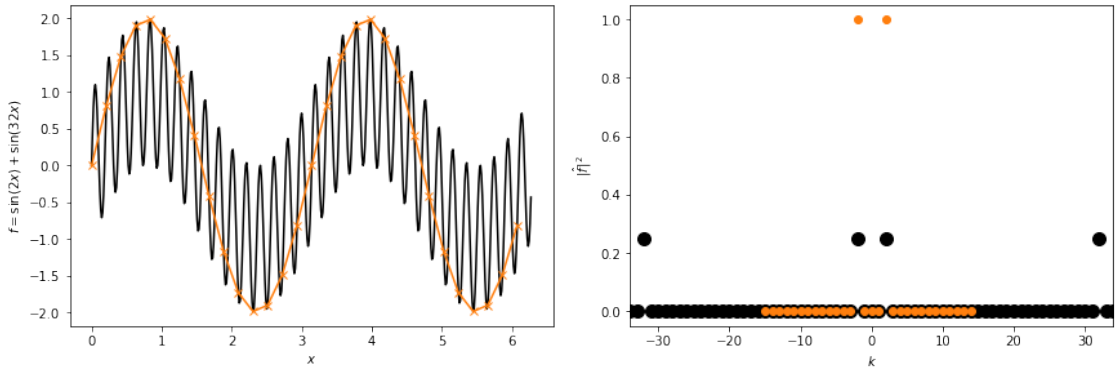
```
interactive(children=(FloatSlider(value=4.0, description='Puntos por longitud de␣
↪onda', layout=Layout(width='a...
```

La figura interactiva permite muestrear una función, en este caso un seno con una cierta longitud de onda, con una cantidad variable de puntos por muestra. El panel superior muestra el efecto del muestreo, mientras que el inferior muestra el espectro de potencias (i.e. la magnitud de los $\hat{f}(k_m)$).

Vemos que si muestreamos nuestra función con más de dos puntos por longitud de onda, el espectro queda correctamente resuelto. En el espacio físico también vemos que una estimación del período de la función naranja coincide con la de la función real (en negro). Sin embargo, cuando empezamos a tener menos de dos muestras por longitud de onda, el período de la curva naranja empieza a ser distinto que el de la curva negra. En el espacio de Fourier, esto lo vemos con picos en el espectro de potencias que en lugar de estar en $k = 50$, aparecen para valores menores de k. En particular, cuando tenemos 3 muestras cada dos longitudes de onda $\Delta x = 3\lambda/2$, vemos una duplicación del período.

Sin embargo, esto es un problema, ya que si no conocemos f, un mal muestreo puede resultar en que veamos energía en escalas que nuestro problema no tiene. Más aún, esta energia *espuria* se vuelve indistinguible de la energía que realmente corresponde a dicho modo. Veamos un ejemplo para la función $f(x) = \text{sen}(2x) + \text{sen}(32x)$, con un muestreo que no alcanza a resolver el modo asociado al segundo término:

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

NX = 30
k  = 2

SS = 16

x1, dx1 = np.linspace(0, 2*np.pi,    NX, endpoint=False, retstep=True)
x2, dx2 = np.linspace(0, 2*np.pi, SS*NX, endpoint=False, retstep=True)

f1 = np.sin(k*x1) + np.sin(16*k*x1)
f2 = np.sin(k*x2) + np.sin(16*k*x2)

F1 = np.fft.fftshift(np.abs(np.fft.fft(f1)))**2/NX**2
F2 = np.fft.fftshift(np.abs(np.fft.fft(f2)))**2/(SS*NX)**2

k1 = np.fft.fftshift(2*np.pi*np.fft.fftfreq(x1.size, dx1))
k2 = np.fft.fftshift(2*np.pi*np.fft.fftfreq(x2.size, dx2))

fig, axs = plt.subplots(1, 2, figsize=(12,4), constrained_layout=True)
axs[0].plot(x2, f2, 'k')
axs[0].plot(x1, f1, '-x', c="C1")
axs[0].set_xlabel("$x$")
axs[0].set_ylabel(f"$f = \mathrm{{sin}}({k}x) + \mathrm{{sin}}({16*k}x)$")

axs[1].plot(k2, F2, 'ok', markersize=10)
axs[1].plot(k1, F1, 'o', c="C1")
axs[1].set_xlim(-17*k, 17*k)
axs[1].set_xlabel("$k$")
axs[1].set_ylabel("$|\hat f|^2$");
```



Si solo tuvieramos la información en naranja, concluiriamos que hay mucha más energía sociada al modo $k = 2$ de la que nuestra señal realmente tiene. Esto es debido a que, por el fenómeno de *aliasing*, se nos está mezclando la energía correspondiente a $k = 2$ con aquella para $k = 32$.

Aliasing no-lineal De la discusión previsa surge que para resolver numéricamente un problema con un método de Fourier[†] debemos contar con una grilla que provea más de dos puntos de grilla por longitud característica del problema. En caso que existan varias longitudes características, debemos representar correctamente la menor de ellas. Para problemas lineales esto implica sencillamente que debemos elegir un valor de N apropiado para nuestra grilla.

Sin embargo, los problemas no-lineales dependientes del tiempo introducen una dificultad adicional, y es que en cada paso se introducen armónicos de orden superior. Para ver fácilmente esto basta considerar $\text{sen}^2(x) = (1 - \cos(2x))/2$. La consecuencia de esto es que, si la integración temporal es suficientemente larga, no importa la resolución que usemos siempre vamos a terminar con energía de armónicos superiores, que no resolvemos correctamente, mezclada en nuestra grilla, por efecto del *aliasing*[‡].

La estrategia para que, dado un cierto número de onda máximo $k_{\text{máx}}$ (o lo que es lo mismo, una cierta resolución espacial N) podamos asegurar que la energía presente en nuestra solución corresponde realmente a los armónicos $(-k_{\text{máx}}, k_{\text{máx}})$ es utilizar un filtro. Lo más sencillo es realizar los cálculos con una resolución mayor a la que nos interesa, N', que dará lugar a un correspondiente $k'_{\text{máx}}$. Luego, si elegimos correctamente $k'_{\text{máx}}$ para que los términos no-lineales queden correctamente resueltos (i.e. no tengan *aliasing*), basta con eliminar todos los modos tales que $|k_{\text{máx}}| < k < |k'_{\text{máx}}|$.

Si bien la solución mencionada requiere un mayor costo computacional (terminamos con grillas espaciales más finas de lo que efectivamente resolvemos), nos permite resolver el solucionar del aliasing de manera **exacta**. Existen otras técnicas que buscan minimizar la amplitud del *aliasing* no-lineal por debajo de un cierto umbral prefijado, no veremos técnicas de este tipo.

Para no-linealidades cuadráticas (como la que aparece en la ecuación de Navier-Stokes incompresible), basta con escoger

$$k'_{\text{máx}} = \frac{3}{2} k_{\text{máx}},$$

esquema conocido como ***regla de los tres medios***. A este filtrado se lo denomina también ***regla de los dos tercios***. El origen de este segundo nombre surge de pensar la misma situación pero al

revés, se tiene una grilla de N puntos con un cierto $k_{\text{máx}}$, para evitar el aliasing debo anular los modos para todos los k tales que

$$|k| > \frac{2}{3} k_{\text{máx}},$$

y por lo tanto las escalas que voy a resolver en mi simulación son aquellas menores a $2k_{\text{máx}}/3$.

La forma de hacer esto computacionalmente es utilizar el cuadrado de los números de onda normalizados

$$\tilde{k}_m^2 = \frac{k_m^2}{\max(k_m^2)}.$$

Luego basta con imponer

$$\hat{f}_m = 0 \qquad \forall\, m : \tilde{k}_m^2 > \frac{4}{9}.$$

Esta tarea puede realizarse fácilmente en Python mediante el uso de máscaras. Por ejemplo, mediante el código

```
kn = k**2 / np.max(np.abs(k))**2
F[kn > 4/9] = 0
```

donde `k` es un vector que contiene a los números de onda y `F` uno complejo que contiene a una serie de coeficientes $\hat{f}_m$.

Finalmente, vale resaltar que el aliasing no-lineal no afecta a los métodos de tipo Galerkin (aquellos que resuelven solo en el dominio espectral, sin pasar por el dominio real). Sin embargo, en los mismos, el costo de calcular términos no-lineales escala como $\mathcal{O}(N^\ell)$, con ℓ el grado de la no-linealidad, mientras que en el método pseudo-espectral (que calcula no-linealidades en el espacio físico), el costo es $\ell\mathcal{O}(N \log N)$.

† y ‡: Si bien no lo hicimos explicito, limitaciones similares afectan a los métodos de diferencias finitas o volúmenes finitos.

1.1.6 Series de Fourier en 2D

Las ideas mencionadas anteriormente resultan fáciles de generalizar a dos dimensiones espaciales. Vamos a querer expresar nuestra función como

$$f_{pq} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \hat{\hat{f}}_{mn} e^{-i(k_m^x x_p + k_n^y y_q)} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \hat{\hat{f}}_{mn} e^{-i\boldsymbol{k}_{mn}\cdot \boldsymbol{r}_{pq}},$$

con $\boldsymbol{r} = (x, y)$ el vector posición y $\boldsymbol{k}$ el vector número de onda (k^x, k^y). Naturalmente, la discretización en cada dirección es independiente, resultando $k_m^x = m2\pi/L_x$ y $k_n^y = n2\pi/L_y$.

El proceso para obtener los coeficientes $\hat{\hat{f}}_{mn}$ podemos pensarlo como un proceso escalonado, donde primero obtenemos, por ejemplo, $\hat{f}_m(y_n)$ y luego $\hat{\hat{f}}_{mn}$, como indica el siguiente esquema:

$$f(x_p, y_q) \quad \overset{\text{TDF en } x}{\to} \quad \hat{f}(k_m^x, y_m) \quad \overset{\text{TDF en } y}{\to} \quad \hat{\hat{f}}(k_m^x, k_n^y).$$

Para alivianar la notación, en adelante utilizaremos un único ˆ para denotar coeficientes de Fourier, mientras que la cantidad de transformadas realizadas se desprenderá de los índices que acompañan a $\hat{f}$.

Sobre la convergencia, solo mencionaremos que siguen valiendo las mismas observaciones generales que realizamos para el caso 1D.

Derivadas en 2D Resulta muy sencillo obtener derivadas a partir de $\hat{f}_{nm}$. Para ello, vamos a usar las siguientes representaciones matriciales de $N_x \times N_y$

$$\hat{\mathbb{F}} = \begin{pmatrix} \hat{f}_{0,0} & \hat{f}_{0,1} & \cdots & \hat{f}_{0,N_y-1} \\ \hat{f}_{1,0} & \hat{f}_{1,1} & \cdots & \hat{f}_{1,N_y-1} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{f}_{N_x-1,0} & \hat{f}_{N_x-1,1} & \cdots & \hat{f}_{1,N_y-1} \end{pmatrix}$$

$$\mathbb{K}_x = \begin{pmatrix} k_0^x & k_0^x & \cdots & k_0^x \\ k_1^x & k_1^x & \cdots & k_1^x \\ \vdots & \vdots & \ddots & \vdots \\ k_{N_x-1}^x & k_{N_x-1}^x & \cdots & k_{N_x-1}^x \end{pmatrix} = \begin{pmatrix} | & | & \cdots & | \\ \mathbf{k}^x & \mathbf{k}^x & \cdots & \mathbf{k}^x \\ | & | & \cdots & | \end{pmatrix}$$

$$\mathbb{K}_y = \begin{pmatrix} k_0^y & k_1^y & \cdots & k_{N_y-1}^y \\ k_0^y & k_1^y & \cdots & k_{N_y-1}^y \\ \vdots & \vdots & \ddots & \vdots \\ k_0^y & k_1^y & \cdots & k_{N_y-1}^y \end{pmatrix} = \begin{pmatrix} — & \mathbf{k}^y & — \\ — & \mathbf{k}^y & — \\ \vdots & \vdots & \vdots \\ — & \mathbf{k}^y & — \end{pmatrix}$$

Luego, el resultado de derivar r veces con respecto a x y s veces con respecto a y puede expresarse en el dominio espectral como

$$\hat{\mathbb{F}}^{(r)(s)} = (i\mathbb{K}^x)^{\{r\}} \circ (i\mathbb{K}^y)^{\{s\}} \circ \hat{\mathbb{F}}$$

y antitransformando $\hat{\mathbb{F}}^{(r)(s)}$ se obtiene $\mathbb{F}^{(r)(s)}$, que es la matriz que contiene a los elementos $(\partial_x^r \partial_y^s f)_{pq}$.

Vale resaltar que en caso que solo se busque derivar con respecto a x o a y, basta con realizar transformadas y antitransformadas solo en la dirección deseada, procediéndose a derivar en el espacio mixto. Por ejemplo, en caso que se desee derivar solo con respecto a x, puede hacerse en el dominio $\hat{f}(k^x, y)$. Esto permite disminuir el computo necesario. Sin embargo en caso que no quede claro como realizarlo, no se preocupen y realicen las derivadas en el dominio espectral $\hat{f}(k^x, k^y)$, luego de hacer una transformada de Fourier 2D.

Aliasing no-lineal en 2D Para remover el aliasing no-lineal en dimensiones superiores, por ejemplo para el caso 2D, podemos sencillamente utilizar el mismo concepto de número de onda normalizado pero ahora considerando cada componente de $\boldsymbol{k}$:

$$\tilde{k}_{mn}^2 = \frac{(k_m^x)^2}{\max[(k_m^x)]} + \frac{(k_m^y)^2}{\max[(k_m^y)^2]},$$

y filtrando de forma tal que

$$\hat{f}_{mn} = 0 \qquad \forall\, (m,n) : \tilde{k}_{mn}^2 > \frac{4}{9}.$$

FFT 2D en Python Para calcular la serie de Fourier 2D en Python podemos sencillamente utilizar dos transformadas 1D encadenadas, variando el argumento `axis` de `numpy.fft.fft`. Por ejemplo, dado un arreglo `f[x,y]` podríamos hacer algo como lo siguiente:

```
# Obtener F = f(kx,ky)
F = numpy.fft.fft(numpy.fft.fft(f, axis=0), axis=1)
```

Sin embargo, las interfaces de `numpy.fft` tienen una función que realiza la transformada de Fourier 2D sobre el par de ejes solicitados. La misma puede accederse mediante `numpy.fft.fft2` y admite un argumento `axes` que permite especificar sobre que par de ejes desea realizarse la transformada de Fourier. Por ejemplo, si tenemos un arreglo `f[t,x,y]` y queremos obtener la transformada de Fourier 2D espacial para cada tiempo, podemos usar sencillamente

```
# Obtener F = f(t, kx, ky)
F = numpy.fft.fft2(f, axes=(1,2))
```

Para los números de onda, deberemos generar cada vector por separado, por ejemplo mediante

```
kx = np.fft.fftfreq(Nx, dx)
ky = np.fft.fftfreq(Ny, dy)
```

1.2 Método de Chebyshev

Como mostramos anteriormente, las series de Fourier son una herramienta poderosa para el desarrollo de métodos numéricos. Sin embargo, a diferencia de lo que sucede en el tratamiento analítico, su uso computacional requiere emplear una cantidad finita de modos. Esto deteriora la calidad de las aproximaciones en caso que nuestro problema no tenga una buena cantidad de derivadas suaves y periódicas.

Por otro lado, muchos problemas físicos presentan condiciones de contorno que no son periódicas. Esto no quiere decir que no haya física interesante para estudiar en problemas periódicos. Por ejemplo, en el estudio de flujos turbulentos en regiones suficientemente lejos de paredes puede ser apropiado y computacionalmente atractivo el uso de recintos periódicos.

Sin embargo, hay problemas que requieren inexorablemente resolver la dinámica del sistema cerca de algún contorno físico. La formación de una capa límite es un ejemplo de esto. Veamos una forma posible de estudiar este tipo de problemas con métodos espectrales.

1.2.1 Transformada de Chebyshev

Una forma de utilizar la teoría de Fourier para el tratamiento de problemas con contornos, es sencillamente mediante un cambio de variables. Para ello, recordemos primeramente que el principal inconveniente del método de Fourier para este tipo de problemas es el deterioro en la convergencia para representar funciones no periódicas.

No obstante, si queremos determinar $f(x)$ en el intervalo $[-1,1]$, con f arbitraria, podemos reformular el problema para hallar $f(\cos(\theta))$ para $\theta \in [0,\pi]$. La composición con el coseno hace que $f(\cos(\theta))$ sea periódica y por tanto podrá representarse mediante una serie de Fourier truncada de manera precisa siempre que f sea suave. Naturalmente, en caso que x esté definida sobre otro intervalo, podemos reescribir el problema sobre el intervalo $[-1,1]$ mediante un cambio de variables adicional.

Expresando entonces $f(\cos(\theta))$ mediante una serie de Fourier truncada, tenemos

$$f(\cos(\theta)) = \sum_{m=0}^{M} \hat{f}_m e^{im\theta} = \sum_{m=0}^{M} \hat{f}_m \cos(m\theta), \tag{5}$$

donde usamos que $f(\cos(\theta))$ tiene período 2π (y por tanto el número de onda $k_m = m$) y también que es par, por lo que la parte imaginaria de la suma debe necesariamente se anula. Los coeficientes $\hat{f}_m$ quedan dados por

$$\hat{f}_m = \frac{2}{\pi c_m} \int_0^{\pi} f(\cos(\theta)) \cos(m\theta)\, \mathrm{d}\theta, \tag{6}$$

con $c_m = 1$ excepto por $c_0 = c_M = 2$.

Dado que el cambio de variables escogido es inversible, podemos escribir $\theta = -\arccos(x)$ y entonces tenemos una expresión en términos de nuestra variable de interés x

$$\begin{aligned} f(x) &= \sum_{m=0}^{M} \hat{f}_m \cos(m \arccos(x)), \\ \hat{f}_m &= \frac{2}{\pi c_m} \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} f(x) \cos(m \arccos(x))\, \mathrm{d}x. \end{aligned}$$

Puede mostrarse que el conjunto $\{\cos(m \arccos(x)),\ \forall m \in \mathbb{N}\}$ corresponde a un conjunto de polinomios ortogonales con respecto a un producto interno con peso $(1-x^2)^{-1/2}$. Estos polinomios reciben el nombre de polinomios de Chebyshev (o cualquier otra grafía de su elección) de primera especie, y se los nota como $T_m(x)$.

La transformada directa e inversa de Chebyshev queda entonces definida como

$$\begin{aligned} f(x) &= \sum_{m=0}^{M} \hat{f}_m T_m(x), \\ \hat{f}_m &= \frac{2}{\pi c_m} \int_{-1}^{1} \frac{1}{\sqrt{1-x^2}} f(x) T_m(x)\, \mathrm{d}x. \end{aligned}$$

Solo utilizando un cambio de variables, logramos escribir nuestro problema en términos de una serie con buenas propiedades de convergencia para funciones suaves y no-periódicas.

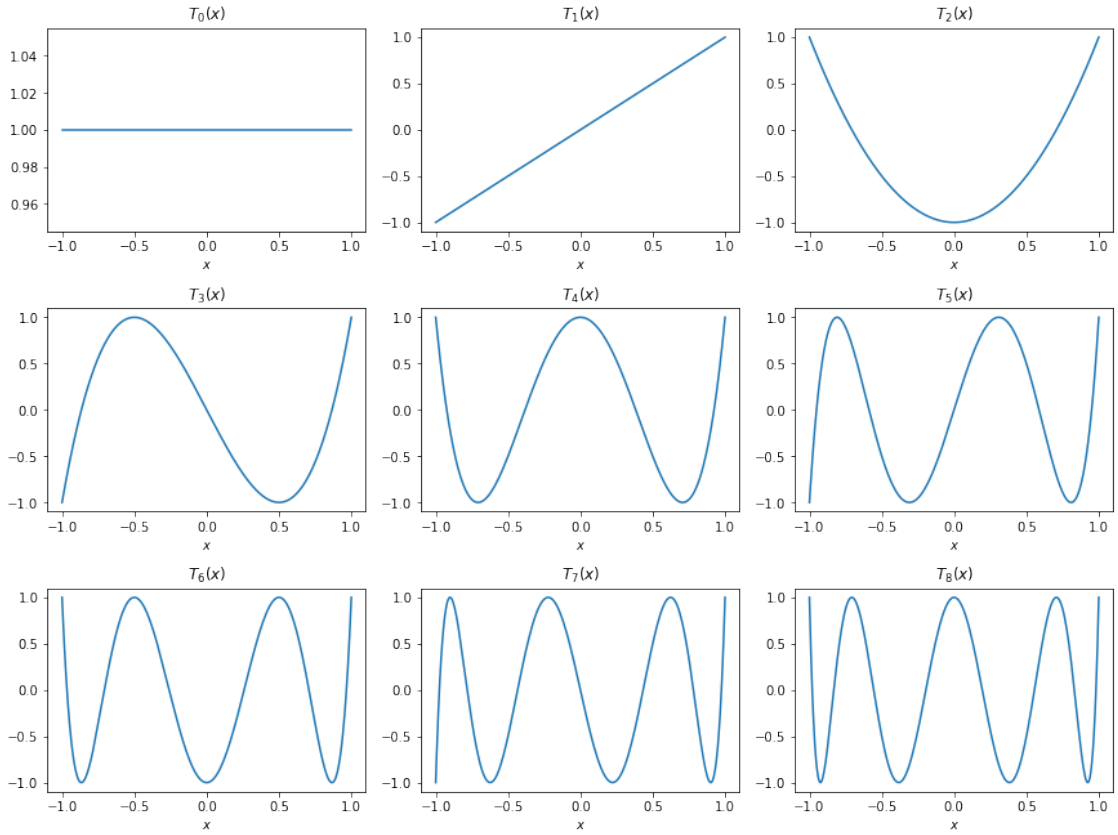
1.2.2 Algunas propiedades de los polinomios de Chebyshev

Dado su conexión con los polinomios trigonométricos, los polinomios de Chebyshev presentan algunas relaciones de recurrencia y diferenciación muy útiles:

- $T_0(x) = 1$;
- $T_1(x) = x$;
- $T_m(-x) = (-1)^m T_m(x)$;
- $T_{m+1}(x) = 2xT_m(x) - T_{m-1}(x)$;
- $T_m(x)T_{m'}(x) = \frac{1}{2}\left[T_{m+m'}(x) + T_{m-m'}(x)\right]$;
- $(1-x^2)T'_m(x) = \frac{m}{2}\left[T_{m-1}(x) - T_{m+1}(x)\right]$;
- $T_m(x) = \frac{1}{2}\left[\frac{T'_{m+1}(x)}{m+1} - \frac{T'_{m-1}(x)}{m-1}\right]$.

Y grafiquemos también algunos polinomios, por completitud.

[ ]:
```python
# Gráfico de polinomios de Chebyshev
import numpy as np
import matplotlib.pyplot as plt

def polinomio_chebyshev(n):
    """Devuelve una función que computa el polinimio de Chebyshev de grado N."""
    if n == 0:
        return lambda x: np.ones_like(x)
    if n == 1:
        return lambda x: x
    else:
        return lambda x: 2*x*polinomio_chebyshev(n-1)(x) \
                         - polinomio_chebyshev(n-2)(x)

N = 256
theta = np.linspace(0, np.pi, N)
x     = np.cos(theta)

fig, axs = plt.subplots(3, 3, figsize=(12,9), constrained_layout=True)
axs      = axs.flatten()
for i, ax in enumerate(axs):
    ax.set_title(f"$T_{i}(x)$")
    ax.plot(x, polinomio_chebyshev(i)(x))
    ax.set_xlabel("$x$")
```



1.2.3 Cálculo de derivadas a partir de una serie de Chebyshev

Usando las propiedades introducidas, podemos obtener $f'(x)$ a partir de $\hat{f}_m$. Para ello, consideremos

$$f'(x) \approx \sum_{m=1}^{M} \hat{f}_m T'_m(x) = \sum_{m=0}^{M} \hat{f}_m^{(1)} T_m(x), \qquad \hat{f}_M^{(1)} = 0.$$

El segundo miembro surge de derivar término a término la serie de Chebyshev para f. Notar que como $T_m(x)$ es un polinomio de grado m, desaparece el término asociado a $m = 0$. Por otro lado, el tercer miembro surge de considerar que f' tendrá su propia representación en términos de $T_m(x)$. Sin embargo, es necesario anular $\hat{f}_M^{(1)}$, para que los grados a ambos lados de la igualdad coincidan. Utilizando las propiedades mencionadas anteriormente, tenemos entonces

$$\begin{aligned}\sum_{m=1}^{M} \hat{f}_m T'_m(x) &= \hat{f}_0^{(1)} + \hat{f}_1^{(1)} T_1(x) + \sum_{m=2}^{M-1} \frac{\hat{f}_m^{(1)}}{2} \left[\frac{T'_{m+1}}{m+1} - \frac{T'_{m-1}}{m-1} \right] \\ &= \frac{\hat{f}_{M-1}^{(1)}}{2M} + \sum_{m=1}^{M-2} \frac{1}{2m} \left(c_{m-1} \hat{f}_{m-1}^{(1)} - \hat{f}_{m+1}^{(1)} \right) T'_m,\end{aligned}$$

donde en la última expresión procedimos a ordenar la suma de acuerdo al grado de $T'_m(x)$, utilizando $T'_1(x) = 1$ y $T'_2(x) = 4T_1(x)$. Comparando los términos izquierdo y derecho de esta

expresión, y utilizando que los T'_m son ortogonales[†], obtenemos la relación de recurrencia

$$\hat{f}_M^{(1)} = 0, \qquad \hat{f}_{M-1}^{(1)} = 2M\hat{f}_M, \qquad \hat{f}_{m-1}^{(1)} = \frac{2m\hat{f}_m + \hat{f}_m^{(1)}}{c_{m-1}}, \tag{7}$$

que permite obtener los coeficientes de la serie de Chebyshev de f' a partir de los de f. Notar que se obtienen primero los coeficientes asociados a los polinomios de mayor grado. Adicionalmente, recuerden que $c_m = 1$ excepto por $c_0 = c_M = 2$.

[†]: Es fácil probarlo a partir de las relaciones de la sección previa.

1.2.4 Transformada discreta de Chebyshev

Resolver un cierto problema diferencial de manera numérica va a implicar conocer f sobre una cantidad discreta de puntos $f(x_n) = f_n$. Sin embargo, dado que queremos representar $f(\cos\theta)$ mediante el par de transformadas (5) y (6), vamos a precisar que θ se distribuya de manera equiespaciada. De esta manera, los coeficientes $\hat{f}_m$ pueden aproximarse mediante una regla trapezoidal de manera precisa.

Si contamos con $N+1$ puntos en el espacio físico, entonces $\theta_n = n\pi/N$ y el siguiente esquema discreto

$$\hat{f}_m = \frac{2}{c_m}\sum_{n=0}^{N}\frac{1}{c_n} f_n \cos\left(\frac{\pi m n}{N}\right), \tag{TDC}$$

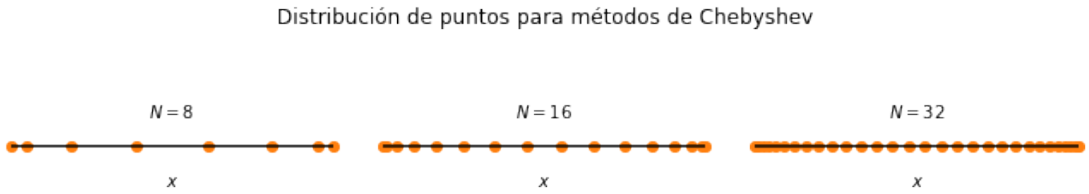
$$f_n = \sum_{m=0}^{N} \hat{f}_m \cos\left(\frac{\pi m n}{N}\right), \tag{ITDC}$$

será una buena aproximación al problema continuo presentado en la sección previa. Por analogía con la transformada de Fourier, vamos a llamar a estas expresiones la transformada discreta de Chebyshev (TDC) y transformada discreta inversa de Chebyshev (ITDC)

Obtuvimos una representación en serie que converge rápido, puede computarse en poca cantidad de operaciones y que, como veremos, es fácil de derivar. Vale mencionar, no obstante, que el costo que pagamos por ello es que ahora $f_n = f(x_n) = f(\cos(\theta_n))$ ya no queda determinado sobre una grilla equiespaciada. Más aún, podemos estimar fácilmente que si $\Delta_0 x$ es el espaciamiento típico cerca del origen, en los bordes tendremos un espaciamiento de orden $(\Delta_0 x)^2$. Veamos como quedan distribuidos estos puntos, llamados de Gauss-Lobatto (o a veces de Chebyshev-Gauss-Lobatto).

[ ]:
```
# Distribución de puntos de Gauss-Lobato
import numpy as np
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 3, figsize=(9,2), constrained_layout=True)
fig.suptitle("Distribución de puntos para métodos de Chebyshev")
for i, N in enumerate([8, 16, 32]):
    theta = np.linspace(0, np.pi, N)
    x     = np.cos(theta)

    axs[i].plot(x, np.ones_like(x), 'o', color="C1", zorder=-i)
```

```
    axs[i].axis("off")
    axs[i].hlines(1, x[0], x[-1])
    axs[i].text(0, 1.02, f"$N={N}$", va="center", ha="center")
    axs[i].text(0, 0.98,      "$x$", va="center", ha="center")
```



Esta distribución de puntos puede ser deseable en algunos problemas que desarrollan dinámicas en escalas particularmente pequeñas cerca de los bordes. El estudio de efectos peliculares en electromagnetismo, es un ejemplo con estas características. Sin embargo, para problemas de tipo parabólico, esto quiere decir que la condición de CFL se volverá particularmente restrictiva. Por ejemplo, para una ecuación de difusión, la condición de CFL establece que $\Delta t \propto (\Delta_0 x)^4$. Es usual que métodos basados en representaciones de Chebyshev integren los términos parabólicos de una EDP mediante esquemas implícitos para sortear este inconveniente.

1.2.5 Matriz de diferenciación de Chebyshev

Una forma de utilizar el método de Chebyshev para calcular derivadas dado f_j (con x_j siguiendo una distribución de Gauss-Lobatto) es haciendo algo similar a lo que hacíamos mediante el método de Fourier. Comenzaremos por obtener los $\hat{f}_m$. Para ello se pueden utilizar transformadas rápidas, pero por motivos pedagógicos consideremoslo primero mediante una cierta matriz de proyección al dominio espectral $\mathbb{P}^e$. Notar que basta definir

$$\mathbb{P}^e = \frac{1}{N}\begin{pmatrix} 1/4 & 1/2 & 1/2 & \dots & 1/2 & 1/4 \\ 1/2 & \cos(\pi/N) & \cos(2\pi/N) & \dots & \cos((N-1)\pi/N) & -1/2 \\ 1/2 & \cos(2\pi/N) & \cos(4\pi/N) & \dots & \cos(2(N-1)\pi/N) & 1/2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1/2 & \cos((N-1)\pi/N) & \cos((N-1)2\pi/N) & \dots & \cos((N-1)^2\pi/N) & (-1/2)^{N-1} \\ 1/4 & -1/2 & 1/2 & \dots & (-1/2)^{N-1} & (-1/4)^N \end{pmatrix}$$

, para obtener $\hat{\mathbf{f}} = (\hat{f}_0, \hat{f}_1, \dots, \hat{f}_N)$ como

$$\hat{\mathbf{f}} = \mathbb{P}^e \hat{\mathbf{f}}.$$

Por otro lado, la relación de recurrencia hallada en la sección previa, también puede representarse

matricialmente, de forma que

$$\hat{\mathbf{f}}^{(1)} = \mathbb{D}^e \hat{\mathbf{f}}, \qquad \mathbb{D}^e = \begin{pmatrix} 0 & 1 & 0 & 3 & 0 & 5 & \dots & N-1 & 0 & N \\ 0 & 0 & 4 & 0 & 8 & 0 & \dots & 0 & 2(N-1) & 0 \\ 0 & 0 & 0 & 6 & 0 & 10 & \dots & 4(N-1) & 0 & 2N \\ 0 & 0 & 0 & 0 & 8 & 0 & \dots & 0 & 2(N-1) & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 4(N-1) & 0 & 2N \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 2(N-1) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 2N \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \end{pmatrix},$$

con $\mathbb{D}^e$ la matriz de diferenciación en el dominio espectral (de Chebyshev). Notar que asumimos N par (caso contrario la primer fila termina con un 0).

Finalmente para obtener la derivada buscada $\mathbf{f}'$ podemos realizar proyectar los $\hat{\mathbf{f}}^{(1)}$ al espacio físico como

$$' = \mathbb{P}^f \hat{\mathbf{f}}^{(1)},$$

con

$$\mathbb{P}^f = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ 1 & \cos(\pi/N) & \cos(2\pi/N) & \dots & \cos((N-1)\pi/N) & -1 \\ 1 & \cos(2\pi/N) & \cos(4\pi/N) & \dots & \cos(2(N-1)\pi/N) & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & \cos((N-1)\pi/N) & \cos((N-1)2\pi/N) & \dots & \cos((N-1)^2\pi/N) & (-1)^{N-1} \\ 1 & -1 & 1 & \dots & (-1)^{N-1} & (-1)^N \end{pmatrix}.$$

Todas estas operaciones podemos sintetizarlas en una única matriz de diferenciación de Chebysheb en el espacio físico $\mathbb{D}^f = \mathbb{P}^f \mathbb{D}^e \mathbb{P}^e$ que verifica

$$\mathbf{f}' = \mathbb{D}^f \mathbf{f}.$$

Puede obtenerse una forma cerrada para D_s, que queda definida como

$$[\mathbb{D}^f]_{ij} = \begin{cases} \dfrac{2N^2+1}{6} & \text{para} \quad i = j = 0, \\ \dfrac{-x_j}{2(1-x_j^2)} & \text{para} \quad i = j = 1, \dots, N-1, \\ -\dfrac{2N^2+1}{6} & \text{para} \quad i = j = N, \\ \dfrac{c_i}{c_j} \dfrac{(-1)^{i+j}}{x_i - x_j} & \text{para} \quad i \neq j. \end{cases}$$

Noten que esta matriz fue construida para $x_n = \cos(\theta_n)$ y por tanto opera sobre el vector $\mathbf{x} = (1, \dots, -1)$. Si queremos una matriz que opera con $\mathbf{x}$ y $\mathbf{f}$ ordenados en sentido creciente, basta con utilizar la regla de la cadena, lo que implica que debemos multiplicar esta matriz por -1.

Adjuntamos en el apunte una función matriz_diferenciacion_chebyshev que calcula una matriz de diferenciación de este tipo dada la cantidad de puntos de grilla $N+1$ y que pueden usar en los ejercicios.

Matrices para derivadas de órdenes superiores Es inmediato mostrar que para obtener matrices que estimen derivadas de órdenes superiores basta con elevar $\mathbb{D}^f$ a una potencia igual al orden de la derivada que se desea estimar. Por ejemplo, la matriz que permite obtener la derivada segunda $\mathbb{D}^{(2)f}$ puede obtenerse como

$$\mathbf{f}'' \approx \mathbb{D}^{(2)f}\mathbf{f} = (\mathbb{D}^f)^2\mathbf{f}.$$

1.2.6 Condiciones de Dirichlet en la matriz de diferenciación de Chebyshev

Una ventaja de la formulación matricial es que para incorporar condiciones de contorno en nuestro problema, podemos proceder de manera completamente análoga a lo realizado en la práctica 5. La idea es que $\mathbb{D}^f$ calcula la derivada en todos los puntos del dominio, incluido los bordes.

Por ejemplo, dado un problema de Poisson en el que conocemos f_0 y f_N, podemos escribir el siguiente sistema de ecuaciones

$$\tilde{\mathbb{D}}^{(2)f}\mathbf{f} = \tilde{\mathbf{q}},$$

con

$$\tilde{\mathbb{D}}^{(2)f} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ \hline & & & [\mathbb{D}^{(2)f}]_{1:N-1} & & & \\ & & & & & & \\ \hline 0 & 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix}, \qquad \tilde{\mathbf{q}} = \begin{pmatrix} f_0 \\ q_1 \\ q_2 \\ \vdots \\ q_{N-1} \\ f_N \end{pmatrix},$$

donde $[\mathbb{D}^{(2)f}]_{1:N-1}$ representa las filas 1 a $N-1$ de $\mathbb{D}^{(2)f}$ y q_n es el forzante de la ecuación sobre el punto x_n.

De manera análoga podríamos tratar contornos de tipo Neumann, utilizando la primer y última fila de $\mathbb{D}^f$ en lugar de $(1\,0\,\dots\,0)$ y $(0\,0\,\dots\,1)$ respectivamente, con el correspondiente cambio en el primer y último elemento de $\tilde{\mathbf{q}}$.

1.2.7 Transformada rápida de Chebyshev

Si bien en esta práctica vamos a usar matrices de diferenciación para tratar problemas con el método de Chebyshev, esto es poco usual, ya que requiere $\mathcal{O}(N^2)$ operaciones para un problema 1D.

Existe otra opción que es proceder de manera completamente análoga al caso de Fourier, puesto que la TDC y la ITDC pueden computarse con esquemas rápidos en $\mathcal{O}(N\log(N))$. Para ello podemos valernos de algoritmos de Transformada Coseno Rápida (FCT) que una gran cantidad de bibliotecas numéricas implementan, entre otras Scipy.

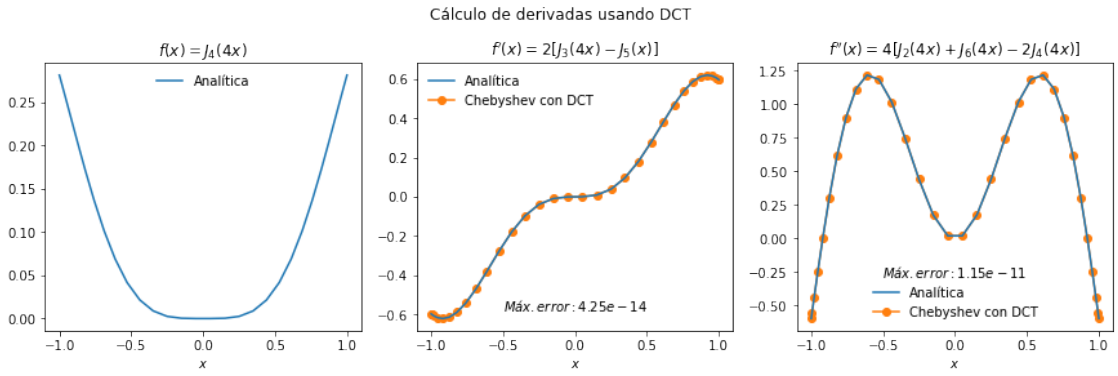
Luego, si contamos con una forma rápida de obtener los coeficientes $\hat{f}_m$, podemos usar la relación de recurrencia (7) para obtener los coeficientes de la derivada en $\mathcal{O}(N)$ operaciones. En caso que deseemos obtener derivadas de orden superior, podemos aplicar sucesivamente la misma regla de recurrencia, o derivar una para el orden de derivación buscado.

Veamos un código que realiza esta tarea:

[ ]:
```
!pip install --upgrade scipy>=1.6.1
import numpy as np
```

```python
import matplotlib.pyplot as plt
from scipy.special import jv

def derivada_chebyshev_fct(f, o=1):
    import scipy.fft as spfft
    """
        Computa la derivada de orden arbitrario de f usando un método rápido
    de Chebyshev. f debe estar definida sobre puntos x_i = cos[(N-i-1)/(N-1)],
    es decir sobre una grilla de Gauss-Lobatto ordenada de manera creciente.

    Entrada:
        -`f`:  Vector (N) con los valores de una cierta función sobre los
           puntos de Gauss-Lobatto.
        -`o`:  Orden de la derivada buscada.

    Salida:
        -`fp`: Vector (N) con los valores de la o-ésima derivada sobre la grilla
           de Gauss-Lobatto.
    """
    N  = f.size
    F  = spfft.dct(f, type=1, norm="forward")
    Fp = np.zeros_like(F)
    # Hago la recurrencia o veces
    for i in range(1, o+1):
        # Trato los términos iniciales
        Fp[-i-1] = 2*(N-i)*F[-i]
        Fp[-i]   = 0
        # Aplico la recurrencia
        for j in range(N-i-2, -1, -1):
            Fp[j] = 2*(j+1)*F[j+1] + Fp[j+2]

        # Guardo los coeficientes para la derivada en F. Y vuelvo a iterar si
        # preciso una derivada de orden superior.
        F = Fp.copy()

    fp = spfft.idct(F, type=1, norm="forward")
    return (-1)**o * fp

N  = 32                                          # Cantidad de puntos
th = np.linspace(0, np.pi, N)                    # Grilla equiespaciada
x  = - np.cos(th)                                # Grilla de Gauss-Lobatto

f   = jv(4, 4*x)                                 # Función a derivar
fp  = 2*(jv(3,4*x) - jv(5,4*x))                  # Derivada analítica
fpp = 4*(jv(2,4*x) + jv(6,4*x) - 2*jv(4,4*x))    # Derivada segunda analítica

fc  = derivada_chebyshev_fct(f)                  # Derivada con Chebyshev
```

```
fcc = derivada_chebyshev_fct(f, o=2)            # Derivada segunda con Chebyshev

err_p  = np.max(np.abs(fp  - fc ))              # Error en la primer derivada
err_pp = np.max(np.abs(fpp - fcc))              # Error en la segunda derivada

# Gráficos
fig, axs = plt.subplots(1, 3, figsize=(12,4), constrained_layout=True)
fig.suptitle("Cálculo de derivadas usando DCT")

axs[0].plot(x, f, label="Analítica")
axs[0].set_title("$f(x) = J_4(4x)$")

axs[1].plot(x, fp, label="Analítica")
axs[1].plot(x, fc,  "-o", zorder=-1, label="Chebyshev con DCT")
axs[1].set_title("$f'(x) = 2[J_3(4x) - J_5(x)]$")
axs[1].text(0, -0.55, f"$Máx. error:{err_p:.2e}$", va="center", ha="center")

axs[2].plot(x, fpp, label="Analítica")
axs[2].plot(x, fcc, "-o", zorder=-1, label="Chebyshev con DCT")
axs[2].set_title("$f''(x) = 4[J_2(4x) + J_6(4x) - 2J_4(4x)]$")
axs[2].text(0, -0.25, f"$Máx. error:{err_pp:.2e}$", va="center", ha="center")

[ax.legend(frameon=False) for ax in axs]
[ax.set_xlabel("$x$") for ax in axs];
```



Noten que para poder utilizar una normalización similar a la que venimos usando hasta ahora (y que es usual en la bibliografía), debemos usar norm="forward"'. Esta normalización solo la incorporan versiones de Scipy mayores o iguales a 1.6 que Google Colab no incorpora por defecto. Adicionalmente, la transformada realizada por Scipy considera $c_m = 1 \; \forall m$, por lo que no es necesario tratar de forma separada los modos 0 y N en la relación de recurrencia.

1.3 Comentarios finales sobre métodos espectrales

Vale resaltar que en el caso de Fourier también puede realizarse la diferenciación numérica mediante una cierta matriz de diferenciación. Adicionalmente, tanto en el caso de Chebyshev como en el de Fourier las matrices de diferenciación pueden obtenerse como un aproximante de diferencias finitas de orden máximo para el caso equiespaciado periódico (Fourier) o una grilla de Gauss-Lobatto (Chebyshev).

Sin embargo, quedarse únicamente con esa manera de ver a los métodos espectrales resultaría bastante limitada. Por un lado, como vimos anteriormente, el uso de matrices de diferenciación requiere $\mathcal{O}(N^2)$ operaciones, mientras que los métodos que diferencian en el dominio espectral y emplean transformadas rápidas, requieren $\mathcal{O}(N \log N)$ operaciones. Además del cósto asintótico de cada método (que es una característica muy relevante), en problemas no lineales, vimos el problema del *aliasing no-lineal*, cuyo origen y solución resulta considerablemente más natural en el enfoque espectral. Adicionalmente, las propiedades de convergencia de estos métodos son sencillas de estudiar a partir de conocimientos sobre representaciones en series de funciones y en particular de series de Fourier.

Más aún, y como vieron en teóricas, los métodos espectrales permiten evolucionar directamente los coeficientes de una cierta representación para nuestra función de interés (método de Galerkin). Bajo este método, pueden evolucionarse de manera natural funciones que pertenezcan a un cierto subespacio, como por ejemplo, aquellas que posean una cierta condición de contorno, o algún otro tipo de condición.

Es importante remarcar que el enfoque presentado en esta práctica sobre métodos espectrales, i.e. un método rápido y preciso para calcular derivadas, apenas tiene un objetivo introductorio y pedagógico. Las técnicas espectrales suelen ser muy apropiadas en el estudio de numérico de sistemas físicos, y esperamos que estos ejercicios les ayuden a resolver problemas de su interés.

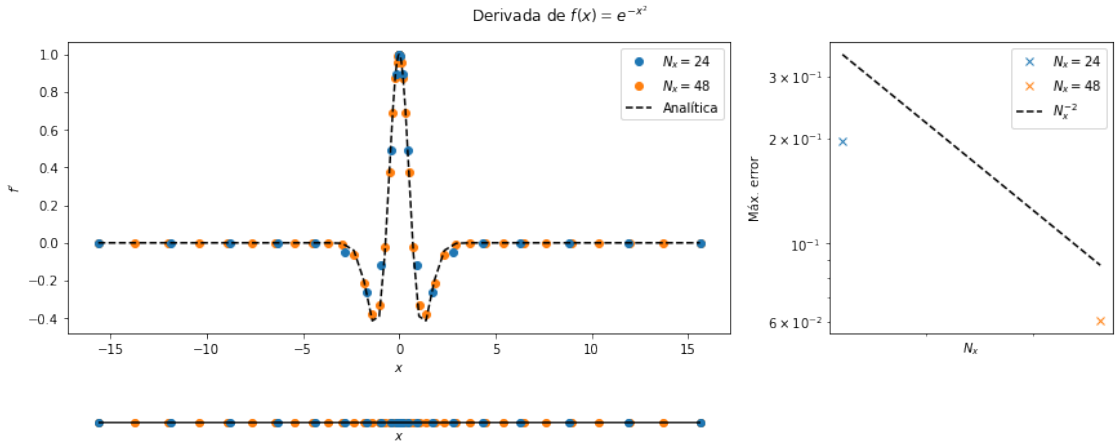
1.4 Findiff en grillas no equiespaciadas

También necesitarán en para un ejercicio de esta práctica, calcular derivadas mediante esquemas de diferencias finitas para grillas no uniformes. Esto pueden hacerlo mediante Findiff, y de manera muy similar a como vienen utilizando esta biblioteca.

Hasta el momento, la generación de matrices de diferenciación la realizaban mediante `FinDiff(0, dx, ord, acc=prec).matrix((Nx,1))`, donde `0` denota el eje en el arreglo de destino sobre el que debe operar la matriz, `dx` el espaciamiento entre puntos, `ord` el orden de la derivada buscada y `pres` el orden de precisión deseado. En caso que la grilla sea no uniforme, basta con reemplazar el escalar `dx` por un arreglo de puntos `x` con las coordenadas de cada punto de grilla. Veamos un ejemplo tomado de la documentación de Findiff:

[ ]:
```
# Instalar findiff si no está instalado
!if ! pip list | grep findiff >> /dev/null; then pip install findiff; fi
from findiff import FinDiff
import numpy as np
import matplotlib.pyplot as plt

NXS = np.array([24, 48])
```

```
fig, axs = plt.subplots(2, 2, figsize=(12,5), constrained_layout=True,
                        sharex="col", gridspec_kw={"width_ratios" : [.7,.3],
                                                   "height_ratios": [.8,.2]})

axs[1,1].remove()
for i, NX in enumerate(NXS):
    t = np.linspace(-2.5, 2.5, NX)
    x = t**3
    axs[1,0].plot(x, np.ones_like(x), 'o', zorder=-i)

    f   =  np.exp(-x**2) * x
    fp =  np.exp(-x**2) * (1 - 2*x**2)

    D   = FinDiff(0, x, 1, acc=2).matrix((NX,1))
    num = D @ f

    err = np.max(np.abs(fp - num))
    axs[0,0].plot  (x , num, 'o', label=f"$N_x={NX}$", zorder=-i)
    axs[0,1].loglog(NX, err, 'x', label=f"$N_x={NX}$")

axs[0,0].plot(x, fp, '--k', label="Analítica")
axs[0,0].set_xlabel("$x$")
axs[0,0].set_ylabel("$f'$")

axs[0,1].loglog(NXS, 2e2*NXS**-2.0, '--k', label="$N_x^{-2}$")
axs[0,1].set_xlabel("$N_x$")
axs[0,1].set_ylabel("Máx. error")

axs[1,0].axis("off")
axs[1,0].hlines(1, x[0], x[-1])
axs[1,0].text(0, 0.98, "$x$", va="center", ha="center")

[ax.legend()                                for ax in axs[0].flatten()]
[ax.xaxis.set_tick_params(labelbottom=True) for ax in axs[0].flatten()]
fig.suptitle("Derivada de $f(x) = e^{-x^2}$");
```



Como muestra el panel de la derecha, tenemos el orden de convergencia requerido, aún cuando la grilla es no uniforme. Para ayudar a visualizar el espaciamiento entre puntos, colocamos abajo un eje con los puntos de cada grilla. Vemos que claramente es no uniforme, con más puntos cerca del centro (donde el gradiente es mayor) y menos hacia los extremos.

Solo una nota de cautela: en lo que he podido notar Findiff no permite obtener el orden de precisión requerido en caso para derivadas pares en caso que las grillas no varíen suavemente. En este caso por ejemplo, donde $x = t^3 = \xi(t)$ con t equiespaciada y ξ una función suave, esto no es problema. Pero si nuestra grilla fuera aleatoria y quisieramos obtener la p-ésima derivada, donde p es par, con orden de precisión o, el esquema generado por FinDiff resultaría en una estimación con orden de precisión $o - 1$. Los motivos por los que esto sucede los vieron en la práctica sobre el Método de Volumenes Finitos (MVF), en un contexto ligeramente distinto.

1.5 Referencias

- L. N. Trefethen. 2000. *Spectral methods in MATLAB*. Society for Industrial and Applied Mathematics, USA.
- C. Canuto, M. Y. Hussaini, A. Quarteroni- T. A. Zang Jr. (1988). *Spectral Methods in Fluid Dynamics*. Part of Springer Series in Computational Physics. Springer.
- J. Shen, T. Tang, L. Wang. (2011). *Spectral methods*. Part of Springer Series in Computational Mathematics. Springer.
- Orszag, S. A. (1971). *On the Elimination of Aliasing in Finite-Difference Schemes by Filtering High-Wavenumber Components*, Journal of Atmospheric Sciences, 28(6), 1074-1074.

2 Funciones útiles

Adjuntamos a continuación algunas funciones que pueden resultarles de utilidad para la realización de esta Práctica o el análisis de los resultados obtenidos.

2.0.1 Matrices de diferenciación para condiciones de contorno periódicas (diferencias finitas)

[ ]:
```
#@title
def diferenciacion_centrada_periodica(N, d, orden=1, precision=2):
    """
    Devuelve una representación rala de la matriz de diferenciación que
    aproxima a la derivada de un cierto orden. Puede devolver esquemas con
    distintos órdenes de precisión.

    Entrada:
        - `N`:         Cantidad de puntos a diferenciar.
          (entero)
        - `d`:         Espaciamiento entre puntos.
          (flotante)
        - `orden`:     Orden de la derivada a aproximar.
          (entero)
        - `precision`: Orden de precisión del aproximante utilizado.
          (entero)

    Salida:
        - `D`: Representación rala de la matriz de diferenciación.
          (`scipy.sparse.dia.dia_matrix`)
    """
    from scipy.sparse import diags

    if precision > N-1:
        raise ValueError("Cantidad de puntos insuficiente para"
                         " la precisión requerida.")

    # Derivada primera
    if orden == 1:
        if precision == 2:
            coefs = [ [-1], [0], [1] ]
            fact  = 1/2
        elif precision == 4:
            coefs = [ [1], [-8], [0], [8], [-1] ]
            fact  = 1/12
        elif precision == 6:
            coefs = [ [-1], [9], [-45], [0], [45], [-9], [1] ]
            fact  = 1/60
        elif precision == 8:
            coefs = [ [3], [-32], [168], [-672], [0], [672], [-168], [32], [-3] ]
            fact  = 1/840
        else:
            raise ValueError("Orden de precisión inexistente o"
                             " no implementado.")
```

```
        fact *= 1/d

    # Derivada segunda
    elif orden == 2:
        if precision == 2:
            coefs  = [ [1], [-2], [1] ]
            fact   = 1
        elif precision == 4:
            coefs  = [ [-1], [16], [-30], [16], [-1] ]
            fact   = 1/12
        elif precision == 6:
            coefs  = [ [2], [-27], [270], [-490], [270], [-27], [2] ]
            fact   = 1/180
        elif precision == 8:
            coefs  = [ [-9], [128], [-1008], [8064], [-14350] ]
            coefs += [ [8064], [-1008], [128], [-9] ]
            fact   = 1/5040
        else:
            raise ValueError("Orden de precisión inexistente o"
                             " no implementado.")

        fact *= 1/d**2
    else:
        raise ValueError("Orden de derivación inexistente o no implementado.")

    # Periodicidad
    l        = len(coefs)
    coefs   += coefs[:l//2] + coefs[l//2+1:]
    offsets  = list(range(-l//2+1, l//2+1))
    offsets += [  N + offsets[i] for i in range(0     , l//2) ]
    offsets += [ -N + offsets[i] for i in range(l//2+1,    l) ]

    return fact*diags(coefs, offsets=offsets, shape=(N,N))
```

2.0.2 Matriz de diferenciación de Chebyshev de primer orden

[ ]:
```
#@title
def diferenciacion_chebyshev(N):
    """
        Crea una matriz de diferenciación de Chebyshev para N puntos que se
    ubican sobre una grilla x_i = cos[(N-i-1)/(N-1)] (Gauss-Lobato).

    Entrada:
        -`N`: Tamaño de la matriz de diferenciación.
    Salida:
```

```
        -`D`: Matriz (NxN) de diferenciación.
    """
    import numpy as np

    # Deshabilito los warnings de división por cero.
    with np.errstate(invalid="ignore", divide="ignore"):
        # Genero la grilla
        x = np.cos(np.linspace(0, np.pi, N))
        # Armo la matriz de signos
        D = (-1)**(np.arange(0, N)[:,None] + np.arange(0, N)[None,:])
        # Divido por las distancias entre puntos
        D = D/(x[:,None] - x[None,:])
        # Escaleo la primer y última fila (excepto las esquinas)
        D[[0,-1], 1:-1] = 2*D[[0,-1], 1:-1]
        # Escaleo la primer y última columna (excepto las esquinas)
        D[1:-1, [0,-1]] = .5*D[1:-1, [0,-1]]
        # Elimino NaN de la diagonal. Elijo cero como valor por el próximo paso.
        np.fill_diagonal(D, 0)
        # Cada fila debe sumar 0, con esto completo la diagonal.
        np.fill_diagonal(D, - np.sum(D, axis=1))

    # Devuelvo -D porque quiero usar una grilla con x creciente
    return -D
```

2.0.3 Gráfico 1D animado

```
[ ]: #@title
def grafico1d_animado(abscisas, ordenadas, dt, leyendas=None, titulo="",
                      etiqueta_x="", etiqueta_y="", paso=1, rescalar=False,
                      figsize=None):
    """
    Genera un gráfico animado 1D.

    Entrada:
        - `abscisas`:   arreglo 1D o lista de arreglos 1D con las abscisas para
                        cada conjunto de datos.
        - `ordenadas`:  arreglo de dimensión 2 con el valor de las ordenadas
                        para cada tiempo. La cantidad de filas corresponde
                        a la cantidad de niveles temporales y debe ser igual
                        para cada conjunto de datos. La cantidad de columnas
                        debe coincidir con la cantidad de elementos en las
                        abscisas.
        - `dt`:         paso temporal entre muestras.
        - `leyendas`:   string o lista de strings con la etiqueta para cada set
                        de datos.
```

```
    - `titulo`:     string con el título del gráfico.
    - `etiqueta_x`: string con la etiqueta para el eje x [OPCIONAL].
    - `etiqueta_y`: string con la etiqueta para el eje y [OPCIONAL].
    - `paso`:       espaciamiento en los datos para cada fotograma.
    - `rescalar`:   True para recalcular los limites de la figura en cada
                        fotograma.
    - `figsize`:    tamaño de la figura.

Salida:
    - `anim`:   referencia al objeto de animación creado.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

#   Control de errores y flexibilidad para permitir uno o varios
# sets de datos.
if not isinstance(abscisas, list):
    if not isinstance(abscisas, np.ndarray):
        print("El primer argumento de `grafico1d_animado` debe ser un "
              "arreglo con las abscisas o una lista de arreglos.")
        raise
    else:
        abscisas = [abscisas]

if (not isinstance(ordenadas, list)):
    if (not isinstance(ordenadas, np.ndarray)):
        print("El segundo argumento de `grafico1d_animado` debe ser un "
              "arreglo con los datos o una lista de arreglos.")
        raise
    else:
        ordenadas = [ordenadas]

if len(abscisas) != len(ordenadas):
    print("La cantidad de arreglos de abscisas y de ordenadas debe "
          "ser la misma.")
    raise

if leyendas is None:
    leyendas = [None]*len(ordenadas)
elif leyendas and isinstance(leyendas, list):
    if len(ordenadas) != len(leyendas):
        print("Ordenadas y leyendas deben tener la misma cantidad de ",
              "elementos.")
        raise
else:
```

```
    leyendas = [leyendas]

# Guardo el estado de plt
params_viejos = plt.rcParams
plt.rc('animation', html='jshtml')

num_foto = ordenadas[0].shape[0]

fig, ax = plt.subplots(1, 1, figsize=figsize, constrained_layout=True)
plt.close();  # Cerrar la figura, animation va a crear la suya propia

# Inicializo las curvas
plots = [ ax.plot([], [], label=leyendas[i])[0]
          for i in range(len(ordenadas)) ]
ax.set_title(titulo + f" $t=0$")
ax.set_xlabel(etiqueta_x)
ax.set_ylabel(etiqueta_y)

def init():
    """ Inicializador de la figura y gráfico de condiciones iniciales."""
    for i, (x, f) in enumerate(zip(abscisas, ordenadas)):
        plots[i].set_xdata(x)
        plots[i].set_ydata(f[0])

    ax.relim()
    ax.autoscale_view()

    return plots

def actualizar(t):
    """ Actualiza los datos al fotograma actual."""
    print(f"\rCalculando fotograma {t//paso} de {(num_foto-1)//paso}",
          end="")
    for i, f in enumerate(ordenadas):
        plots[i].set_ydata(f[t])

    ax.set_title(titulo + f" $t={t*dt:.5f}$")

    if rescalar:
        ax.relim()
        ax.autoscale_view()

    return plots

anim = animation.FuncAnimation(fig, actualizar, init_func=init,
                               frames=range(0, num_foto, paso),
                               blit=True, repeat=True)
```

```
    # Restauro el estado de plt
    plt.rc(params_viejos)

    return anim
```

2.0.4 Gráfico 2D animado

[ ]:
```
#@title
def grafico2d_animado(x, y, escalar, dt, titulo="", etiqueta_x="",
                      etiqueta_y="",etiqueta_escalar="", figsize=None,
                      paso=1):
    """
    Genera un gráfico animado 2D.

    Entrada:
        - `x`:       arreglo 1D (NX) con las abscisas de los datos datos.
        - `y`:       arreglo 1D (NY) con las ordenadas de los datos datos.
        - `escalar`: arreglo 2D (NX,NY) con los valores del campo escalar sobre
                     la grilla cartesiana.
        - `dt`:      paso temporal entre muestras.

        - `titulo`:           string con el título del gráfico [OPCIONAL].
        - `etiqueta_x`:       string con la etiqueta para el eje x [OPCIONAL].
        - `etiqueta_y`:       string con la etiqueta para el eje y [OPCIONAL].
        - `etiqueta_escalar`: string con la etiqueta del campo escalar
                              [OPCIONAL].
        - `figsize`:          2-upla con el tamaño de la figura.
        - `paso`:             espaciamiento en los datos para cada fotograma.
                              [OPCIONAL]

    Salida:
        - `anim`:   referencia al objeto de animación creado.
    """

    import numpy as np
    import matplotlib.pyplot as plt
    import matplotlib.animation as animation

    # Guardo el estado de plt
    params_viejos = plt.rcParams
    plt.rc('animation', html='jshtml')

    num_foto =escalar.shape[0]
```

```
    fig, ax = plt.subplots(1, 1, figsize=figsize, constrained_layout=True)
    plt.close();  # Cerrar la figura, animation va a crear la suya propia

    # Inicializo las curvas
    plot = ax.imshow( np.ones((x.size, y.size)), extent=(x[0],x[-1],y[0],y[-1]),
                      origin="lower", interpolation='gaussian',
                      vmin=escalar.min(), vmax=escalar.max())

    cbar = fig.colorbar(plot, ax=ax, orientation="horizontal")
    cbar.set_label(etiqueta_escalar)

    ax.set_title(titulo + f" $t=0$")
    ax.set_xlabel(etiqueta_x)
    ax.set_ylabel(etiqueta_y)

    def init():
        """ Inicializador de la figura y gráfico de condiciones iniciales."""
        plot.set_data(escalar[0].T)
        return plot,

    def actualizar(t):
        """ Actualiza los datos al fotograma actual."""
        print(f"\rCalculando fotograma {t//paso} de {(num_foto-1)//paso}",
              end="")
        plot.set_data(escalar[t].T)

        ax.set_title(titulo + f" $t={t*dt:.5f}$")
        return plot,

    anim = animation.FuncAnimation(fig, actualizar, init_func=init,
                                   frames=range(0, num_foto, paso),
                                   blit=True, repeat=True)

    # Restauro el estado de plt
    plt.rc(params_viejos)

    return anim
```