Partial Differential Equations

Computational Methods, Nov. 2017, Kenji Doya

References:

- Svein Linge & Hans Petter Langtangen: Programming for Computations – Python. Springer (2016).
    - Chapter 5: Solving Partial Differential Equations

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from mpl_toolkits.mplot3d import Axes3D
```

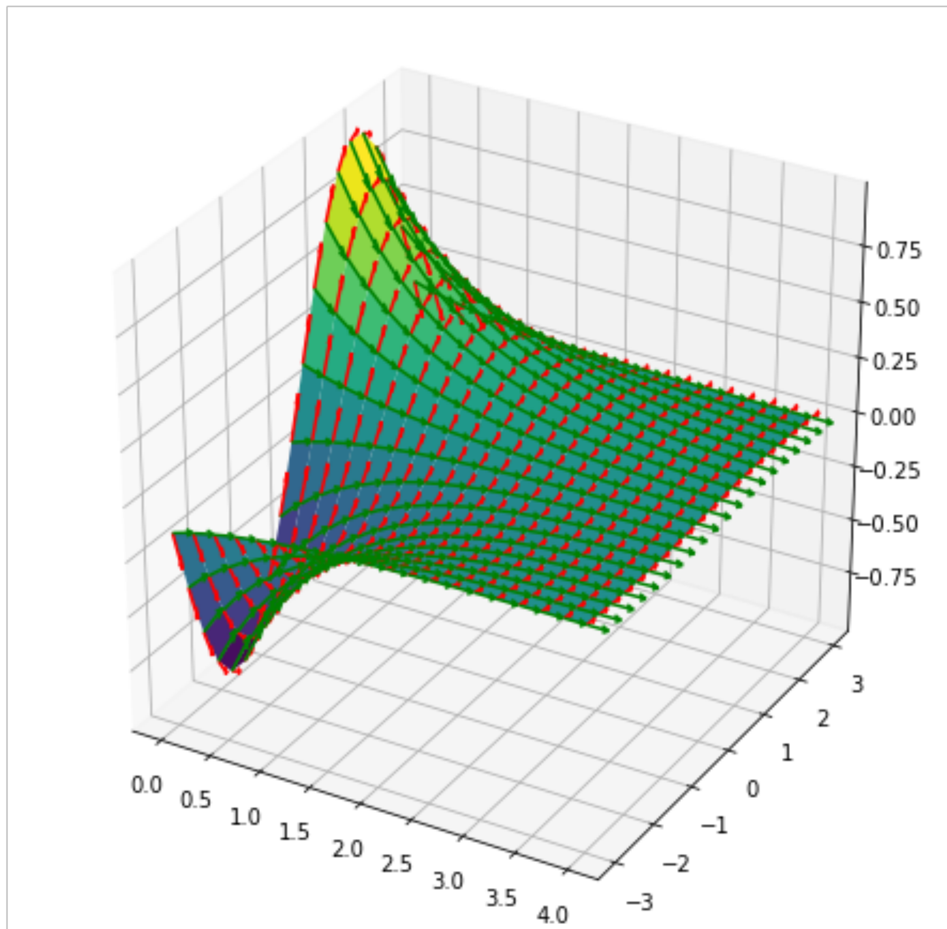
Partial derivative

For a function with multiple inputs $f(x_1, \ldots, x_n)$, a *partial derivative*
$$\frac{\partial f(x_1, \ldots, x_n)}{\partial x_i}$$
is the derivative with respect to an input x_i while other inputs are held constant.

For example, for
$$f(x, y) = e^{-x} \sin y,$$
partial derivatives are
$$\frac{\partial f(x, y)}{\partial x} = -e^{-x} \sin y$$
$$\frac{\partial f(x, y)}{\partial y} = e^{-x} \cos y$$

In [2]:
```
x, y = np.meshgrid(np.linspace(0, 4, 20), np.linspace(-3, 3, 20))
f = np.exp(-x) * np.sin(y)
dfdx = -np.exp(-x) * np.sin(y)
dfdy = np.exp(-x) * np.cos(y)
# 3D plot
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, f, cmap='viridis')
x0 = np.zeros_like(x)
x1 = np.ones_like(x)
ax.quiver(x, y, f, x1, x0, dfdx, color='g', length=0.2)
ax.quiver(x, y, f, x0, x1, dfdy, color='r', length=0.2)
```

Out[2]: <mpl_toolkits.mplot3d.art3d.Line3DCollection at 0x113cbde80>



Partial Differential Equation (PDE)

A *partial differential equation (PDE)* is an equation that includes partial derivatives $\frac{\partial f(x_1,\ldots,x_n)}{\partial x_i}$ of an unknown function $f(x)$.

The most typical example is the *diffusion equation* (or heat equation)

$$\frac{\partial y}{\partial t} = D\frac{\partial^2 y}{\partial x^2} + g(y,x,t)$$

that describes the evolution of concentration (or temperature) y in space x and time t with input $g(y,x,t)$ and the diffusion coefficient D.

Analytic Solution of PDE

For a diffusion equation without input

$$\frac{\partial y}{\partial t} = D\frac{\partial^2 y}{\partial x^2},$$

the solution is given by *separation of variables*.

By assuming that the solution is a product of temporal and spatial components $y(x,t) = u(t)v(x)$, the PDE becomes

$$\frac{\partial u(t)}{\partial t}v(x) = Du(t)\frac{\partial^2 v(x)}{\partial x^2}$$
$$\frac{1}{Du(t)}\frac{\partial u(t)}{\partial t} = \frac{1}{v(x)}\frac{\partial^2 v(x)}{\partial x^2}$$

For this equation to hold for any t and x, a possible solution is for both sides to be a constant C. Then we have two separate ODEs:

$$\frac{du(t)}{dt} = CDu(t)$$
$$\frac{d^2 v(x)}{dx^2} = Cv(x)$$

for which we know analitic solutions.

By setting $C = -b^2 \le 0$, we have

$$u(t) = C_0 e^{-b^2 Dt},$$
$$v(x) = C_1 \sin bx + C_2 \cos bx.$$

Thus we have a solution

$$y(x,t) = e^{-b^2 Dt}(C_1 \sin bx + C_2 \cos bx)$$

where b, C_1 and C_2 are determined by the initial condition $y(x,0)$.
The equation tells us that higher spatial frequency components decay quiker.

Boundary condition

For uniquely specifying a solution of a PDE in a bounded area, e.g., $x_0 < x < x_1$, we need to specify either

- the value $y(x, t)$ (Dirichlet boundary condition)
- or the derivative $\frac{\partial y(x,t)}{\partial x}$ (Neumann boundary condition)

at the boundary x_0 and x_1 to uniquely determine the solution.

In [ ]:

```

```

From PDE to ODE

The standard way of dealing with space and time on a digital computer is to discretize them. For a PDE

$$\frac{\partial y}{\partial t} = D\frac{\partial^2 y}{\partial x^2} + g(y, x, t),$$

defined in a range $x_0 \leq x \leq x_0 + L$, we consider a spatial discretization by $\Delta x = L/N$

$$x_i = x_0 + i\Delta x$$

$(i = 1, \dots, N)$ and

$$y_i(t) = y(x_i, t).$$

Then the PDE can be approximated by a set of ODEs

$$\frac{dy_i}{dt} = D\frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} + g(y_i, x_i, t).$$

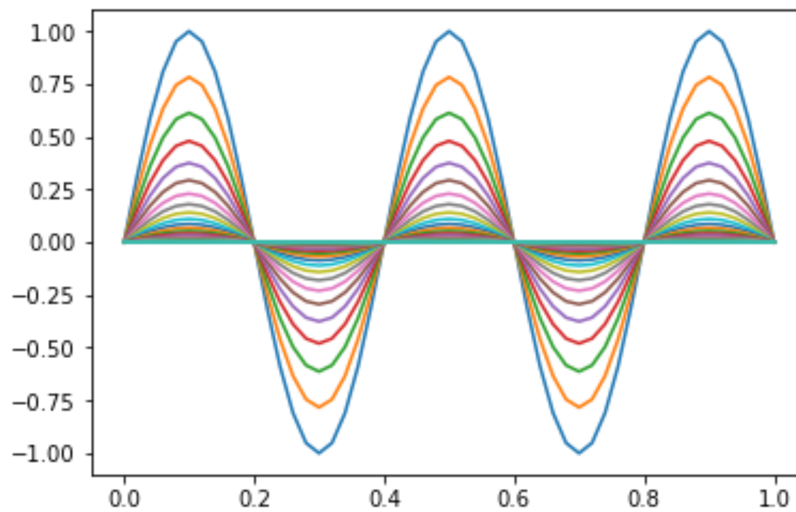
In [3]:

```
from scipy.integrate import odeint
```

In [4]:

```
def diff1D(y, t, x, D, input=None):
    """1D Diffusion equaiton with constant boundary condition
    y: state vector
    t: time
    x: positions
    D: diffusion coefficient
    input: function(y,x,t)"""
    # finite different approximation
    # shift to left and right
    d2ydx2 = (y[:-2] -2*y[1:-1] + y[2:])/(x[1] - x[0])**2
    d2ydx2 = np.hstack((0, d2ydx2, 0))  # add 0 to both ends
    if input == None:
        return(D*d2ydx2)
    else:
        return(D*d2ydx2 + input(y, x, t))
```

In [14]:
```
N = 50
x = np.linspace(0, 1, N+1)
y0 = np.zeros_like(x)  # initial condition
y0[50] = 1
y0 = np.sin(np.pi*x)
y0 = np.sin(5*np.pi*x)
t = np.arange(0, 1, 0.01)
y = odeint(diff1D, y0, t, (x, 0.1))
p = plt.plot(x, y.T)
```

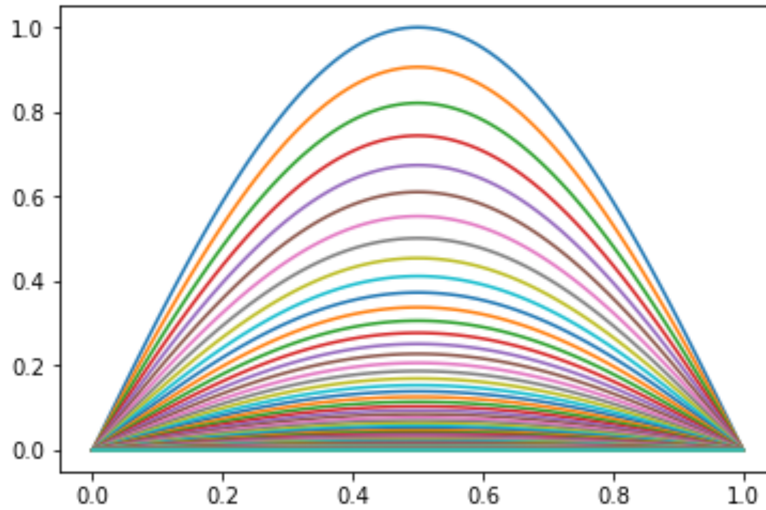


Stability

For an accurate solution, Δx have to be small. But making Δx small can make the ODE *stiff* such that solution can be unstable.
It is known that stable solution by Euler method requires

$$\Delta t \leq \frac{\Delta x^2}{2D}.$$

In [18]:
```
N = 100
x = np.linspace(0, 1, N+1)
y0 = np.sin(np.pi*x)
t = np.arange(0, 1, 0.01)
y = odeint(diff1D, y0, t, (x, 1), hmin=0)
p = plt.plot(x, y.T)
```

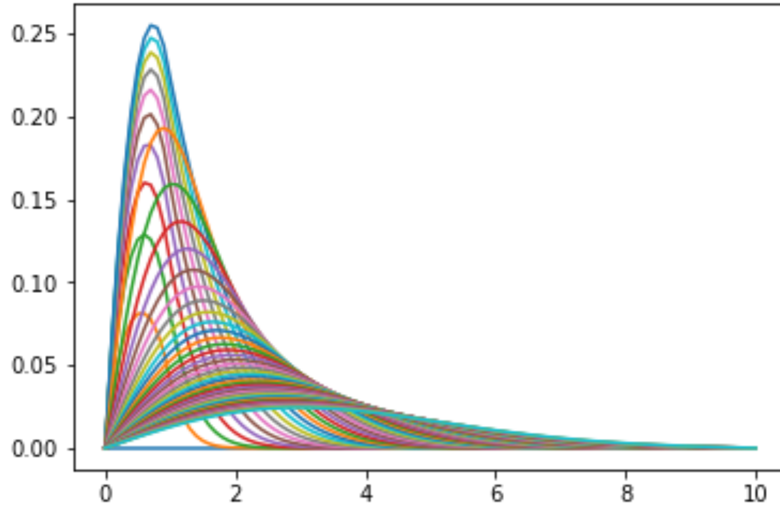


In [ ]:
```
```

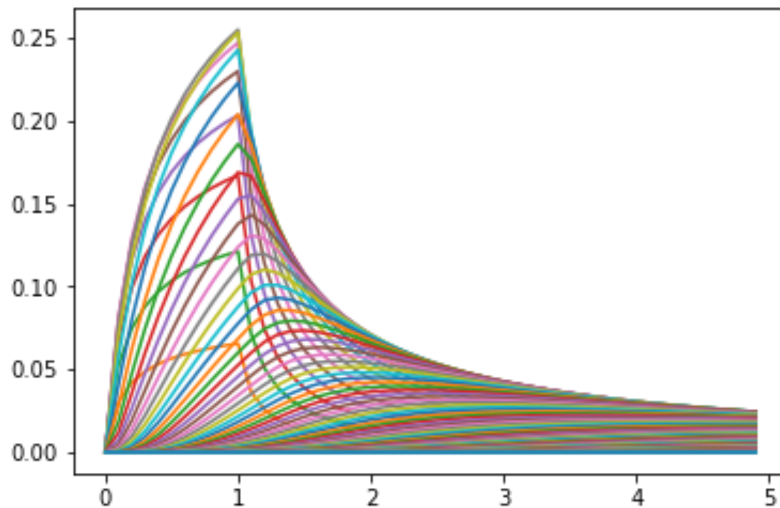
Let us see the case with an input.

In [19]:
```
def pulse(y, x, t):
    """1 for 0<x<1 at 0<t<1"""
    return( (x>0)*(x<1)*(t>0)*(t<1)*1.)
```

In [23]:
```
N = 100
x = np.linspace(0, 10, N+1)
y0 = np.zeros_like(x)  # initial condition
t = np.arange(0, 5, 0.1)
y = odeint(diff1D, y0, t, (x, 1, pulse))
p = plt.plot(x, y.T)
```

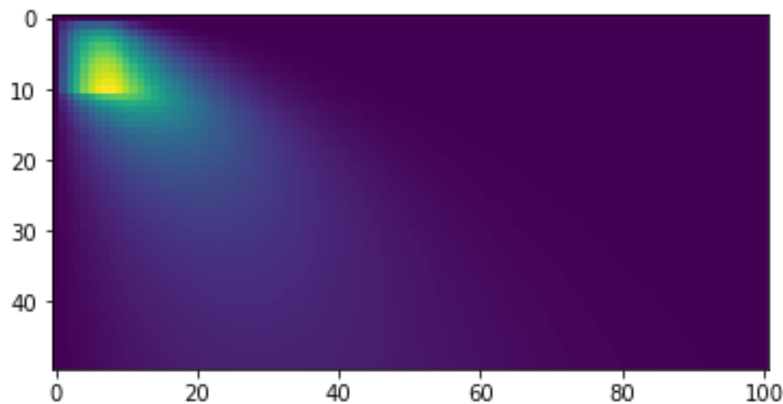


In [24]:
```
p = plt.plot(t, y)
```



In [25]:
```
plt.imshow(y)
```

Out[25]: <matplotlib.image.AxesImage at 0x11aa82e80>



In [ ]:

Wave Equation

If y has a second-order dynamics, there can be traveling waves. For example $y = (u, v)$ for position and velocity

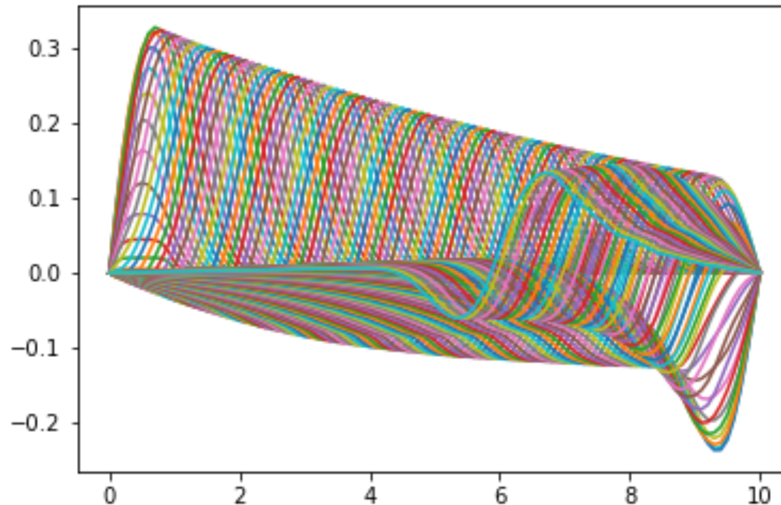
$$\frac{\partial u}{\partial t} = v$$

$$\frac{\partial v}{\partial t} = -ku - cv + D\frac{\partial^2 u}{\partial x^2} + g(u, x, t),$$

where k is stiffness and c is damping coefficients.

In [26]:
```
def wave1D(y, t, x, k, c, D, input=None):
    """1D wave equaiton with constant boundary
    y: state vector hstack(u, v)
    t: time
    x: positions
    k: spring coefficient
    c: damping coefficient
    D: diffusion coefficient
    input: function(y,x,t)"""
    n = int(len(y)/2)
    u, v = y[:n], y[n:]
    # finite different approximation
    # shift to left and right
    d2udx2 = (u[:-2] -2*u[1:-1] + u[2:])/(x[1] - x[0])**2
    d2udx2 = np.hstack((0, d2udx2, 0))  # add 0 to both ends
    if input == None:
        return(np.hstack((v, -k*u -c*v + D*d2udx2)))
    else:
        return(np.hstack((v, -k*u -c*v + D*d2udx2 + input(y, x, t))))
```

In [30]:
```
N = 100
x = np.linspace(0, 10, N+1)
y0 = np.zeros(2*(N+1))  # initial condition
t = np.arange(0, 15, 0.1)
y = odeint(wave1D, y0, t, (x, 0.2, 0.1, 1, pulse))
p = plt.plot(x, y[:,:N+1].T)
```



In [ ]:

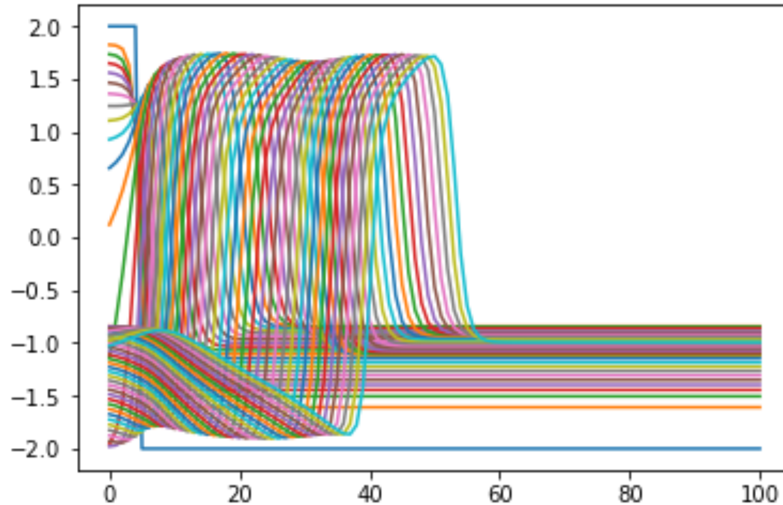
Reaction-Diffusion Equation

Linear waves can decay, reflect, and overlay.
Some nonlinear waves, like neural spike, can travel without decaying. FitzHugh-Nagumo model can be embedded in a diffusive axon model

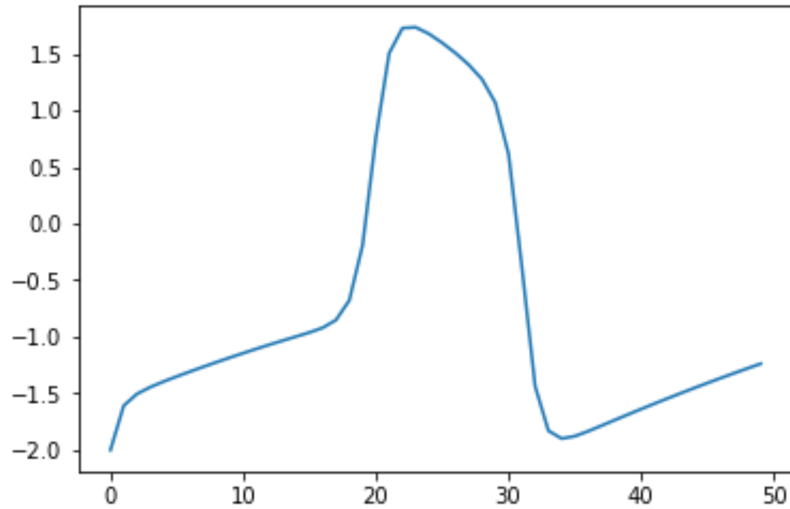
$$dv/dt = v - \frac{v^3}{3} - w + D\frac{\partial^2 v}{\partial x^2} + I$$
$$dw/dt = \phi(v + a - bw)$$

In [31]:
```
def fhnaxon(y, t, x, I=0, D=1, a=0.7, b=0.8, phi=0.08):
    """FitzHugh-Nagumo axon model
    y: state vector hstack(v, w)
    t: time
    x: positions
    I: input current
    D: diffusion coefficient
    """
    n = int(len(y)/2)
    v, w = y[:n], y[n:]
    # finite difference approximation
    d2vdx2 = (v[:-2] -2*v[1:-1] + v[2:])/(x[1] - x[0])**2
    d2vdx2 = np.hstack((0, d2vdx2, 0))  # add 0 to both ends
    dvdt = v - v**3/3 - w + D*d2vdx2 + I
    dwdt = phi*(v + a -b*w)
    return(np.hstack((dvdt, dwdt)))
```

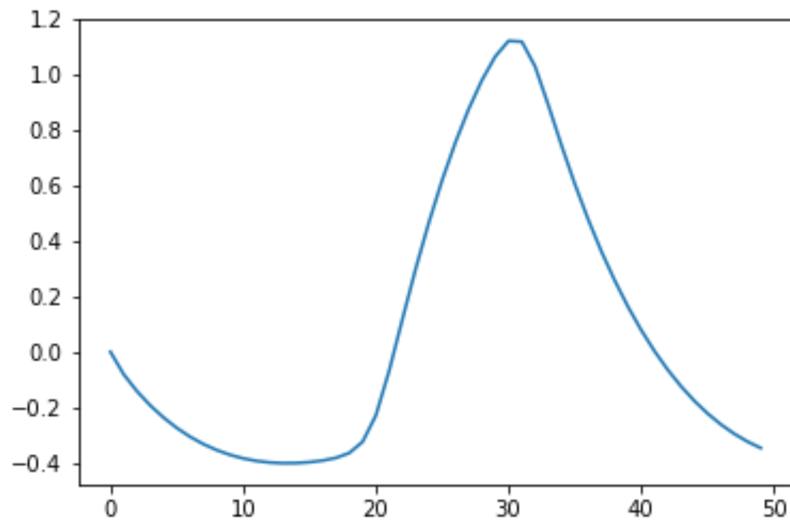
In [42]:
```
N = 100
x = np.linspace(0, 100, N+1)
y0 = np.zeros(2*(N+1))  # initial condition
y0[0:N+1] = -2
y0[0:5] = 2
t = np.arange(0, 50, 1)
y = odeint(fhnaxon, y0, t, (x, 0.3))
p = plt.plot(x, y[:,:N+1].T) # plot in space
```



In [46]:
```
p = plt.plot(t, y[:,20])  # plot in time
```

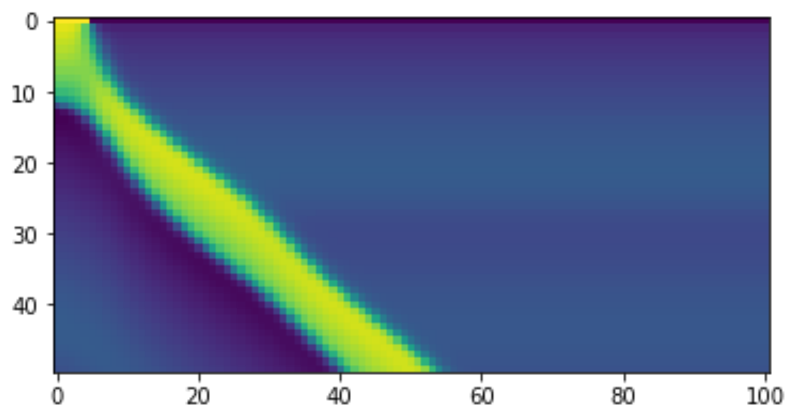


In [48]:
```
p = plt.plot(t, y[:,N+21])  # plot in time
```



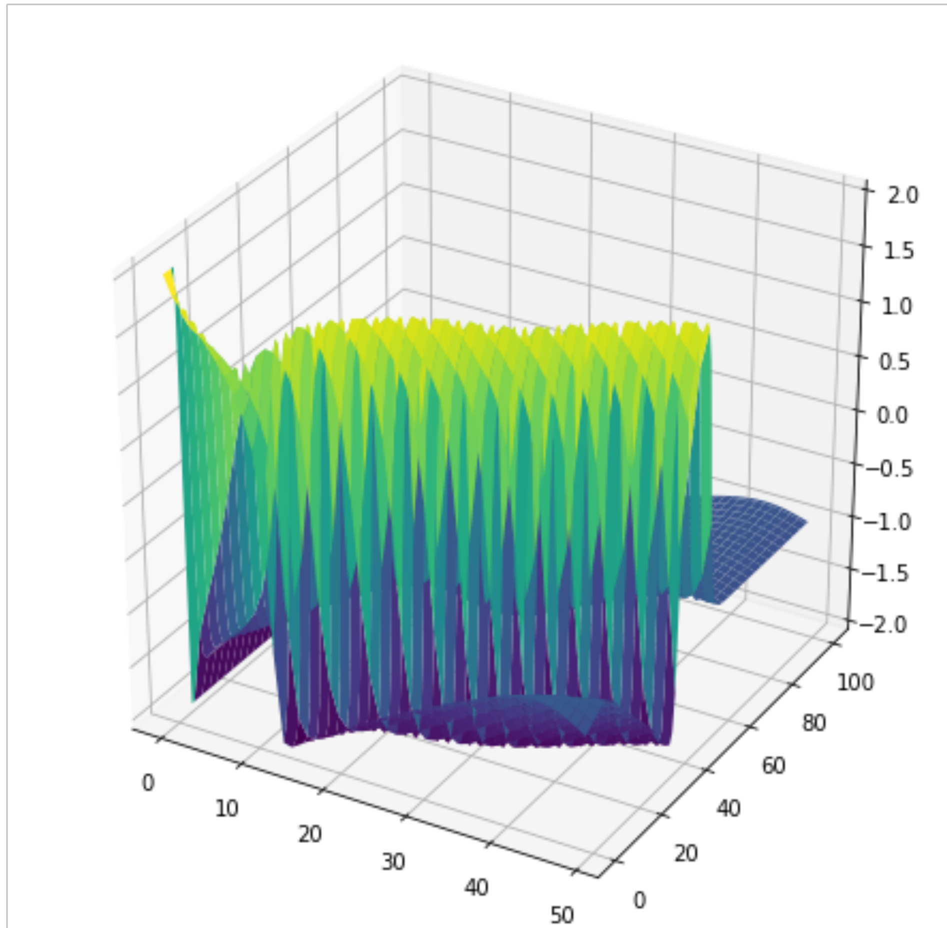
In [50]:
```
plt.imshow(y[:,:N+1])
```

Out[50]: <matplotlib.image.AxesImage at 0x11a9adbe0>



In [68]:
```
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
T, X = np.meshgrid(t, x)
ax.plot_surface(T, X, y[:,:N+1].T, cmap='viridis')
```

Out[68]: <mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x11a74b9b0>



In [ ]: