Ecuaciones diferenciales parciales

¡Bienvenidos al cuarto laboratorio! En el laboratorio anterior vimos ecuaciones diferenciales ordinarias (EDO), donde la función desconocida depende de una sola variable. En las ecuaciones diferenciales parciales (EDP) esta función puede depender de cualquier número de variables. En este laboratorio vamos a aprender a resolver ecuaciones diferenciales parciales numéricamente.

Ya que vivimos en un mundo tridimensional, y además el tiempo pasa inexorablemente, encontramos EDPs en la mayoría de los modelos que usamos para simular fenómenos físicos: la ecuación de onda, de calor, electrodinámica, Navier-Stokes, etc.

La clase pasada revisamos la discretización de EDPs usando la ecuación de convección y difusión en 1D como ejemplo, lo que repasaremos a continuación. En este laboratorio ustedes implementarán varias cosas: las ecuaciones de convección no lineal, difusión y Burgers en una dimensión. Como extra, implementarán las ecuaciones de convección no lineal y difusión en 2D, lo que pueden hacer fuera del horario de clases si es que no alcanzan.

Teoría

Discretización

Para repasar los principales conceptos, usaremos la ecuación de convección lineal en una dimensión:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0.$$

En este caso, existen dos derivadas: una temporal (t) y otra espacial (x), y necesitamos generar una malla que abarque ambas componentes:

En el laboratorio 2 vimos que podemos tratar las derivadas usando diferencia adelantada, centrada, o atrasada; por lo tanto, tenemos 6 combinaciones posibles de discretización. Por ejemplo, la discretización atrasada en espacio y adelantada en tiempo da la siguiente ecuación:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + c\frac{u_{i+1}^n - u_i^n}{\Delta x},$$

donde el subíndice i recorre la componente espacial de la malla (con espaciamiento Δx) y el superíndice n la componente temporal (con pasos de tiempo Δt). Podemos reescribir esta ecuación poniendo las incógnitas a la izquierda y lo conocido a la derecha:

$$u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}\left(u_i^n - u_{i-1}^n\right)$$

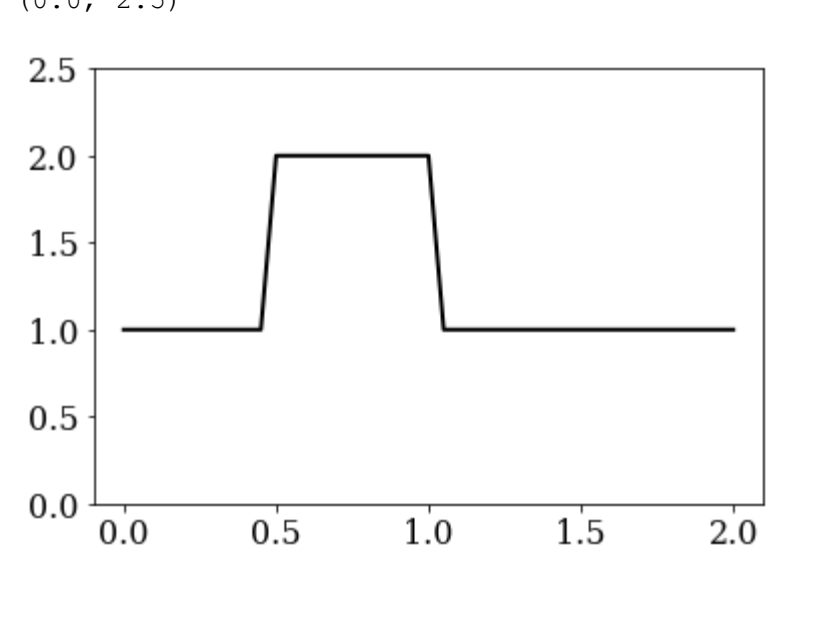
Gráficamente, podemos representar la ecuación de la siguiente forma:

Ejemplo: una onda cuadrada

Vamos a calcular numéricamente la ecuación de convección lineal en un dominio entre 0 y 2, con $c = 1$. La condición inicial es una función que vale 1 en todas partes, menos entre 0.5 y 1, donde es dos. Dibujemos la condición inicial.

In [1]:
```
import numpy
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import rcParams
rcParams['font.family'] = 'serif'
rcParams['font.size'] = 16

nx = 41
dx = 2./(nx-1)
x  = numpy.linspace(0, 2, nx)

u_inicial = numpy.ones(nx)
u_inicial[int(0.5/dx):int(1/dx+1)] = 2.0
plt.plot(x, u_inicial, c='k', lw=2)
plt.ylim([0,2.5])
```

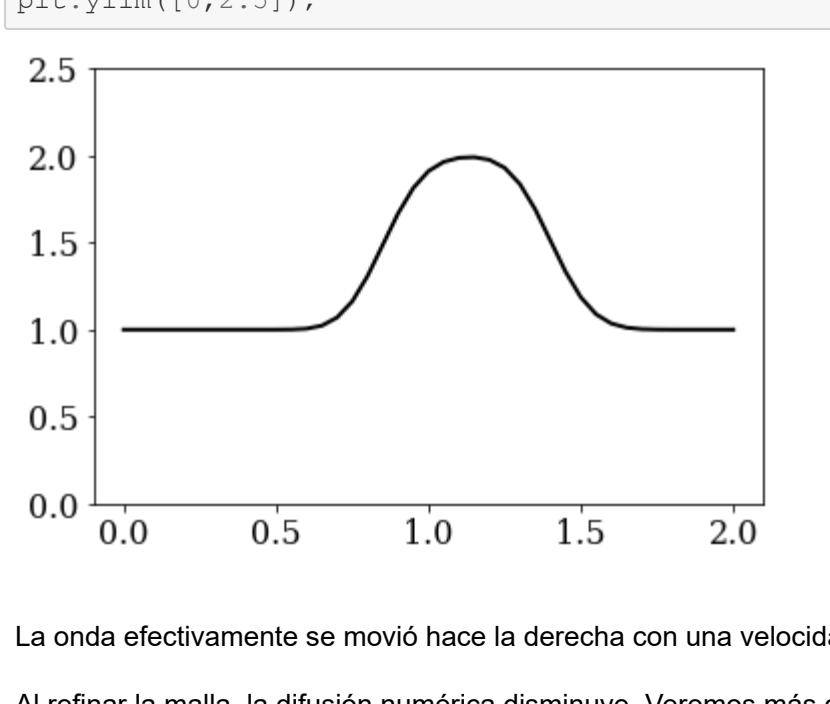
Out[1]: (0.0, 2.5)



La ecuación de convección lineal es una forma de la ecuación de onda. Para una condición inicial $u_0(x)$ la solución exacta es $u(x-ct)$, por lo tanto, esperaríamos que esta función cuadrada se mueva hacia la derecha con velocidad c manteniendo su forma: es una onda cuadrada.

Evaluemos la función después de 20 pasos de tiempo a ver si es así. Para probar, usemos $\Delta t = 0.02$

In [2]:
```
def ftbs(u, nt, nx, c, dt, dx):
    for n in range(1,nt):
        un = u.copy()
        for i in range(1,nx):
            u[i] = un[i]-c*dt/dx*(un[i]-un[i-1])
    return u

nt = 20
dt = 0.02
c  = 1
u = u_inicial.copy()
u  = ftbs(u, nt, nx, c, dt, dx)
plt.plot(x, u, c='k', lw=2)
plt.ylim([0,2.5]);
```



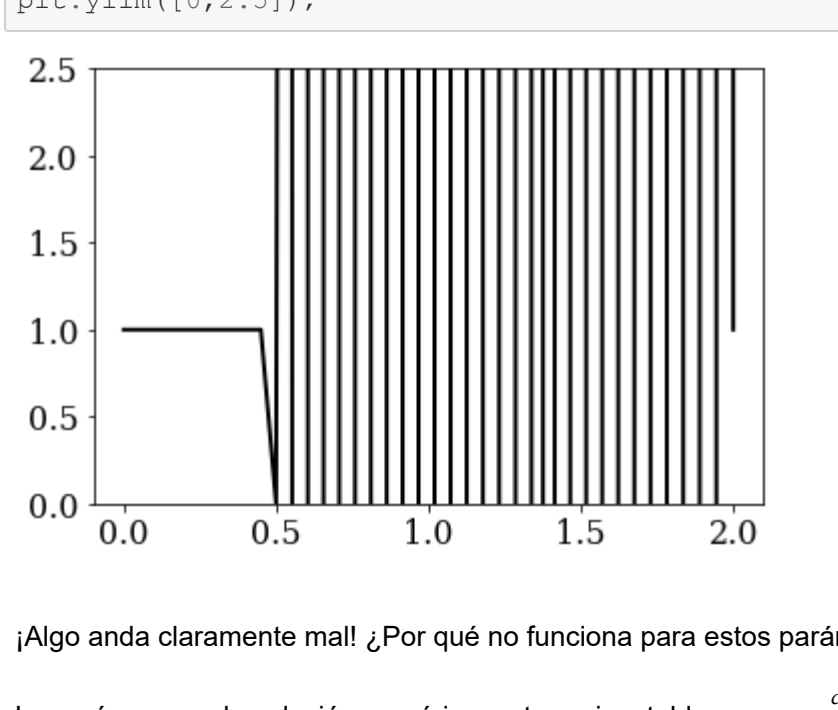
La onda efectivamente se movió hace la derecha con una velocidad $c = 1$, pero perdió su forma cuadrada por efectos de difusión numérica.

Al refinar la malla, la difusión numérica disminuye. Veremos más detalle de esto en clases.

Condición Courant-Friedrichs-Lewy

Sería ideal que la solución avanzara más rápido. Del laboratorio 2 sabemos que la ecuación discretizada con diferencias adelantadas en tiempo cae como $O(\Delta t)$, pero si no estamos tan preocupados de la exactitud, podemos aumentar Δt para obtener la solución antes ¿Existe algún límite para esto? Probemos $\Delta t = 0.1$

In [3]:
```
dt = 0.1
u = u_inicial.copy()
u  = ftbs(u, nt, nx, c, dt, dx)
plt.plot(x, u, c='k', lw=2)
plt.ylim([0,2.5]);
```



¡Algo anda claramente mal! ¿Por qué no funciona para estos parámetros?

La razón es que la solución numérica se torna inestable para $\sigma = \frac{c\Delta t}{\Delta x} > 1$, donde σ es el número de Courant. Esto se conoce como la condición CFL.

En este caso $\sigma = \frac{1\cdot 0.1}{\frac{2}{40}} = 2$, por lo tanto la solución numérica es inestable. Prueben con otros valores de Δt para convencerse, y veremos la teoría detrás en clases.

Ahora les toca a ustedes

Convección no lineal

La ecuación de convección no lineal es

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0.$$

Como ven, es muy parecida al caso lineal, sin embargo, en vez de tener una velocidad constante c, ésta es u. Para el caso lineal, la onda cuadrada se movía hacia la derecha con velocidad c ¿Qué creen ustedes que pasará ahora?

Al discretizar con diferencia adelantada en el tiempo y atrasada en el espacio, nos queda:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i\frac{u_{i+1}^n - u_i^n}{\Delta x},$$

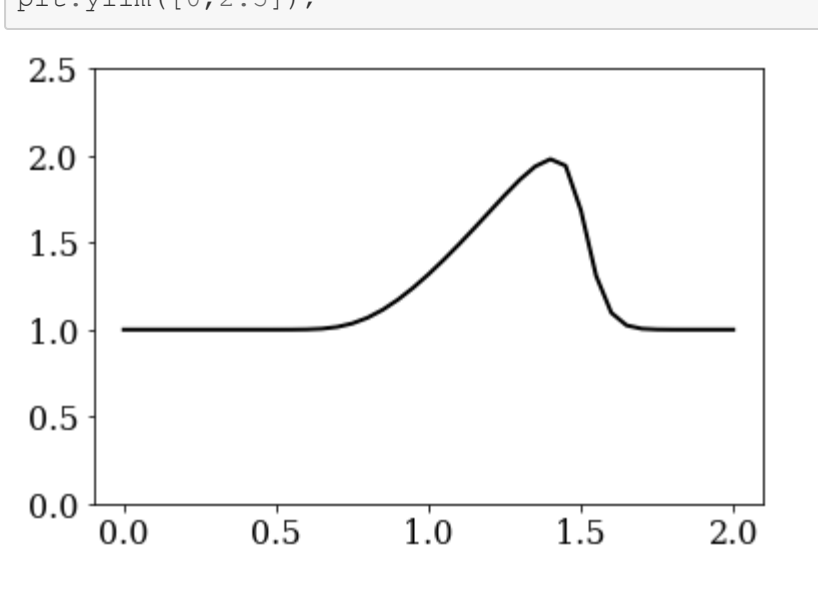
Implementen una función que evalúe la ecuación de convección no lineal para la onda cuadrada tras 20 pasos de tiempo con $\Delta t = 0.02$, con la misma malla que el caso lineal.

Comenten sobre la forma de la onda en el último paso de tiempo ¿Es diferente al caso lineal? ¿Por qué?

Se despeja u_i^{n+1}

$$u_i^{n+1} = u_i^n - \frac{u_i^n\Delta t}{\Delta x}\left(u_i^n - u_{i-1}^n\right)$$

In [4]:
```
### ALUMNO
def ftbs_nl(u, nt, nx, dt, dx):
    for n in range(1,nt):
        un = u.copy()
        for i in range(1,nx):
            u[i] = un[i]-un[i]*dt/dx*(un[i]-un[i-1])
    return u

nx = 41
dx = 2./(nx-1)
nt = 20
dt = 0.02
c  = 1
u = u_inicial.copy()
u  = ftbs_nl(u, nt, nx, dt, dx)
plt.plot(x, u, c='k', lw=2)
plt.ylim([0,2.5]);
```



Claramente es diferente al caso lineal ya que se pierde la simetría que existía. La onda cuadrada parece desaparecer por completo por una onda distinta. Esto tiene relación con que la velocidad de cada nodo en un paso de tiempo depende de la velocidad de ese nodo en un paso anterior y además, de los nodos vecinos. En otras palabras, la velocidad ya no es constante. Hay que tener en cuenta también que existe difusión numérica.

Difusión de la onda cuadrada

La ecuación de convección cae en la clasificación de ecuación hiperbólica. Otra familia de ecuaciones son las parabolicas, y el caso más conocido es la ecuación de difusión. En una dimensión ésta es:

$$\frac{\partial u}{\partial t} - \nu\frac{\partial^2 u}{\partial x^2} = 0.$$

Una aplicación común de la ecuación de difusión es la ecuación de conducción de calor, donde u sería la temperatura.

Vimos en el laboratorio 2 que la mejor manera de discretizar le segunda derivada era usando diferencia centrada. Así, la versión discretizada de la ecuación de difusión es:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} - \nu\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2} = 0.$$

¿Qué pasa si la onda cuadrada se ve sujeta a difusión? Hagan una función que calcule esto numéricamente por 20 pasos de tiempo con $\Delta t = 0.001$, y $\nu = 0.5$. Fíjense que en este caso necesitamos los dos puntos vecinos (el de adelante y atrás) ¿puede el `for` llegar hasta el final del arreglo?

La ecuación de difusión tiene su propia condición CFL:

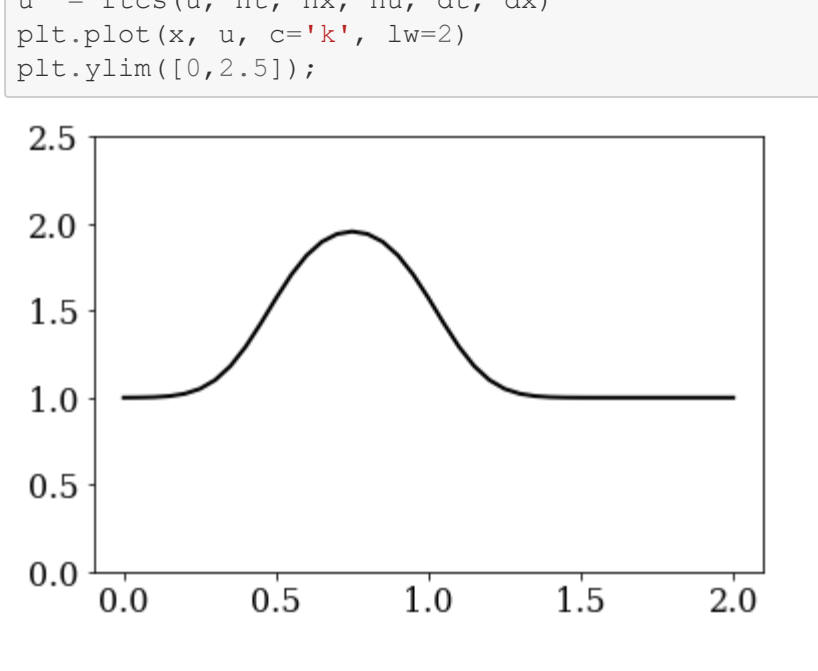
$$\frac{\nu\Delta t}{(\Delta x)^2} < \frac{1}{2}$$

Cambien $\Delta t = 0.005$ ¿Qué pasa? ¿Por qué?

Se despejará u_i^{n+1}

$$u_i^{n+1} = u_i^n + \Delta t\nu\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2}$$

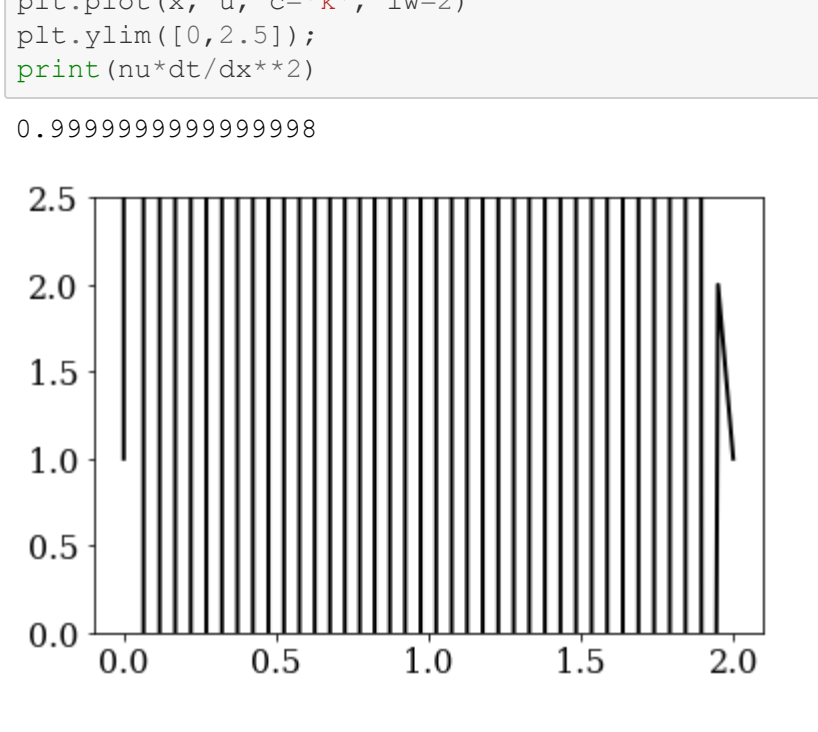
In [5]:
```
### ALUMNO
def ftcs(u, nt, nx, nu, dt, dx):
    for n in range(1,nt):
        un = u.copy()
        for i in range(1,nx-1):
            u[i] = un[i]+nu*dt/(dx**2)*(un[i+1]-2*un[i]+un[i-1])
    return u

nx = 41
dx = 2./(nx-1)
nt = 20
dt = 0.001
nu  = 0.5
u = u_inicial.copy()
u  = ftcs(u, nt, nx, nu, dt, dx)
plt.plot(x, u, c='k', lw=2)
plt.ylim([0,2.5]);
```



Se nota que la onda cuadrada sufrió el fenómeno de difusión. La solución tiene sentido ya que la onda permanece en el mismo lugar, pero cada vez más se suaviza.

In [6]:
```
nx = 41
dx = 2./(nx-1)
nt = 20
dt = 0.005
nu  = 0.5
u = u_inicial.copy()
u  = ftcs(u, nt, nx, nu, dt, dx)
plt.plot(x, u, c='k', lw=2)
plt.ylim([0,2.5]);
print(nu*dt/dx**2)
```

0.9999999999999998



Claramente el método se vuelve inestable ya que se aumento el Δt, y por ende, puede no cuplirse la condición CFL del problema. Como se calculó: $0.999 > 0.5$ por ende el método es inestable

Combinando convección y difusión: ecuación de Burgers en 1-D

La ecuación de Burgers

La ecuación de Burgers viscosa es una suma entre un término convectivo no-lineal y un término difusivo. Se escribe:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0,$$

donde la no-linealidad está dada por que al lado de la derivada temporal hay una u en vez de una constante c.

Esta ecuación es muy parecida a Navier-Stokes (sin el término de presión), y se usa mucho como una simplificación de ésta, por ejemplo, en estudios numéricos de turbulencia.

La ecuación de Burgers discretizada con diferencia adelantada en el tiempo, atrasada en el espacio para la diferencial de primer orden, y centrada para la segunda derivada queda:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i^n\frac{u_i^n - u_{i-1}^n}{\Delta x} - \nu\frac{u_{i+1}^n - 2u_i^n + u_{i+1}^n}{(\Delta x)^2} = 0.$$

Condiciones iniciales y de borde

Para este ejercicio, calcularemos la ecuación de Burgers en el dominio $[0, 2\pi]$ con condición inicial:

$$u = -\frac{2\nu}{\phi}\frac{\partial \phi}{\partial x} + 4$$
$$\phi(t=0) = \phi_0 = \exp\left(\frac{-x^2}{4\nu}\right) + \exp\left(\frac{-(x-2\pi)^2}{4\nu}\right)$$

y condiciones de borde $u(0) = u(2\pi)$.

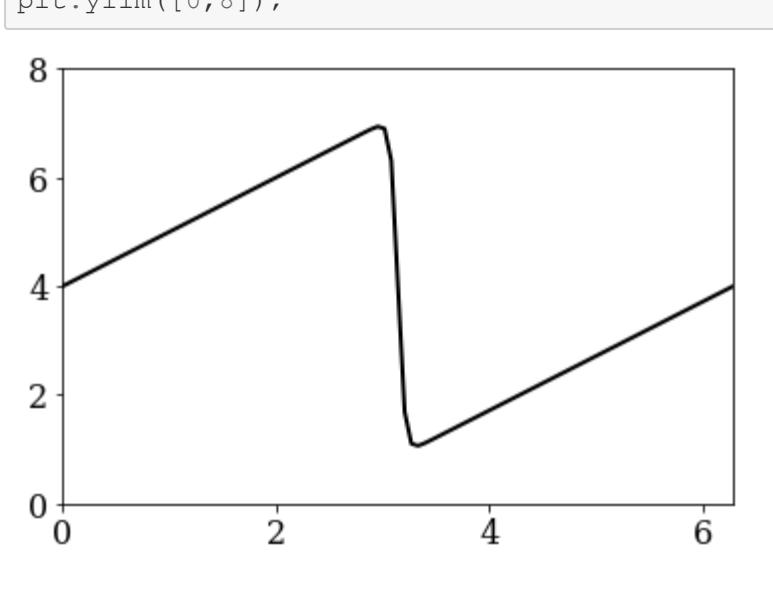
¿Ven algo raro en esas condiciones de borde? Más que especificar un valor para u o su derivada, estamos diciendo que cualquiera sea el valor en el borde derecho, éste se traspasa al borde izquierdo. Esto se conoce como una condición de borde periódica, y es equivalente a decir que lo que pasa que el dominio se repite cada 2π. En otras palabras, el punto en 0 y en 2π son el mismo punto. Cuando implementen las condiciones de borde, piensen en quienes son los nodos vecinos del nodo en 0 y 2π.

La siguiente función es una implementación de la condición inicial:

In [7]:
```
from math import pi

def cond_inicial(x, nu):
    phi    = numpy.exp(-x*x/(4*nu)) + numpy.exp(-(x-2*pi)**2/(4*nu))
    dphidx = -x/(2*nu) * numpy.exp(-x*x/(4*nu)) + \
             -(x-2*pi)/(2*nu) * numpy.exp(-(x-2*pi)**2/(4*nu))

    return -2*nu/phi * dphidx + 4
```

Grafiquémosla para tener una mejor idea. Usemos una malla con 101 nodos y $\nu = 0.1$:

In [8]:
```
nx = 101
x  = numpy.linspace(0, 2*pi, nx)
nu = 0.1

u0 = cond_inicial(x, nu)

plt.plot(x, u0, c='k', lw=2)
plt.xlim([0,2*pi])
plt.ylim([0,8]);
```



Solución analítica

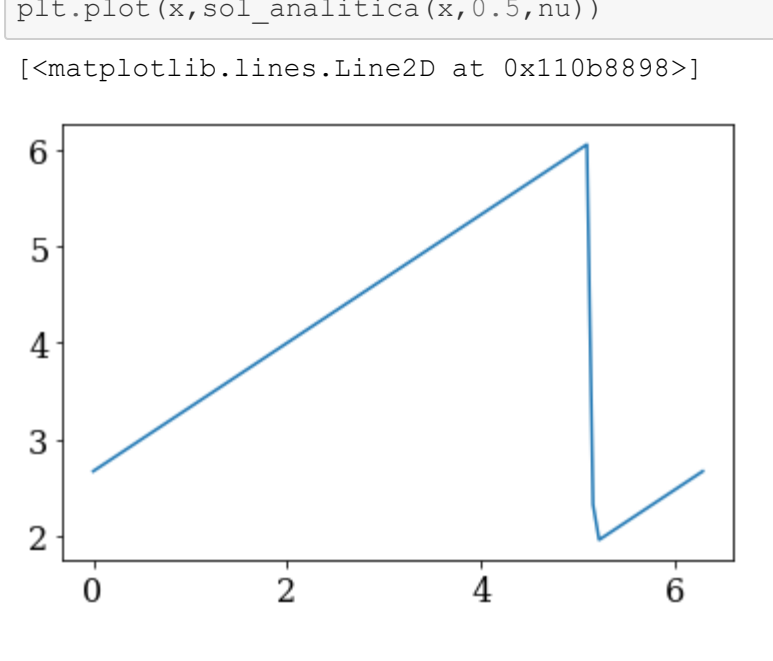
La ecuación de Burgers con las condiciones iniciales y de borde descritas arriba tiene la siguiente solución analítica:

$$u = -\frac{2\nu}{\phi}\frac{\partial \phi}{\partial x} + 4$$
$$\phi = \exp\left(\frac{-(x-4t)^2}{4\nu(t+1)}\right) + \exp\left(\frac{-(x-4t-2\pi)^2}{4\nu(t+1)}\right)$$

Para ayudarlos un poco, les entregaremos una función que entrega la solución analítica dado x, t, y ν:

In [9]:
```
def sol_analitica(x, t, nu):
    phi    = numpy.exp(-(x-4*t)**2/(4*nu*(t+1))) + numpy.exp(-(x-4*t-2*pi)**2/(4*nu*(t+1)))
    dphidx = -(x-4*t)/(2*nu*(t+1)) * numpy.exp(-(x-4*t)**2/(4*nu*(t+1))) + \
             -(x-4*t-2*pi)/(2*nu*(t+1)) * numpy.exp(-(x-4*t-2*pi)**2/(4*nu*(t+1)))

    return -2*nu/phi * dphidx + 4

nu = 0.01
nx = 101
x  = numpy.linspace(0, 2*pi, nx)
plt.plot(x,sol_analitica(x,0.5,nu))
```

Out[9]: [<matplotlib.lines.Line2D at 0x110b8898>]



Ejercicio

Compare con la solución analítica para mallas con 21, 41, 81 y 161 nodos y $\Delta t = 0.001$ en el tiempo $t = 0.5$ con $\nu = 0.5$. Usen el error L2 (Laboratorio 2), y hagan un gráfico *log-log* comparando el error y Δx para cada caso. ¿Cuál es el orden de aproximación con respecto a Δx? Considerando que tenemos aproximaciones de primer y segundo orden (derivada y segunda derivada) ¿Es lo que esperaba?

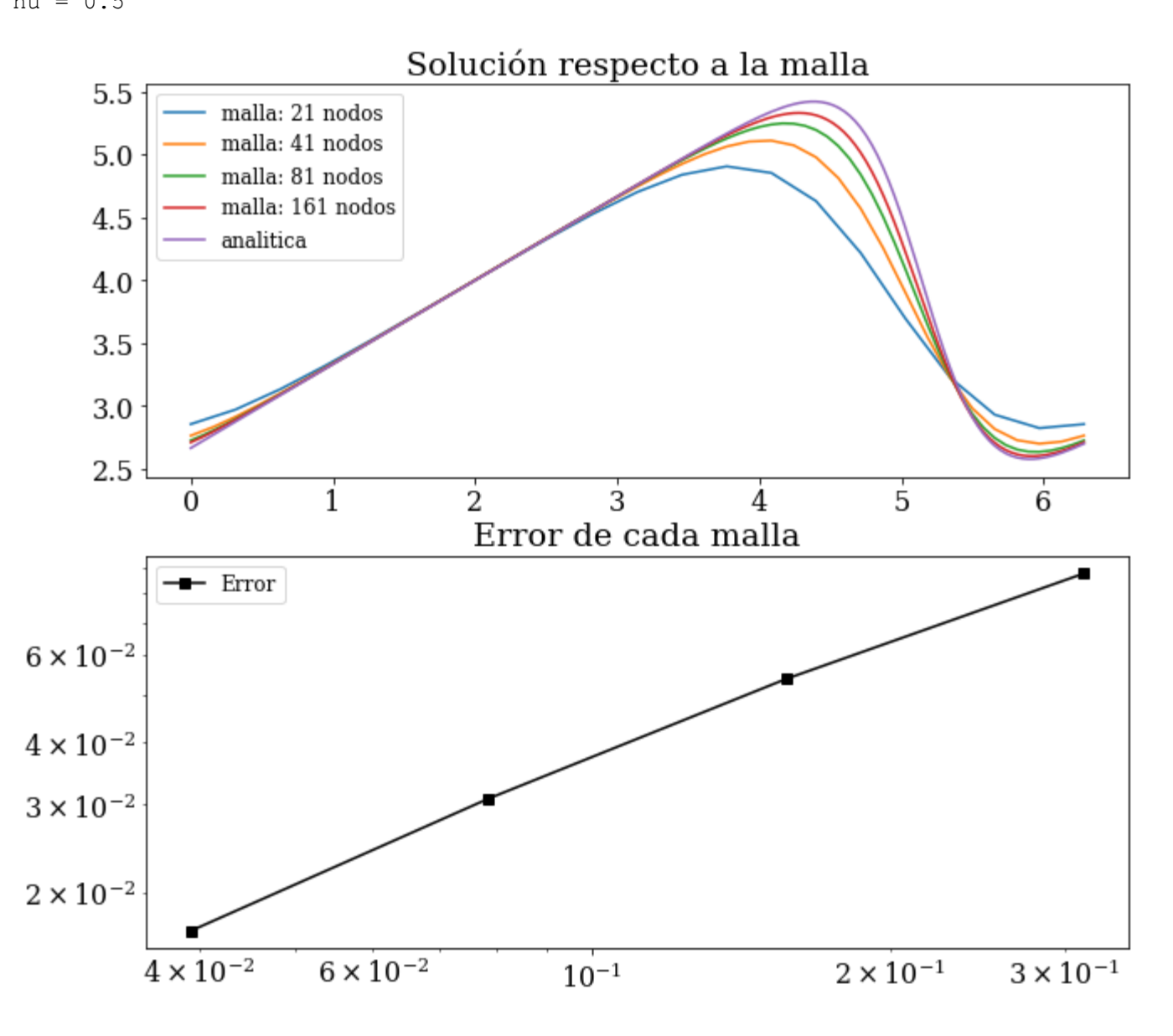
Cambien ν a $\nu = 0.01$ y verán que la solución tiene una pendiente mayor. En un gráfico, sobreponga la solución analítica y la numérica para cada caso. ¿Es la solución numérica capaz de reproducir la forma "zig-zag" de la solución? ¿Por qué? ¿Cuál es la influencia de la malla en esto? Si grafican el error versus Δx parece que el método ya no converge ¿Qué deben hacer para que si converja?

Cambien el valor de ν a $\nu = 0.001$. ¿Qué ocurre? ¿Por qué es inestable si no hemos cambiado Δt ni Δx?

In [10]:
```
import numpy as np


def burgers(u, nu, dt, dx, nx, nt):
    for n in range(1,nt):
        un = u.copy()
        for i in range(0,nx-1):
            if i == 0:
                u[i] = un[i]+dt*(nu/(dx**2)*(un[i+1]-2*un[i]+un[-2]) - un[i]*(un[i]-un[-2])/dx)
                u[-1] = u[0]
            else:
                u[i] = un[i]+dt*(nu/(dx**2)*(un[i+1]-2*un[i]+un[i-1]) - un[i]*(un[i]-un[i-1])/dx)
    return u

def L2_error(f_ap,f_an):
    df_an_app = (f_ap-f_an)**2
    f_an2 = f_an**2
    return np.sqrt(np.sum(df_an_app)/np.sum(f_an2))


def calculo_graficar(nu,L_nodos,t,dt):
    fig, axs = plt.subplots(2)
    fig.set_size_inches(10.5, 9.5)

    L_dx = 2*pi/(L_nodos-1)
    nt = int(t/dt)
    error = numpy.zeros(len(L_nodos))

    for k in range(len(L_nodos)):
        nx = L_nodos[k]
        dx = L_dx[k]
        x  = np.linspace(0, 2*pi, nx)
        u0 = cond_inicial(x, nu)
        u_burg = burgers(u0,nu,dt,dx,nx,nt)
        axs[0].plot(x,u_burg,label=f'malla: {nx} nodos')

        u_an = sol_analitica(x,t,nu)
        error[k] = L2_error(u_burg,u_an)

    axs[0].plot(x,sol_analitica(x,t,nu),label='analitica')
    axs[0].set_title("Solución respecto a la malla")
    axs[0].legend(loc='best',prop={'size':12})


    axs[1].loglog(L_dx,error,label="Error",c='k', ls='-', marker='s',)
    axs[1].set_title("Error de cada malla")
    axs[1].legend(loc='best',prop={'size':12})
```

In [11]:
```
dt = 0.001
t = 0.5
nu = 0.5
L_nodos = np.array([21,41,81,161])
print(f'nu = {nu}')
calculo_graficar(nu,L_nodos,t,dt)
```

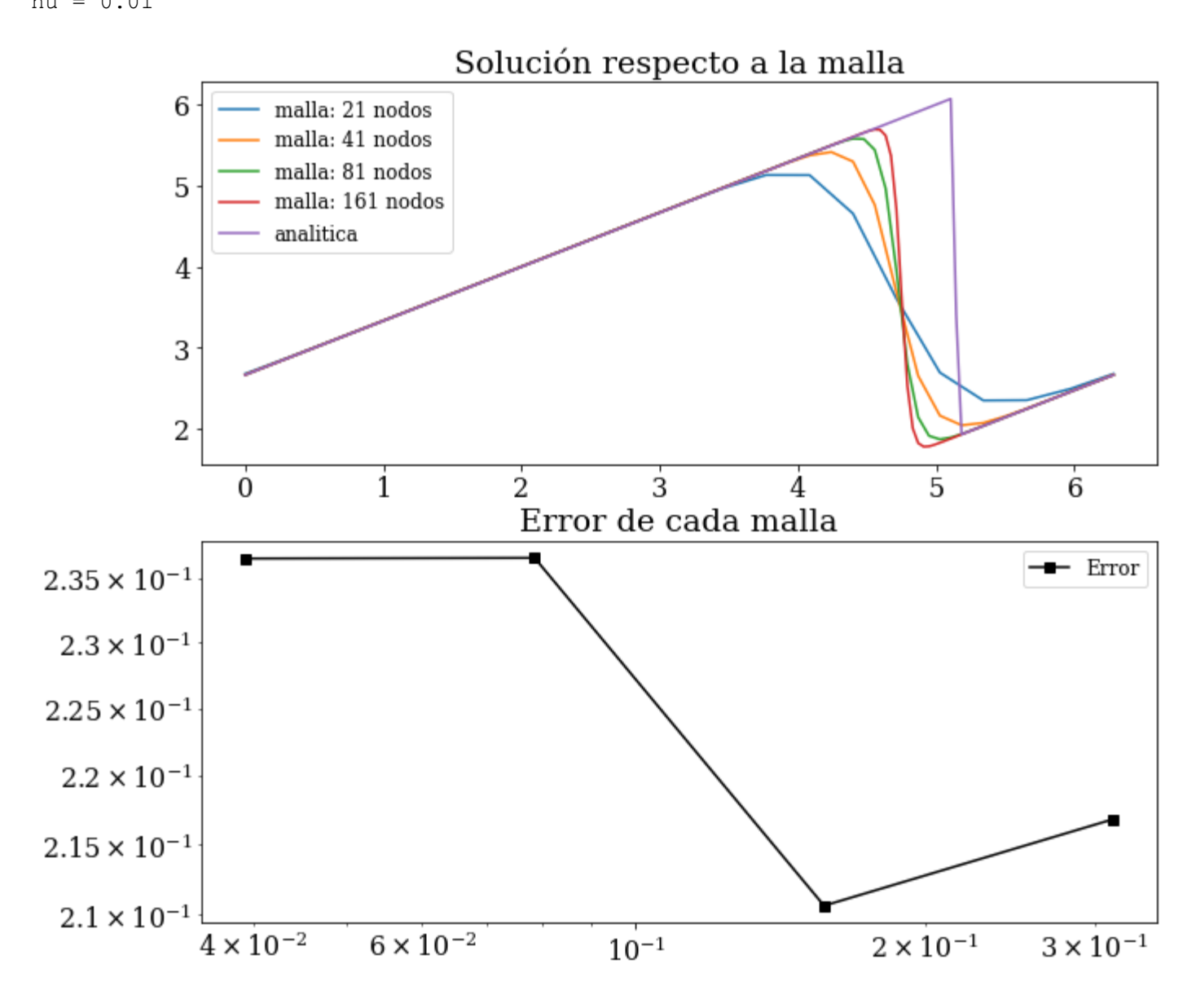
nu = 0.5



Se nota que el orden de aproximación es lineal (orden 1). Esto se debe a que se tienen aproximaciones de primer y segundo orden, donde predomina el orden más bajo. Refinando la malla se aproxima a la solución analítica.

In [12]:
```
nu = 0.01
dt = 0.001
print(f'nu = {nu}')
calculo_graficar(nu,L_nodos,t,dt)
```

nu = 0.01



Se nota que la solución numérica no puede modelar correctamente el "zigzag", predice la pendiente antes. Esto se debe a que la pendiente es muy alta, por lo que al tener un método de orden 1, se aprecian errores por difusión numérica.

El gráfico de error muestra que refinando la malla el error crece. Esto significa que el método se vuelve inestable. Para el ν en particular, lo ideal sería cambiar la malla temporal y espacial para que no se inestabilice el método, o simplemente cambiar el método. Lo ideal sería conocer la condición CFL para usar valores dentro del rango.

In [27]:
```
nu = 0.001
print(f'nu = {nu}')
#calculo_graficar(nu,L_nodos,t,dt)
```

nu = 0.001

Al parecer achicando mucho el ν la pendiente se vuelve tan elevada que python lo interpreta como división por cero. De todas formas, el método hubiera sido inestable por lo anterior.

Extra: EDP en dos dimensiones

Como esfuerzo extra, vamos a revisar la solución de convección y difusión en dos dimensiones.

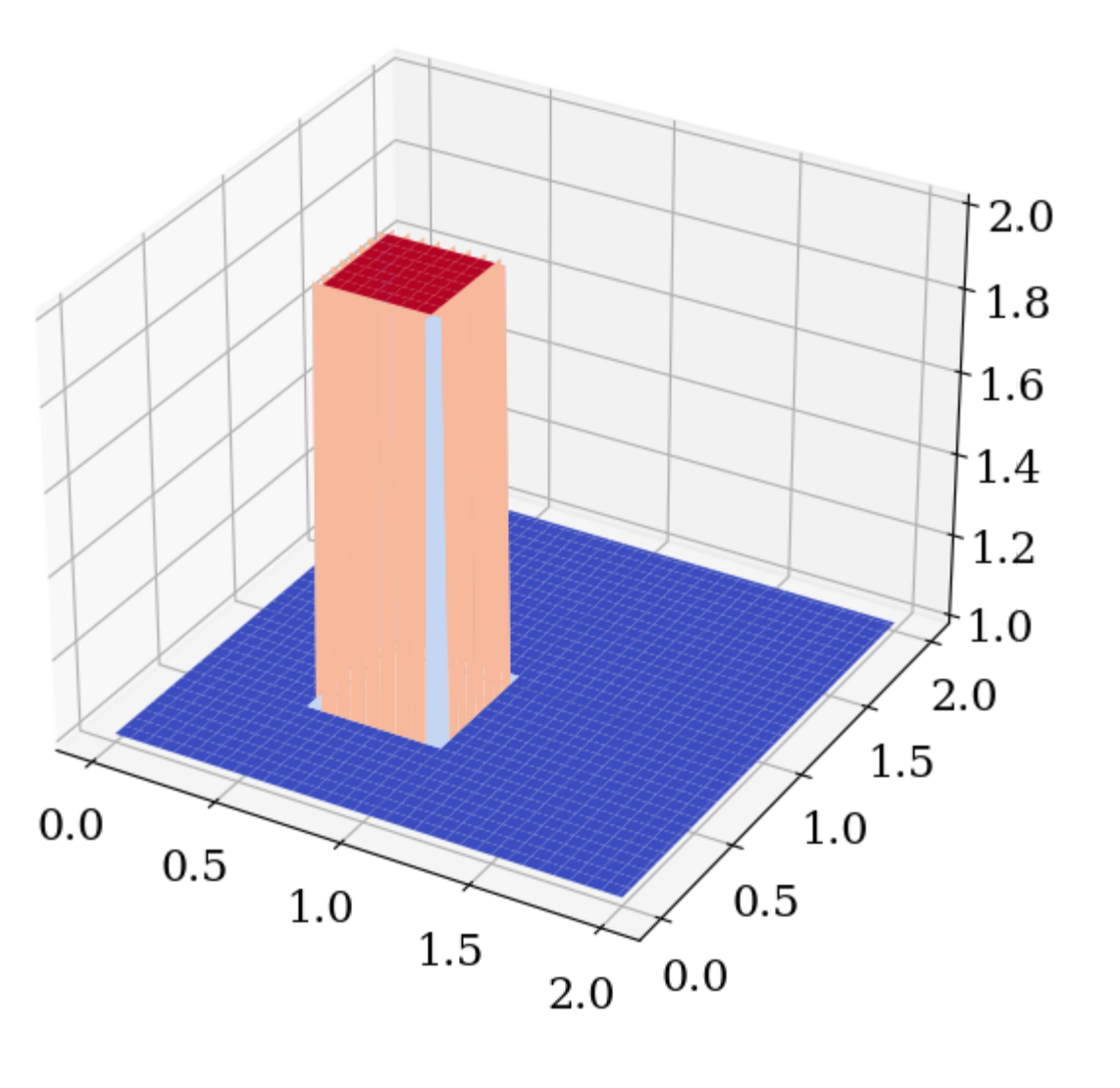
Convección lineal

Veamos primero el caso de convección lineal. Vamos a usar la condición incial equivalente: una función cuadrada en 2 dimensiones, en un dominio $[0,2]\times[0,2]$, con condición inicial es $u = 1$ en todas partes menos en $[0.5,1]\times[0.5,1]$, donde $u = 2$. Usaremos $\Delta x = \Delta y = 0.02$ y $\Delta t = 0.005$, y graficaremos u en $t = 0.5$.

In [14]:
```
nx = 101
ny = 101
dx = 2./(nx-1)
dy = 2./(ny-1)
dt = 0.005
nt = int(0.5/dt)

x = numpy.linspace(0,2,nx)
y = numpy.linspace(0,2,ny)

u_inicial = numpy.ones((ny,nx))
u_inicial[int(.5/dy):int(1/dy+1),int(.5/dx):int(1/dx+1)]=2
```

Para visualizar los resultados, usaremos una librería que grafica en 3D. Por ejemplo, la condición inicial se ve así:

In [15]:
```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(11,7), dpi=100)
ax = fig.gca(projection='3d')
X,Y = numpy.meshgrid(x,y)

ax.plot_surface(X,Y,u_inicial, cmap='coolwarm');
```



En 2 dimensiones, la ecuación de convección lineal es

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} + c\frac{\partial u}{\partial y} = 0$$

la cual, al discretizar con diferencia adelantada en tiempo y atrasada en espacio, queda

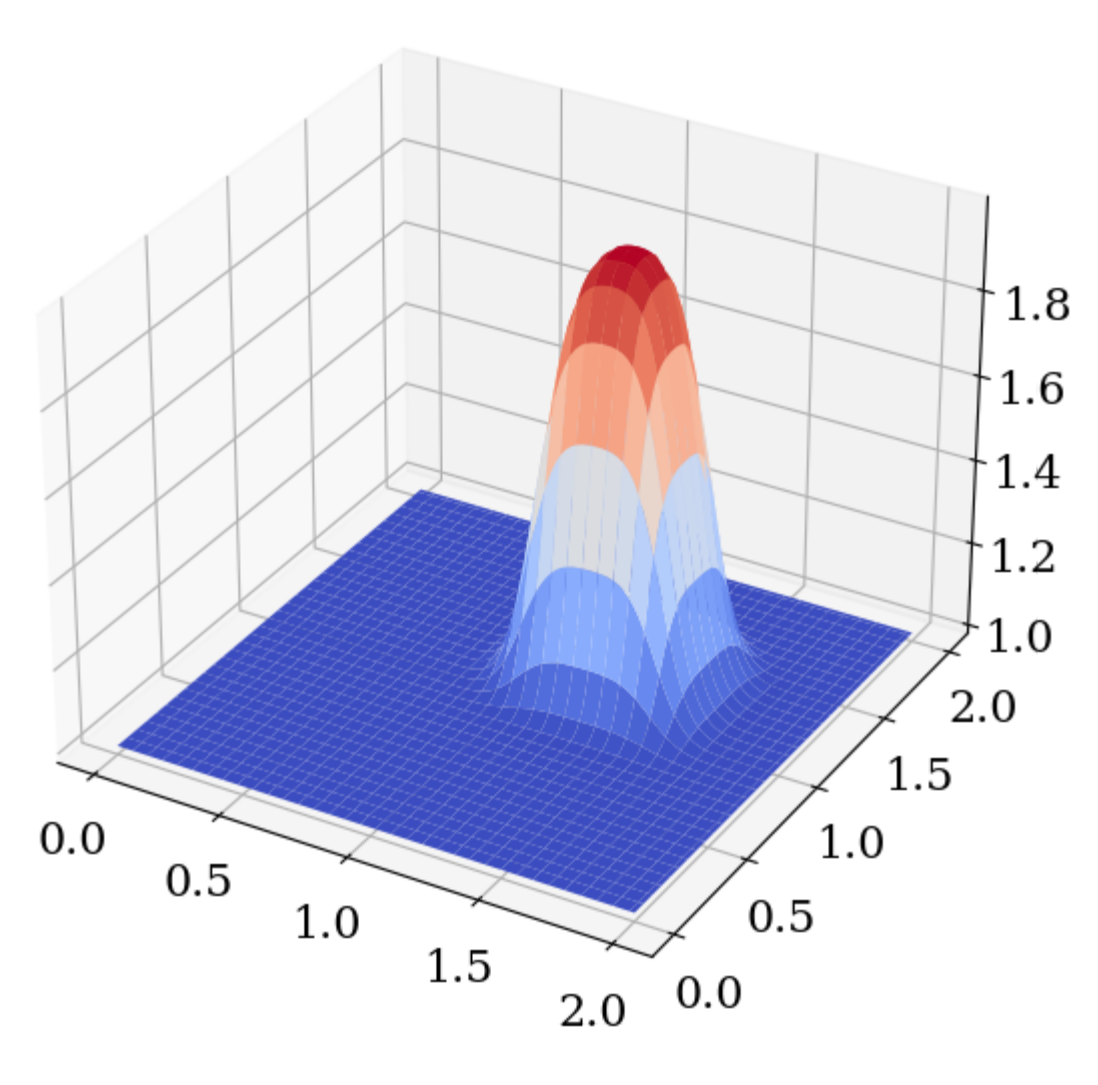
$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} + c\frac{u_{i,j}^n - u_{i-1,j}^n}{\Delta x} + c\frac{u_{i,j}^n - u_{i,j-1}^n}{\Delta y} = 0$$

Donde i recorre el eje x y j el eje y.

Esta es una onda cuadrada similar al ejemplo que dimos al principio, pero en 2 dimensiones ¿En qué dirección creen ustedes que se moverá? ¿Por qué?

Comenten sobre la forma final de la onda ¿Por qué no mantiene su forma cuadrada?

In [23]:
```
def ftbs2d(u, dx, dy, c, dt, nt):


    for n in range(nt+1): ##loop across number of time steps
        un = u.copy()
        row, col = u.shape
        for i in range(1, row):
            for j in range(1, col):
                u[i,j] = un[i, j] - (c*dt/dx*(un[i,j] - un[i,j-1]))-(c*dt/dy*(un[i,j]-un[i-1,j]))
                u[0,:] = 1
                u[-1,:] = 1
                u[:,0] = 1
                u[:,-1] = 1

    return u

c = 1
u = u_inicial.copy()
u = ftbs2d(u, dx, dy, c, dt, nt)

fig = plt.figure(figsize=(11,7), dpi=100)
ax = fig.gca(projection='3d')
ax.plot_surface(X,Y,u, cmap='coolwarm');
```



La onda avanzó en diagonal, en dirección x e y. Esto coincide con la suposición, ya que la velocidad está multiplicada por la derivada respecto a ambas coordenadas.

Claramente la onda no mantiene su forma cuadrada. Esto se debe a la difusión numérica dada por el orden del método utilizado. Esto mismo ocurría en 1D.

Convección no lineal

La ecuación de convección no lineal es un poco más complicada en 2D. En realidad, es un set de dos ecuaciones diferenciales parciales acopladas

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = 0$$
$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = 0$$

Discretizada con diferencia adelantada en tiempo y atrasada en espacio es:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} + u_{i,j}^n\frac{u_{i,j}^n - u_{i-1,j}^n}{\Delta x} + v_{i,j}^n\frac{u_{i,j}^n - u_{i,j-1}^n}{\Delta y} = 0$$
$$\frac{v_{i,j}^{n+1} - v_{i,j}^n}{\Delta t} + u_{i,j}^n\frac{v_{i,j}^n - v_{i-1,j}^n}{\Delta x} + v_{i,j}^n\frac{v_{i,j}^n - v_{i,j-1}^n}{\Delta y} = 0,$$

y despejando,

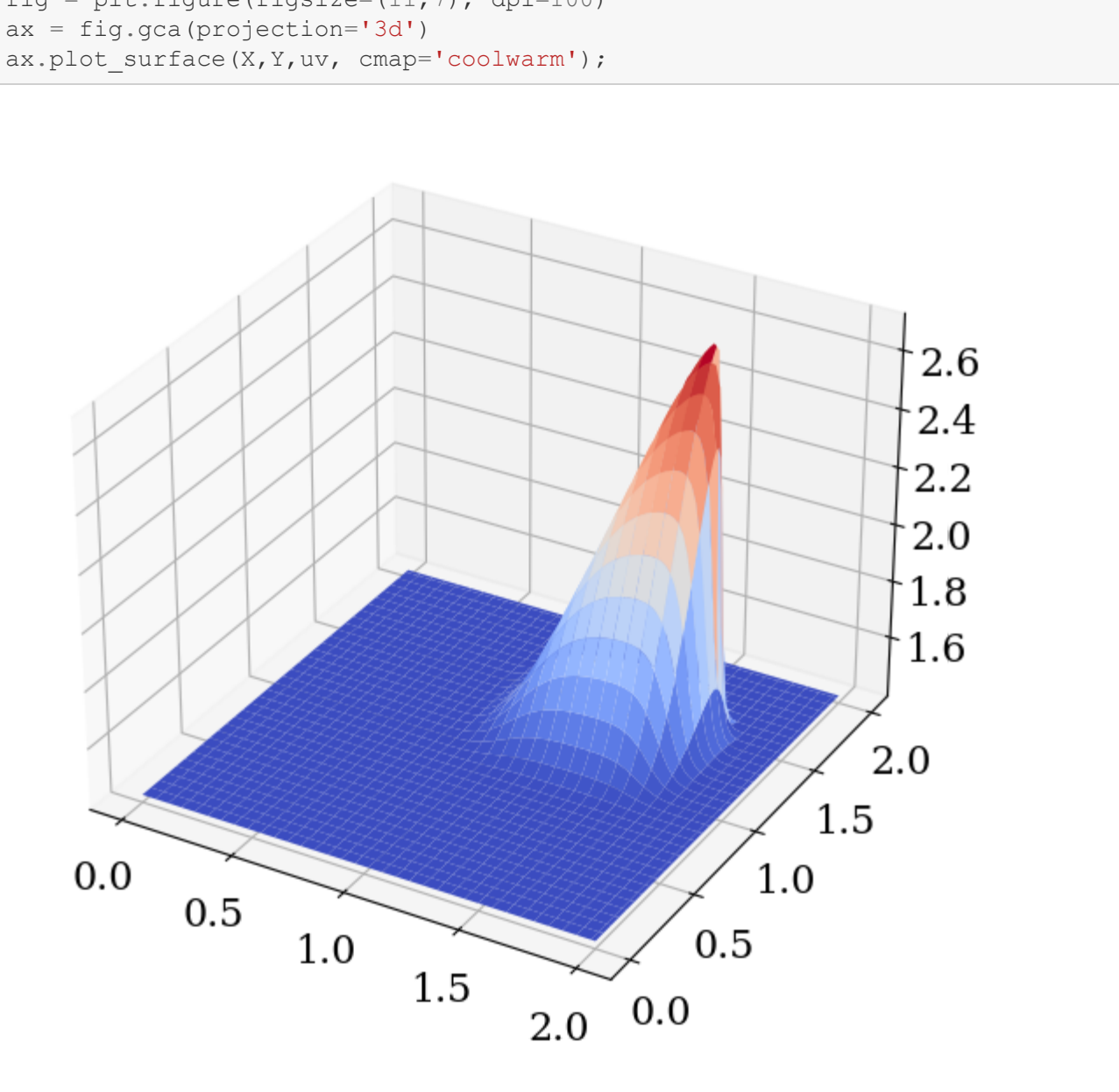
$$u_{i,j}^{n+1} = u_{i,j}^n - u_{i,j}\frac{\Delta t}{\Delta x}(u_{i,j}^n - u_{i-1,j}^n) - v_{i,j}^n\frac{\Delta t}{\Delta y}(u_{i,j}^n - u_{i,j-1}^n)$$
$$v_{i,j}^{n+1} = v_{i,j}^n - u_{i,j}\frac{\Delta t}{\Delta x}(v_{i,j}^n - v_{i-1,j}^n) - v_{i,j}^n\frac{\Delta t}{\Delta y}(v_{i,j}^n - v_{i,j-1}^n).$$

Escriban una función que resuelva este set de ecuaciones acopladas, y grafiquen $\sqrt{u^2+v^2}$ para $t = 0.5$, usando el mismo Δt que antes.

Como ven, es muy parecido al caso lineal, solamente que en la función vamos a evaluar dos ecuaciones en vez de una. La condición inicial de v es la misma que u:

In [20]:
```python
nx = 101
ny = 101
dx = 2./(nx-1)
dy = 2./(ny-1)
dt = 0.005
nt = int(0.5/dt)

x = numpy.linspace(0,2,nx)
y = numpy.linspace(0,2,ny)

u_inicial = numpy.ones((ny,nx))
u_inicial[int(.5/dy):int(1/dy+1),int(.5/dx):int(1/dx+1)]=2

v_inicial = numpy.ones((ny,nx))
v_inicial[int(.5/dy):int(1/dy+1),int(.5/dx):int(1/dx+1)]=2
```

In [24]:
```python
### ALUMNO
def ftbs_2d(u, v, dx, dy, dt, nt):

    for n in range(nt+1): ##loop across number of time steps
        un = u.copy()
        vn = v.copy()
        row, col = u.shape
        for i in range(1, row):
            for j in range(1, col):
                u[i,j] = un[i, j] - (un[i,j]*dt/dx*(un[i,j] - un[i-1,j]))-(vn[i,j]*dt/dy*(un[i,j]-un[i,
j-1]))
                v[i,j] = vn[i, j] - (un[i,j]*dt/dx*(vn[i,j] - vn[i-1,j]))-(vn[i,j]*dt/dy*(vn[i,j]-vn[i,
j-1]))

                u[0,:] = 1
                u[-1,:] = 1
                u[:,0] = 1
                u[:,-1] = 1
                v[0,:] = 1
                v[-1,:] = 1
                v[:,0] = 1
                v[:,-1] = 1
    return u,v

u = u_inicial.copy()
v = v_inicial.copy()
u,v = ftbs_2d(u,v, dx, dy, dt, nt)

uv = numpy.sqrt(u**2+v**2)

fig = plt.figure(figsize=(11,7), dpi=100)
ax = fig.gca(projection='3d')
ax.plot_surface(X,Y,uv, cmap="coolwarm");
```



Se comporta de manera similar a la convección no lineal en 1D pero en 2D.

Difusión

Finalmente, la ecuación de difusión es

$$\frac{\partial u}{\partial t} - \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0,$$

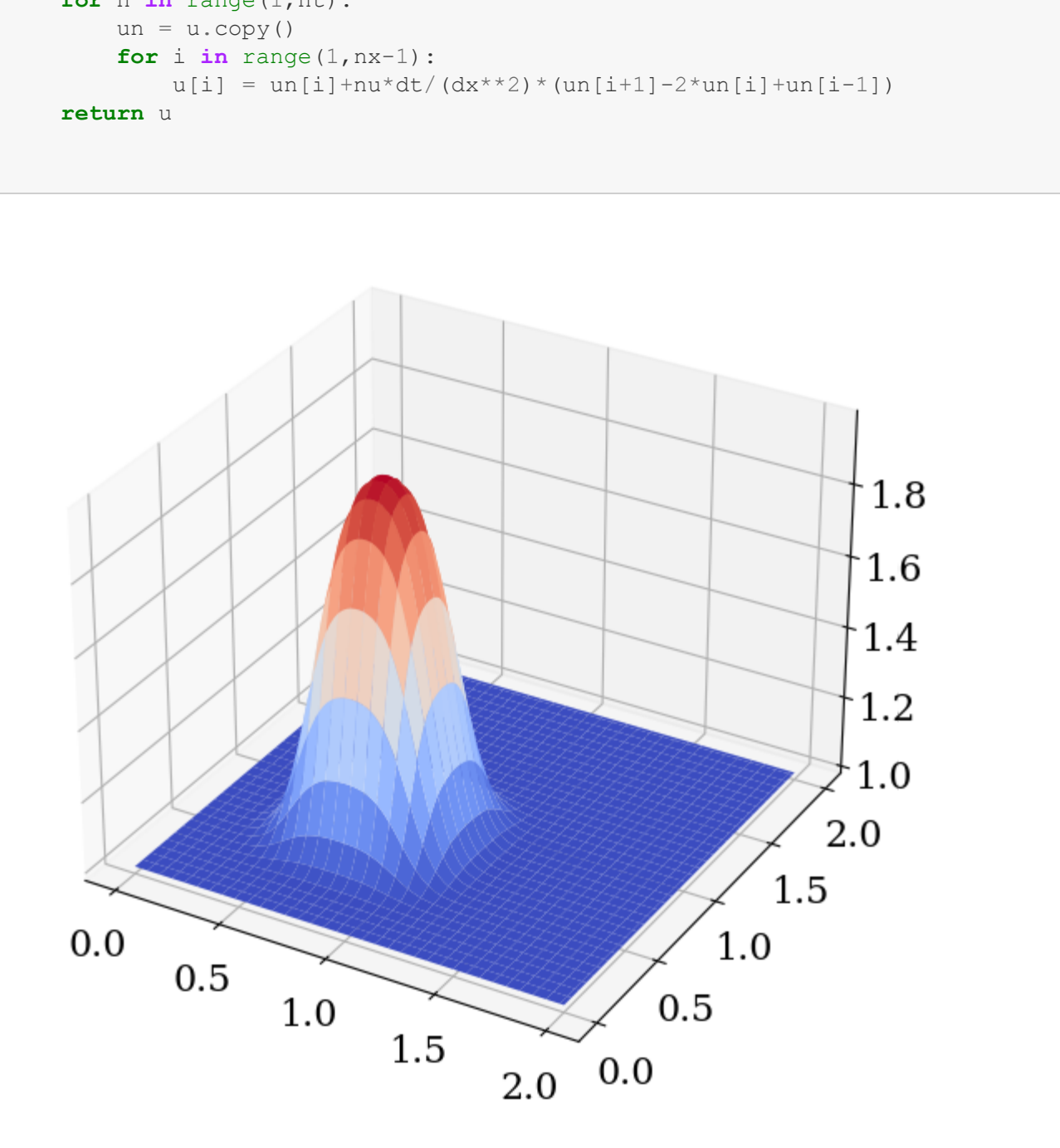
y se discretiza:

$$\frac{u^{n+1}_{i,j} - u^n_{i,j}}{\Delta t} = \nu\frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta x^2} + \nu\frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2}$$

Escriban una función que calcule numéricamente esta ecuación para la condición inicial cuadrada con $\nu = 0.5$, al tiempo $t = 0.01$ y grafiquen. Como sabemos que la ecuación de difusión tiene una condición CFL más estricta, usaremos un paso de tiempo muy pequeño $\Delta t = 0.0001$ ¡La próxima semana veremos formas de evitar este problema!

In [25]:
```python
nx = 101
ny = 101
dx = 2./(nx-1)
dy = 2./(ny-1)
dt = 0.0001
nt = int(0.01/dt)

nu = 0.5

x = numpy.linspace(0,2,nx)
y = numpy.linspace(0,2,ny)

u_inicial = numpy.ones((ny,nx))
u_inicial[int(.5/dy):int(1/dy+1),int(.5/dx):int(1/dx+1)]=2
```

In [26]:
```python
###ALUMNO
def diffuse(u, dx, dy, dt, nt, nu):

    for n in range(nt+1): ##loop across number of time steps
        un = u.copy()
        row, col = u.shape
        for i in range(1, row-1):
            for j in range(1, col-1):
                u[i,j] = un[i, j] +dt*nu*((1/dx**2)*(un[i+1,j]-2*un[i,j]+un[i-1,j])+(1/dy**2)*(un[i,j+1
]-2*un[i,j]+un[i,j-1]))
                u[0,:] = 1
                u[-1,:] = 1
                u[:,0] = 1
                u[:,-1] = 1

    return u

u = u_inicial.copy()
u = diffuse(u, dx, dy, dt, nt, nu)

fig = plt.figure(figsize=(11,7), dpi=100)
ax = fig.gca(projection='3d')
ax.plot_surface(X,Y,u, cmap="coolwarm");

def ftcs(u, nt, nx, nu, dt, dx):
    for n in range(nt):
        un = u.copy()
        for i in range(1,nx-1):
            u[i] = un[i]+nu*dt/(dx**2)*(un[i+1]-2*un[i]+un[i-1])
    return u
```



Se nota que la onda queda en el mismo lugar pero existiendo un proceso de difusión. Coincide con la difusión en 1D.

In [ ]: