[MN] Apunte 4 - 2021

October 4, 2021

Métodos numéricos

Segundo cuatrimestre 2021

Práctica 4: Ecuaciones parabólicas

Cátedra: Pablo Dmitruk

Fecha límite de entrega: 22 de octubre de 2021 23:59

1 Ecuaciones parabólicas

En esta práctica vamos a estar trabajando con ecuaciones parabólicas. Las mismas se caracterizan por fenómenos que se propagan de manera inmediata a todo el espacio. Un ejemplo de este tipo de ecuaciones es la ecuación de difusión del calor en ausencia de fuentes, dada por

$$\frac{\partial T}{\partial t} - \nu \nabla^2 T = 0, \tag{1}$$

con T el campo de temperatura y ν la difusividad térmica (se la asumió constante). Si bien suele usarse κ para denotar a ésta última cantidad, utilizaremos ν en adelante por consistencia con el resto de este documento.

Si bien ecuaciones de este estilo violan la visión relativista de la física y la naturaleza (donde la propagación local de información no puede exceder a la velocidad de la luz en el vacío), pueden resultar extremadamente utiles para modelar fenomenología muy diversa.

Sin embargo, aunque su simplicidad conceptual resulta atractiva, numéricamente puede conllevar algunas dificultades con respecto a los sistemas hiperbólicos de la práctica previa, dado que el dominio de dependencia para un cierto punto en el futuro se vuelve infinito (asociado a una velocidad de propagación de la información infinita). Sin embargo, aún cuando la solución rigurosamente depende del estado previo en cada punto del dominio físico, la mayor contribución a la solución está dada por los puntos más cercanos a donde queramos calcular nuestra solución. Por tanto, dada una cierta precisión buscada, tendremos un dominio de dependencia finito. Este argumento debería corresponderse con su vivencia diaria. Si toman una placa de metal lo suficientemente grande y colocan un encendedor en una punta, no sienten de forma inmediata un cambio apreciable en la temperatura en la punta opuesta al encendedor, aún cuando el fenómeno se describe de forma extremadamente exitosa mediante la ecuación (1).

No obstante, incluso definiendo un dominio de dependencia "práctico", las ecuaciones parabólicas pueden ser desafiantes numéricamente, o rígidas utilizando una expresión más apropiada en el contexto de métodos numéricos. Esto van a descubrirlo por ustedes mismos en los ejercicios 1 y

2, donde verán que mejorar la resolución espacial es mucho más penalizante sobre la estabilidad un integrador explíta comparado con los problemas hiperbólicos de la práctica 1. Asimismo, los problemas parabólicos pueden generar en algunos escenarios capas límites† muy finas, que a su vez volverán imprescindible utilizar resoluciones espaciales altas y podrían resultar en que la integración explícita de un sistema parabólico sea inviable.

†: regiones donde hay un cambio cualtiativo en la dinámica del sistema que por tanto deben resolverse correctamente. Un ejemplo de esta fenomenología es la capa límite de Prandtl para un flujo viscoso paralelo a una placa sólida estacionaria.

1.1 Crank-Nicolson

La estrategia utilizada en la práctica previa para resolver sistemas hiperbólicos seguirá siendo aplicable para resolver sistemas parabólicos. Esto es, podremos utilizar el método de líneas, obteniendo versiones discretas de los operadores diferenciales espaciales e integrando el sistema de EDOs resultante mediante algún integrador temporal. Estos operadores espaciales podrán ser aproximaciones en diferencias, i.e. diferencias finitas.

Sin embargo, como se desprende de la discusión previa, esta estrategia aparejada con un integrador temporal explícito no gozará, en general, de buenas características de estabilidad. Esto es particularmente cierto para $\nu \gg 1$ y para sistemas que desarrollen escalas muy pequeñas que nuestra grilla espacial deberá capturar.

En particular, para la ecuación de calor a difusividad constante y para numerosos problemas de interés en la descripción de la naturaleza, la parte parabólica de la ecuación (i.e. el término difusivo) suele ser lineal, volviendo relativamente sencillo el uso de técnicas implícitas de integración temporal. Esta idea tuvieron Crank y Nicolson, quienes propusieron utilizar un método de Adams-Moulton de 2do orden (i.e. la regla trapezoidal) para integrar el término difusivo de la ecuación de calor. De esta manera, sea $\mathcal{L}$ el operador discreto que aproxima a $\nu\nabla^2$, el método de Crank-Nicolson aplicado a la ecuación de difusión de temperatura resulta

$$T^{n+1} = T^n + \frac{\Delta t}{2}\left(\mathcal{L}\{T^n\} + \mathcal{L}\{T^{n+1}\}\right).$$

En la notación usual del curso, y considerando el caso 1D por simplicidad, $\mathcal{L}(T^n)$ va a poder representarse como $\nu D_{xx}\mathbf{T}^n$ y de esta manera basta con obtener $A = (\mathbb{I} - \frac{\nu\Delta t}{2}D_{xx})^{-1}$ antes de comenzar la integración temporal para poder aplicar eficientemente el siguiente esquema para integrar en tiempo

$$\mathbf{T}^{n+1} = A\left(\mathbf{T}^n + \frac{\nu\Delta t}{2}D_{xx}\mathbf{T}^n\right).$$

Naturalmente, el orden de aproximación espacial quedará dado por aquel asociado a D_{xx}, mientras que la intgración temporal resultará de 2do orden, como verificarán en el problema 3.

1.2 Problemas con dos dimensiones espaciales

Como mencionamos hacia el final del apunte de la práctica previa (que les recomendamos fuertemente repasar), para resolver una EDP en un dominio que presenta dos dimensiones espaciales, una opción es generalizar las ideas que venimos tratando en 1 dimensión. Para ello, consideramos una grilla bidimensional sobre la que quedan definidas coordenadas discretas $(x_i, y_j) =$

$(i\Delta x + x_0, j\Delta y + y_0)$. Naturalmente, Δx y Δy representan el espaciamiento uniforme en la dirección x e y, respectivamente.

De esta manera, si $f(t, x, y)$ es la función incógnita que buscamos obtener mediante una EDP, $f_{ij}^n = f(t^n, x_i, y_j)$ será nuestra aproximación discreta. Dadas condiciones iniciales apropiadas, hallar $f_{ij}^n = f(t^n, x_i, y_j)$ puede verse como evolucionar una matriz en el tiempo, dada por

$$\mathbb{F}^n = \begin{pmatrix} f_{1,1}^n & f_{1,2}^n & f_{1,3}^n & \cdots & f_{1,N_y-1}^n \\ f_{2,1}^n & f_{2,2}^n & f_{2,3}^n & \cdots & f_{2,N_y-1}^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ f_{N_x-1,1}^n & f_{N_x-1,2}^n & f_{N_x-1,3}^n & \cdots & f_{N_x-1,N_y-1}^n \end{pmatrix},$$

donde consideramos que tenemos $N_x + 2$ puntos en la discretización de la direccción x y $N_y + 2$ en la dirección y. Además, dado que contamos con condiciones de contorno apropiadas, no será necesario avanzar $f_{0,j}^n$, $f_{i,0}^n$, $f_{N_x,j}^n$, f_{i,N_y}^n explícitamente en el tiempo.

Para el cálculo de derivadas en la dirección x, podemos reutilizar la estrategia que venimos empleando para los problemas 1D. De esta forma, asumiendo que tenemos condiciones de contorno de Dirichlet en toda la frontera del dominio (i.e. conocemos explícitamente $f_{0,j}^n$, $f_{i,0}^n$, $f_{N_x,j}^n$, f_{i,N_y}^n), podemos escribir para cada línea con y constante (i.e. cada columna de $\mathbb{F}$)

$$\begin{pmatrix} (\partial_x f)_{1,j}^n \\ (\partial_x f)_{2,j}^n \\ \vdots \\ (\partial_x f)_{N_x-1,j} \end{pmatrix} = D_x \begin{pmatrix} f_{1,j}^n \\ f_{2,j}^n \\ \vdots \\ f_{N_x-1,j} \end{pmatrix} + \mathbf{b}_{(x)j},$$

donde $\mathbf{b}_{(x)j}$ es un vector columna que contiene a $f_{0,j}^n$ y $f_{N_x,j}^n$, escalados adecuadamene de manera que la estimación de $(\partial_x f)_{1,j}^n$ y $(\partial_x f)_{N_x,j}^n$. El resultado de realizar este procedimiento para cada linea con coordenada y fija, puede sintetizarse en la operación matricial

$$(\partial x \mathbb{F})^n = D_x \mathbb{F}^n + \mathbb{B}_{(x)},$$

con $\mathbb{B}_{(x)}$ la matriz que surge de apilar horizontalmente los distintos vectores columna $\mathbf{b}_{(x)j}$.

De manera análoga, sea D_y un operador análogo a D_x pero construido con los parámetros de la grilla en y, valdrá la siguiente ecuación para cada línea con x fijo (i.e. cada fila de $\mathbb{F}$)

$$\begin{pmatrix} (\partial_y f)_{i,1}^n & (\partial_y f)_{i,2}^n & \cdots & (\partial_y f)_{i,N_y-1}^n \end{pmatrix} = \begin{pmatrix} f_{i,1}^n & f_{i,2}^n & \cdots & f_{i,N_y-1}^n \end{pmatrix} D_y^t + \mathbf{b}_{(y)i},$$

donde $\mathbf{b}_{(y)i}$ es un vector fila con función completamente análoga a aquella de $\mathbf{b}_{(x)j}$. Nuevamente, esta operación aplicada simultáneamente a todas las líneas de x constante podrá escribirse como

$$(\partial y \mathbb{F})^n = \mathbb{F}^n D_y^t + \mathbb{B}_{(y)}.$$

Noten que D_y podría definirse directamente como su transpuesta, sin embargo resulta más claro conceptualmente definirlo de manera idéntica a como venimos definiendo D_x y transponer luego debido a que debe operar sobre las filas en lugar de las columnas de $\mathbb{F}$.

Definidos los correspondientes operadores espaciales, una vez más el problema se redujo a un conjunto finito de ecuaciones diferenciales ordinarias acopladas, que podemos evolucionar en el tiempo con cualquiera de los algoritmos de integración temporal que venimos usando en la materia.

1.2.1 Esténcil de 5 puntos para el laplaciano

Para fijar ideas, consideremos el esténcil de 5 puntos para el operador laplaciano, que surge sencillamente de aproximar

$$\nabla^2 f \approx D^{(2)}_{xx}\mathbb{F} + \mathbb{F}D^{(2)t}_{yy} + \mathbb{B},$$

donde $D^{(2)}$ denota una diferencia finita centrada de segundo orden en cada dirección y $\mathbb{B} = \mathbb{B}_{(x)} + \mathbb{B}_{(y)}$, en la notación anterior. La forma explícita de estas matrices resulta

$$A = \begin{pmatrix} -2 & 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 & -2 \end{pmatrix}, \qquad D^{(2)}_{xx} = \frac{1}{(\Delta x)^2}A, \qquad D^{(2)}_{yy} = \frac{1}{(\Delta y)^2}A,$$

$$\mathbb{B} = \mathbb{B}_{(x)} + \mathbb{B}_{(y)} = \frac{1}{(\Delta x)^2}\begin{pmatrix} f_{0,1} & f_{0,2} & \dots & f_{0,N_y-2} & f_{0,N_y-1} \\ 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \\ f_{N_x,1} & f_{N_x,2} & \dots & f_{N_x,N_y-2} & f_{N_x,N_y-1} \end{pmatrix} + \frac{1}{(\Delta y)^2}\begin{pmatrix} f_{1,0} & 0 & \dots & 0 & f_{1,N_y} \\ f_{2,0} & 0 & \dots & 0 & f_{2,N_y} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ f_{N_x-2,0} & 0 & \dots & 0 & f_{N_x-2,N_y} \\ f_{N_x-1,0} & 0 & \dots & 0 & f_{N_x-1,N_y} \end{pmatrix}.$$

Resulta inmediato ver por qué este esquema recibe el nombre de esténcil de 5 puntos, ya que se tendrá

$$(\nabla^2 f)^n_{i,j} = \frac{f_{i-1,j} - 2f_{i,j} + f_{i+1,j}}{(\Delta x)^2} + \frac{f_{i,j-1} - 2f_{i,j} + f_{i,j+1}}{(\Delta y)^2},$$

y por lo tanto se utiliza información de los puntos $f_{i,j}$, $f_{i-1,j}$, $f_{i+1,j}$, $f_{i,j-1}$ y $f_{i,j+1}$. De igual manera, se sigue que este esquema nunca utiliza la información de las esquinas del dominio: $f_{0,0}$, f_{0,N_y}, $f_{N_x,0}$ y f_{N_x,N_y}. Este problema puede solucionarse empleando otra aproximación para el laplaciano (o cualquier otro operador) que utilice, por ejemplo, 9 puntos $(f_{i-1,j-1}, f_{i-1,j}, f_{i-1,j+1}, f_{i,j-1}, f_{i,j}, f_{i,j+1}, f_{i+1,j-1}, f_{i+1,j}, f_{i+1,j+1})$.

En general **el esténcil de 5 puntos** producirá buenos resultados a pesar de no utilizar la información de las esquinas y por tanto **es el único que usaremos en esta práctica**. Más aún, permite resolver fácilmente el caso donde sobre las mismas las condiciones de contorno no resultan contínuas. Un ejemplo de este problema puede ser el una placa que ocupa el dominio $(x, y) \in [0, L_x] \times [0, L_y]$ y donde la parte de la placa en (L_x, y) está en contacto con un baño térmico a temperatura T_1 y la parte en (x, L_y) con otro a temperatura $T_2 \neq T_1$. Otra situación común es la de una placa rectangular cuyos extremos se hallan a diferente potencial electrostático. Si bien sabemos que la realidad macroscópica suele ser contínua y suave, las situaciones planteadas pueden ser un modelo exitoso para el caso donde en las puntas de la placa se encuentran recubiertas con un material aislante pero no queremos resolver explícitamente dicha dinámica.

1.3 Condiciones de contorno dependientes del tiempo

En el problema 7 van a encontrar que tienen condiciones de contorno de tipo Dirichlet, pero dependientes del tiempo, es decir dado un cierto problema diferencial asociado a f, van a tener la condición

$$f(t, 0) = b_1(t), \qquad f(t, L_x) = b_2(t).$$

Si bien el problema en cuestión les pide utilizar integradores de Runge-Kutta de primer y segundo orden, a fines de obtener soluciones más precisas podrían intentar implementar un método de Runge-Kutta de 4to orden (y la correspondientes matrices de diferenciación expacial de órdenes superiores). Sin embargo, se encontrarían con el problema de cómo tratar las condiciones de contorno para las etapas intermedias. La idea más intuitiva sería evaluar b_1 y b_2 para los mismos tiempos donde se está calculando la etapa intermedia. Sin embargo, en general, esta estrategia permite alcanzar como máximo 2do orden global. No veremos en la materia cómo solucionar este problema. En caso de que precisemos órdenes de aproximación mayores a problemas con contornos utilizaremos integradores de la familia Adams. Sin embargo, quienes estén intersados, pueden leer sobre este inconveniente con los métodos de Runge-Kutta y algunas posibles soluciones en este paper clásico.

Noten que el problema anterior solo aparece cuando las condiciones de contorno dependen del tiempo. En caso que las mismas sean estacionarias los métodos de Runge-Kutta mantendrán el orden de aproximación esperable a priori.

1.4 Funciones útiles

1.4.1 Matrices de diferenciación para condiciones de contorno periódicas

[ ]:
```
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))
```

1.4.2 Gráfico 1D animado

[ ]:
```
def grafico1d_animado(abscisas, ordenadas, dt, leyendas=None, titulo="",
                      etiqueta_x="", etiqueta_y="", paso=1, rescalar=False):
    """
    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.

    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 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=(8,4), 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
```

1.4.3 Gráfico 2D animado

[ ]:
```
def grafico2d_animado(x, y, escalar, dt, titulo="", etiqueta_x="",
                      etiqueta_y="",etiqueta_escalar="", 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].
        - `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=(8,8), constrained_layout=True)
    plt.close();  # Cerrar la figura, animation va a crear la suya propia

    # Inicializo las curvas
    plot = ax.imshow( np.ones_like(X), 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=False, repeat=True)

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

    return anim
```