A8

Alice Gee, ag67642
Andrew Yang, ay6764
Mohammad Aga, mba929

Exercise 6.52 Now consider the 2D wave equation

$$u_{tt} = c\left(u_{xx} + u_{yy}\right).$$

We want to build a numerical solution to this new PDE. Just like with the 2D heat equation we propose the notation $U_{i,j}^n$ for the approximation of the function $u(t, x, y)$ at the point $t = t_n$, $x = x_i$, and $y = y_j$.

a. Give discretizations of the derivatives u_{tt}, u_{xx}, and u_{yy}.
b. Substitute your discretizations into the 2D wave equation, make the simplifying assumption that $\Delta x = \Delta y$, and solve for $U_{i,j}^{n+1}$. This is the finite difference scheme for the 2D wave equation.
c. Write code to implement the finite difference scheme from part (b) on the domain $(x, y) \in (0,1) \times (0,1)$ with homogeneous Dirichlet boundary conditions, initial condition $u(0, x, y) = \sin(2\pi(x - 0.5))\sin(2\pi(y - 0.5))$, and zero initial velocity.
d. Draw the finite difference stencil for the 2D heat equation.

a

$u_{tt} = \frac{u(t+\Delta t)-2u(t)+u(t-\Delta t)}{\Delta t^2}$
$u_{xx} = \frac{u(x+\Delta x)-2u(x)+u(x-\Delta x)}{\Delta x^2}$
$u_{yy} = \frac{u(y+\Delta y)-2u(y)+u(y-\Delta y)}{\Delta y^2}$

b

$\frac{u(t+\Delta t)-2u(t)+u(t-\Delta t)}{\Delta t^2}$ = c $\left(\frac{u(x+\Delta x)-2u(x)+u(x-\Delta x)}{\Delta x^2} + \frac{u(y+\Delta y)-2u(y)+u(y-\Delta y)}{\Delta y^2}\right)$
$\frac{u(t+\Delta t)-2u(t)+u(t-\Delta t)}{\Delta t^2}$ = c $\left(\frac{u(x+\Delta x)-2u(x)+u(x-\Delta x)+u(y+\Delta y)-2u(y)+u(y-\Delta y)}{\Delta x^2}\right.$, since Δx = Δy
$\frac{u_{i,j}^{n+1}-2u_{i,j}^{n}+u_{i,j}^{n-1}}{\Delta t^2}$ = c$\left(\frac{u_{i+1,j}^{n}-2u_{i,j}^{n}+u_{i-1,j}^{n}+u_{i,j+1}^{n}-2u_{i,j}^{n}+u_{i,j-1}^{n}}{\Delta x^2}\right)$
$\frac{u_{i,j}^{n+1}-2u_{i,j}^{n}+u_{i,j}^{n-1}}{\Delta t^2}$ = c$\left(\frac{u_{i+1,j}^{n}-4u_{i,j}^{n}+u_{i-1,j}^{n}+u_{i,j+1}^{n}+u_{i,j-1}^{n}}{\Delta x^2}\right)$
$u_{i,j}^{n+1} = \frac{c\Delta t^2}{\Delta x^2}\left(u_{i+1,j}^{n} - 4u_{i,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}\right) + 2u_{i,j}^{n} - u_{i,j}^{n-1}$

c

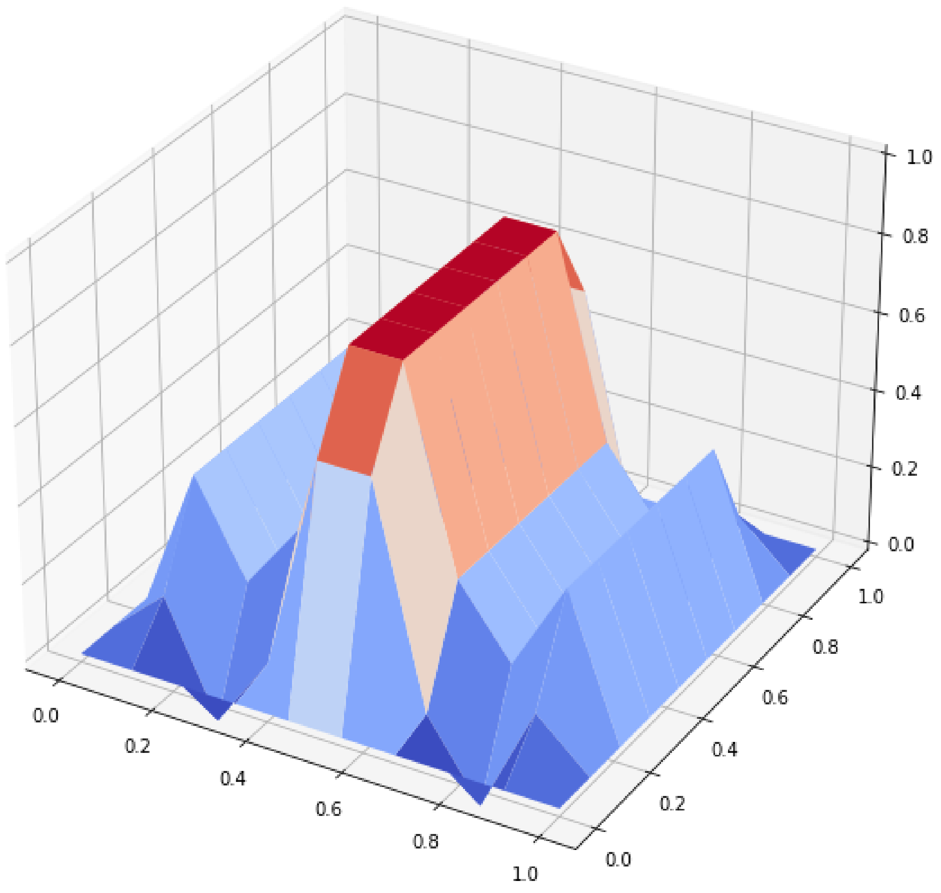
In [59]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm # this allows for color maps
from ipywidgets import interactive

def wrapper (x, y, u, t, X, Y):
    def plotter(Frame):
        fig = plt.figure(figsize=(12,10))
        ax = fig.gca(projection='3d')
        ax.plot_surface(X,Y,u[Frame,:,:], cmap=cm.coolwarm)
        ax.set_zlim(0,1)
        plt.show()
    interactive_plot = interactive(plotter, Frame=(0,len(t)))
    display (interactive_plot)

def main():
    # create an array for t starting at t = 0 and ending at t = 1
    # divide this interval into 10 equal parts
    t = np.linspace(0,1,10)
    dt = 0.1

    # create an array for x and y starting at x = 0 and ending at x = 1
    # divide this interval into 10 equal parts
    x = np.linspace(0,1,10)
    y = x
    dx = 0.1

    X, Y = np.meshgrid(x,y)

    # define the constant
    c = 0.5

    # define parameter a as
    a = c * (dt / dx) ** 2

    # print the values
    print ("dt = ", dt, ", dx = ", dx, ", a = ", a,)

    # build array U to store all approximations at all times and
    # at all spatial points
    u = np.zeros ((len(t), len(x), len(y)))

    # enforce the left boundary condition
    u[:,:, 0] = 0 # y initial
    u[:,0, :] = 0 # x initial

    # enforce the right boundary condition
    u[:, :, -1] = 0 # y initial
    u[:, -1, :] = 0 # x initial

    # enforce the initial condition
    u[0,:, :] = np.sin(2*np.pi*(x - 0.5))*np.sin(2*np.pi*(y - 0.5))
    u[1, :, :] = u[0,:, :]

    # fill the array U one row at a time, leave the boundary conditions
    # fixed and fill indices 1 through -2
    for n in range (1, len(t) - 1):
        u[n + 1, 1:-1, 1:-1] = a*(u[n, 2:,1:-1] - 4*u[n, 1:-1, 1:-1] + u[n, :-2, 1:-1
                                  u[n, 1:-1, :-2]) + 2*u[n, 1:-1, 1:-1] - u[n-1, 1:-1

    # plot the solution
    wrapper (x, y, u, t, X, Y)
```

```
main()
```

```
dt =  0.1 , dx =  0.1 , a =  0.5
```

Frame 3

```
/var/folders/sx/gw6n5mnj28x21kcmksnp_p240000gn/T/ipykernel_55877/494104814.py:9:
MatplotlibDeprecationWarning: Calling gca() with keyword arguments was deprecated
in Matplotlib 3.4. Starting two minor releases later, gca() will take no keyword
arguments. The gca() function should only be used to get the current axes, or if
no axes exist, create new axes with default keyword arguments. To create a new ax
es with non-default arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```



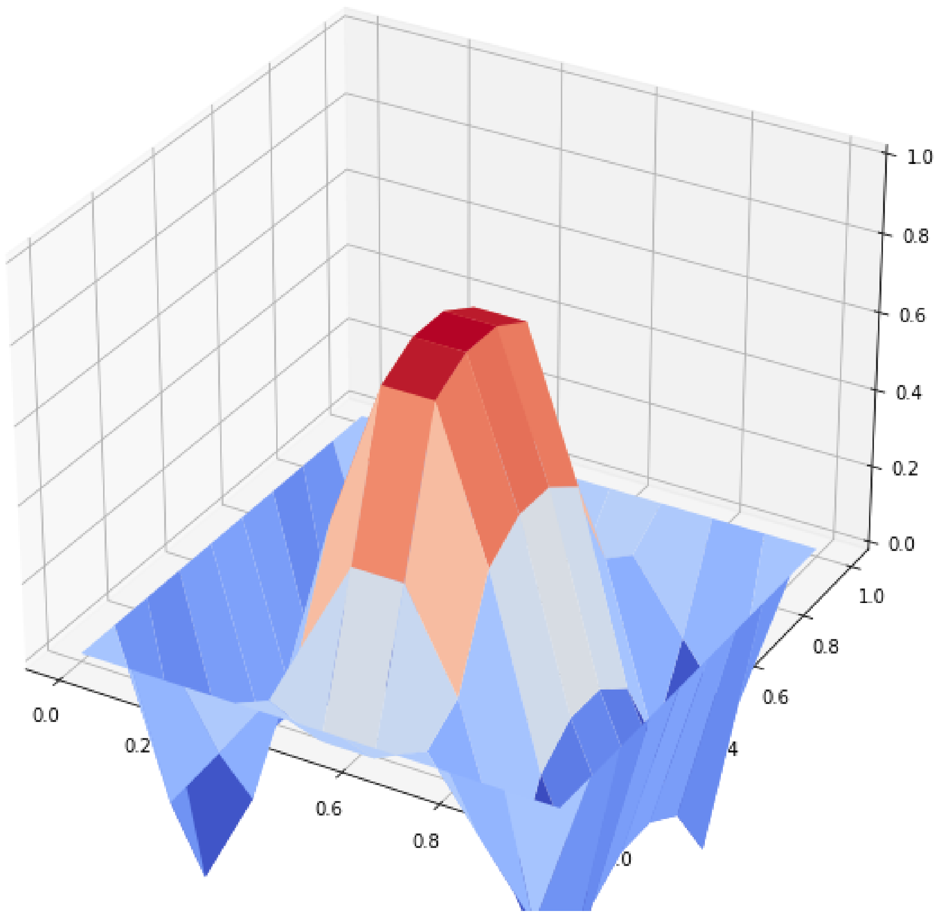
In [60]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm # this allows for color maps
from ipywidgets import interactive

def wrapper (x, y, u, t, X, Y):
    def plotter(Frame):
        fig = plt.figure(figsize=(12,10))
        ax = fig.gca(projection='3d')
        ax.plot_surface(X,Y,u[Frame,:,:], cmap=cm.coolwarm)
        ax.set_zlim(0,1)
        plt.show()
    interactive_plot = interactive(plotter, Frame=(0,len(t)))
    display (interactive_plot)

def main():
    # create an array for t starting at t = 0 and ending at t = 1
    # divide this interval into 10 equal parts
    t = np.linspace(0,1,10)
    dt = 0.1

    # create an array for x and y starting at x = 0 and ending at x = 1
    # divide this interval into 10 equal parts
    x = np.linspace(0,1,10)
    y = x
    dx = 0.1

    X, Y = np.meshgrid(x,y)

    # define the constant
    c = 0.5

    # define parameter a as
    a = c * (dt / dx) ** 2

    # print the values
    print ("dt = ", dt, ", dx = ", dx, ", a = ", a,)

    # build array U to store all approximations at all times and
    # at all spatial points
    u = np.zeros ((len(t), len(x), len(y)))

    # enforce the left boundary condition
    u[:,:, 0] = 0 # y initial
    u[:,0, :] = 0 # x initial

    # enforce the right boundary condition
    u[:, :, -1] = 0 # y initial
    u[:, -1, :] = 0 # x initial

    # enforce the initial condition
    u[0,:, :] = np.sin(2*np.pi*(x - 0.5))*np.sin(2*np.pi*(y - 0.5))
    u[1, :, :] = u[0,:, :]

    # fill the array U one row at a time, leave the boundary conditions
    # fixed and fill indices 1 through -2
    for n in range (1, len(t) - 1):
        u[n + 1, 1:-1, 1:-1] = a*(u[n, 2:,1:-1] - 4*u[n, 1:-1, 1:-1] + u[n, :-2, 1:-1
                                  u[n, 1:-1, :-2]) + 2*u[n, 1:-1, 1:-1] - u[n-1, 1:-1

    # plot the solution
    wrapper (x, y, u, t, X, Y)
```

```
main()
```

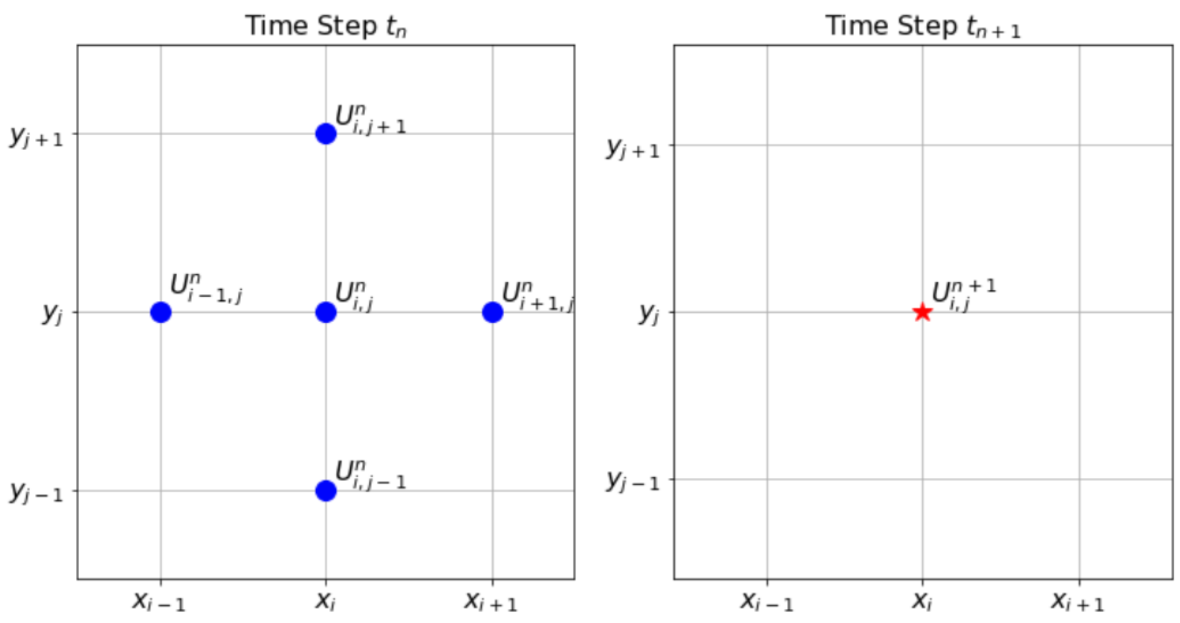
```
dt =  0.1 , dx =  0.1 , a =  0.5
```

Frame 5

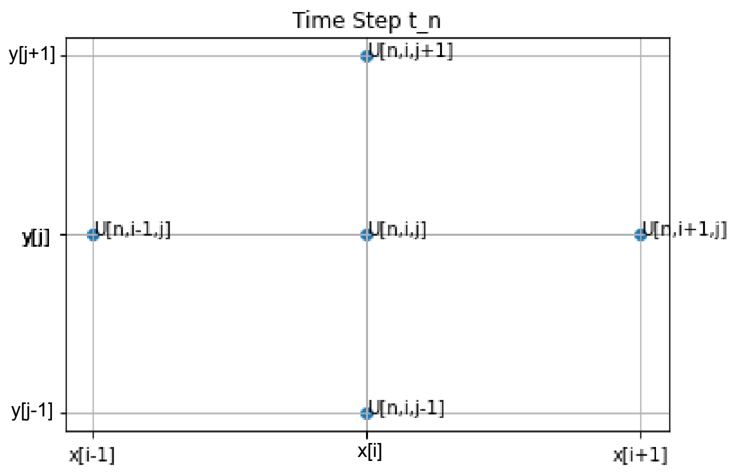
```
/var/folders/sx/gw6n5mnj28x21kcmksnp_p240000gn/T/ipykernel_55877/494104814.py:9:
MatplotlibDeprecationWarning: Calling gca() with keyword arguments was deprecated
in Matplotlib 3.4. Starting two minor releases later, gca() will take no keyword
arguments. The gca() function should only be used to get the current axes, or if
no axes exist, create new axes with default keyword arguments. To create a new ax
es with non-default arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```



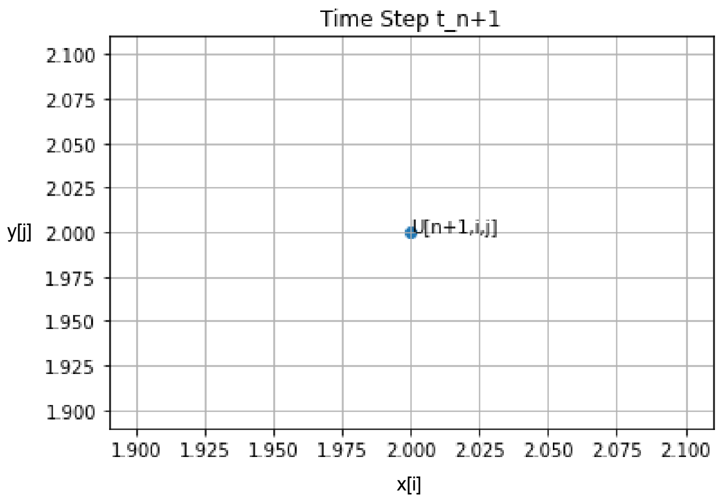
d


Time Step t_n
$U^n_{i,j+1}$
$U^n_{i-1,j}$
$U^n_{i,j}$
$U^n_{i+1,j}$
$U^n_{i,j-1}$
y_{j+1}
y_j
y_{j-1}
x_{i-1}
x_i
x_{i+1}
Time Step t_{n+1}
$U^{n+1}_{i,j}$
y_{j+1}
y_j
y_{j-1}
x_{i-1}
x_i
x_{i+1}

In [71]:
```
import numpy as np
import matplotlib.pyplot as plt
import itertools

x = [1, 2, 2, 2, 3]
y = [2, 3, 2, 1, 2]
p_labels = ["U[n,i-1,j]","U[n,i,j+1]", "U[n,i,j]", "U[n,i,j-1]", "U[n,i+1,j]"]

fig = plt.figure()
ax = fig.gca()
plt.scatter(x, y)

for i, txt in enumerate(p_labels):
    ax.annotate(txt, (x[i], y[i]))

ax_labels = ['x[i-1]','' ,'x[i]','', 'x[i+1]']
ay_labels = ['y[j-1]','','y[j+1]','', 'y[j]']
plt.xticks(x, ax_labels, rotation='horizontal')
plt.yticks(y, ay_labels, rotation='horizontal')

plt.title("Time Step t_n")
plt.grid()
plt.show()
```



In [62]:
```
x = [2]
y = [2]
p_labels = ["U[n+1,i,j]"]

fig = plt.figure()
ax = fig.gca()
plt.scatter(x, y)

for i, txt in enumerate(p_labels):
    ax.annotate(txt, (x[i], y[i]))

plt.title("Time Step t_n+1")
plt.grid()
plt.show()
```



Exercise 6.58 Consider the traveling wave equation $u_t + vu_x = 0$ with initial condition $u(0, x) = f(x)$ for some given function f and boundary condition $u(t, 0) = 0$. To build a numerical solution we will again adopt the notation U_i^n for the approximation to $u(t, x)$ at the point $t = t_n$ and $x = x_i$.

a. Write an approximation of u_t using U_i^{n+1} and U_i^n.
b. Write an approximation of u_x using U_{i+1}^n and U_i^n.
c. Substitute your answers from parts (a) and (b) into the traveling wave equation and solve for U_i^{n+1}. This is our first finite difference scheme for the traveling wave equation.
d. Write Python code to get the finite difference approximation of the solution to the PDE. Plot your finite difference solution on top of the analytic solution for $f(x) = e^{-(x-4)^2}$. What do you notice? Can you stabilize this method by changing the values of Δt and Δx like with did with the heat and wave equations?

a

$$u_t = \frac{u_i^{n+1} - u_i^n}{\Delta t}$$

b

$$u_x = \frac{u_{i+1}^n - u_i^n}{\Delta x}$$

c

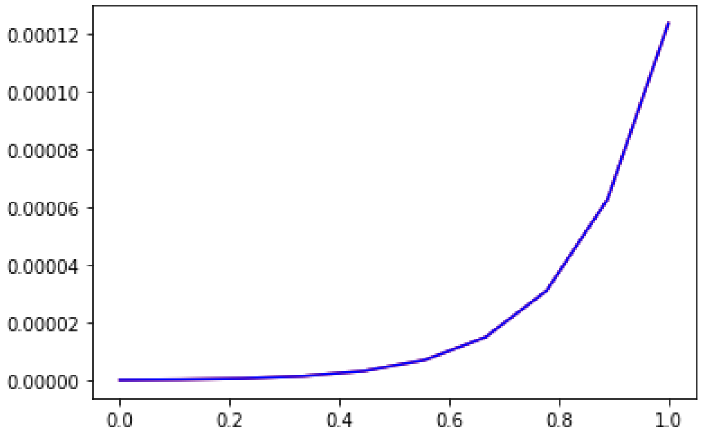
0 = $\frac{u_i^{n+1} - u_i^n}{\Delta t}$ + v$\frac{u_{i+1}^n - u_i^n}{\Delta x}$
u_i^{n+1} = $u_i^n - \frac{v\Delta t}{\Delta x}(u_{i+1}^n - u_i^n)$
since v = $\frac{\Delta x}{\Delta t}$ --> u_i^{n+1} = $2u_i^n - u_{i+1}^n$

d

In [63]:
```
import numpy as np
import matplotlib.pyplot as plt

# create an array for t starting at t = 0 and ending at t = 1
# divide this interval into 10 equal parts
t = np.linspace(0, 1, 10)
dt = 0.1

# create an array for x starting at x = 0 and ending at x = 1
# divide this interval into 10 equal parts
x = np.linspace(0, 1, 10)
dx = 0.1

# print the values
print ("dt = ", dt, ", dx = ", dx)

# build array U to store all approximations at all times and
# at all spatial points
u = np.zeros ((len(t), len(x)))

# enforce the left boundary condition
u[:, 0] = 0

# enforce the right boundary condition
u[:, -1] = 0

# enforce the initial condition
## when t = 0
u[0,:] = np.exp(-(x-4)**2)

# fill the array U one row at a time, leave the boundary conditions
# fixed and fill indices 1 through -2
for n in range (len(t) - 1):
    #u[n+1,1:-1] = u[n,1:-1] - a*(u[n, 2:] - u[n,1:-1])
    u[n+1,1:-1] = 2*u[n,1:-1] - u[n, 2:]

y = np.exp(-(x-4)**2)

# plot the solution
plt.plot(x, u[0], color = "red", label = "u solution" )
plt.plot(x, y, color = "blue", label = "analytical solution")
```

```
dt =  0.1 , dx =  0.1
```

Out[63]:
```
[<matplotlib.lines.Line2D at 0x11b0fbdf0>]
```

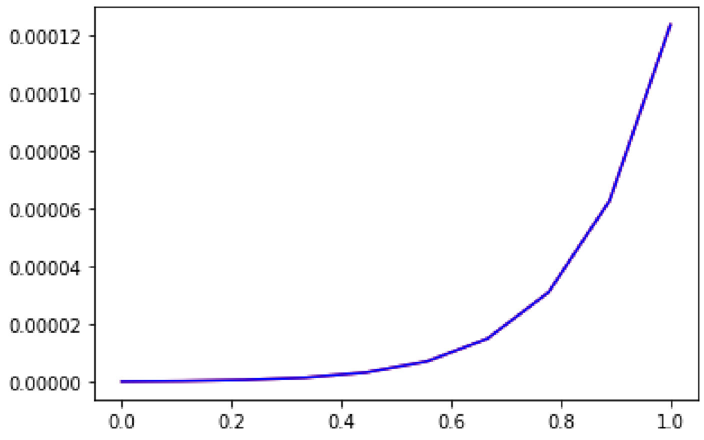


solution where v does not cancel out and an arbirary v value is used --> same result

In [64]:
```
import numpy as np
import matplotlib.pyplot as plt


# create an array for t starting at t = 0 and ending at t = 1
# divide this interval into 10 equal parts
t = np.linspace(0, 1, 10)
dt = 0.1

# create an array for x starting at x = 0 and ending at x = 1
# divide this interval into 10 equal parts
x = np.linspace(0, 1, 10)
dx = 0.1

# set arbitrary velocity value
v = 1

# print the values
print ("dt = ", dt, ", dx = ", dx)

# build array U to store all approximations at all times and
# at all spatial points
u = np.zeros ((len(t), len(x)))

# enforce the left boundary condition
u[:, 0] = 0

# enforce the right boundary condition
u[:, -1] = 0

# enforce the initial condition
## when t = 0
u[0,:] = np.exp(-(x-4)**2)

# fill the array U one row at a time, leave the boundary conditions
# fixed and fill indices 1 through -2
for n in range (len(t) - 1):
    #u[n+1,1:-1] = u[n,1:-1] - a*(u[n, 2:] - u[n,1:-1])
    u[n+1,1:-1] = u[n,1:-1] - (v*dt/dx) * (u[n, 2:]- u[n, 1:-1])

y = np.exp(-(x-4)**2)

# plot the solution
plt.plot(x, u[0], color = "red", label = "u solution" )
plt.plot(x, y, color = "blue", label = "analytical solution")
```

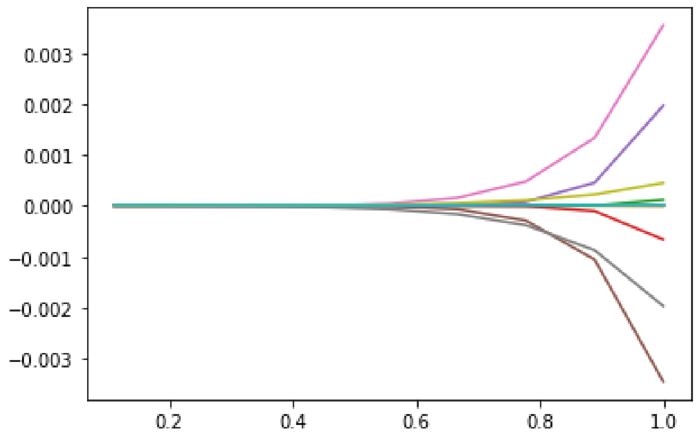
```
dt =  0.1 , dx =  0.1
```

Out[64]:
```
[<matplotlib.lines.Line2D at 0x11acf7f10>]
```



In [65]:
```
plt.plot(x[1:], u[1:], label = "alt. u solutions")
```

Out[65]:
```
[<matplotlib.lines.Line2D at 0x11abf94e0>,
 <matplotlib.lines.Line2D at 0x11abf9540>,
 <matplotlib.lines.Line2D at 0x11abf9660>,
 <matplotlib.lines.Line2D at 0x11abf9780>,
 <matplotlib.lines.Line2D at 0x11abf98a0>,
 <matplotlib.lines.Line2D at 0x11abf99c0>,
 <matplotlib.lines.Line2D at 0x11abf9ae0>,
 <matplotlib.lines.Line2D at 0x11abf9c00>,
 <matplotlib.lines.Line2D at 0x11abf9d20>,
 <matplotlib.lines.Line2D at 0x11abf9e40>]
```



The first solution from the finite difference solution yields the same output as the analytical solution. The graphs are nearly identical. For the remaining outputs from the finite difference solution, the graph show varying exponential plots. Unlike the heat and wave functions, this method cannot be stabilized by changing Δx and Δt because these values are essentially cancelled out by v. We can see this since $v = \frac{\Delta x}{\Delta t}$ such that $\frac{v \Delta t}{\Delta x}$ = 1.

Exercise 6.91 You may recall from your differential equations class that population growth under limited resources is goverened by the logistic equation $x' = k_1 x(1 - x/k_2)$ where $x = x(t)$ is the population, k_1 is the intrinsic growth rate of the population, and k_2 is the carrying capacity of the population. The carrying capacity is the maximum population that can be supported by the environment. The trouble with this model is that the species is presumed to be fixed to a spatial location. Let's make a modification to this model that allows the species to spread out over time while they reproduce. We have seen throughout this chapter that the heat equation $u_t = D(u_{xx} + u_{yy})$ models the diffusion of a substance (like heat or concentration). We therefore propose the model

$$\frac{\partial u}{\partial t} = k_1 u \left(1 - \frac{u}{k_2}\right) + D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

where $u(t, x, y)$ is the population density of the species at time t and spatial point (x, y), (x, y) is a point in some square spatial domain, k_1 is the growth rate of the population, k_2 is the carrying capacity of the population, and D is the rate of diffusion. Develop a finite difference scheme to solve this PDE. Experiment with this model showing the interplay between the parameters D, k_1, and k_2. Take an initial condition of

$$u(0, x, y) = e^{-((x-0.5)^2 + (y-0.5)^2)/0.05}.$$

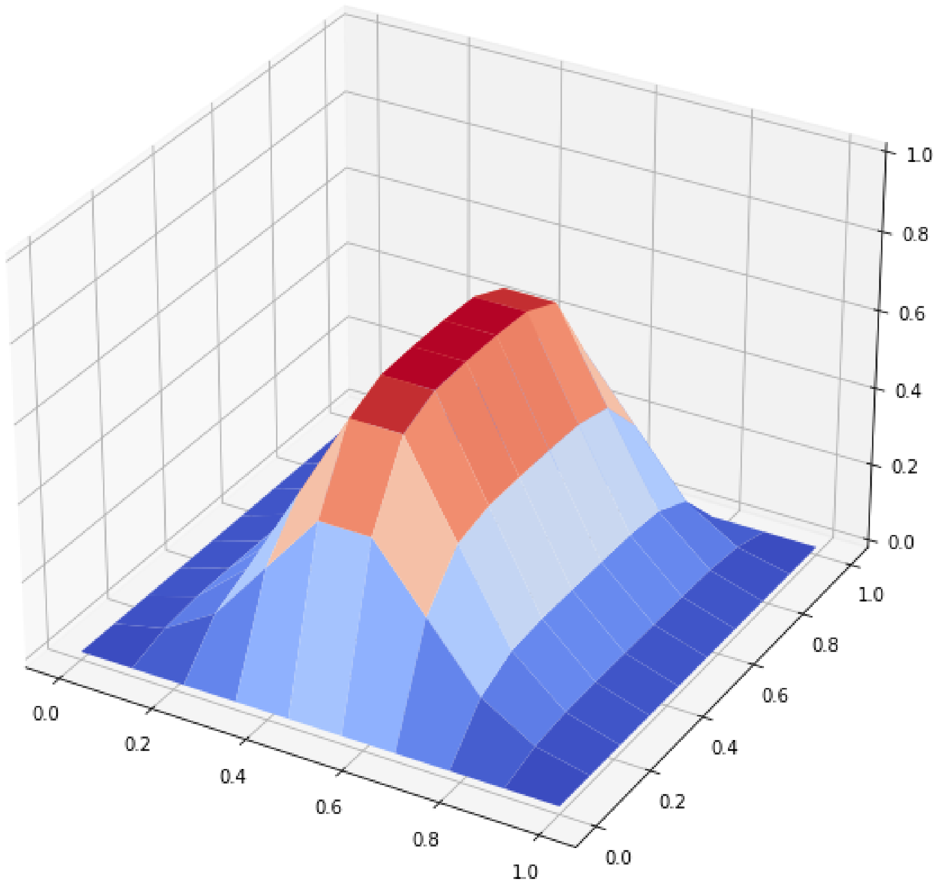
$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t}$ = $k_1 u_{i,j}^{n}(1 - \frac{u_{i,j}^{n}}{k_2}$) + D($\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}$)
$u_{i,j}^{n+1}$ = $(k_1 u_{i,j}^{n}(1 - \frac{u_{i,j}^{n}}{k_2}$) + D($\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{\Delta y^2}$)) $\Delta\, t + u_{i,j}^{n}$

In [66]:
```
def wrapper (x, y, u, t, X, Y):
    def plotter(Frame):
        fig = plt.figure(figsize=(12,10))
        ax = fig.gca(projection='3d')
        ax.plot_surface(X,Y,u[Frame,:,:], cmap=cm.coolwarm)
        ax.set_zlim(0,1)
        plt.show()
    interactive_plot = interactive(plotter, Frame=(0,len(t)))
    display (interactive_plot)

def main():
    # create an array for t starting at t = 0 and ending at t = 1
    # divide this interval into 10 equal parts
    t = np.linspace(0,1,10)
    dt = 1e-3

    # create an array for x and y starting at x = 0 and ending at x = 1
    # divide this interval into 10 equal parts
    x = np.linspace(0,1,10)
    y = np.linspace(0,1,10)

    dx = 0.1
    dy = 0.1

    X, Y = np.meshgrid(x,y)

    # define the wave velocity
    D = 1

    k1 = 5
    k2 = 10

    # print the values
    print ("dt = ", dt, ", dx = ", dx)

    # build array U to store all approximations at all times and
    # at all spatial points
    u = np.zeros ((len(t), len(x), len(y)))

    # enforce the left boundary condition
    u[:,:, 0] = 0 # y initial
    u[:,0, :] = 0 # x initial

    # enforce the right boundary condition
    u[:, :, -1] = 0 # y initial
    u[:, -1, :] = 0 # x initial


    # enforce the initial condition
    u[0,:, :] = np.exp(-((x-0.5)**2 + (y-0.5)**2)/0.05)

    # fill the array U one row at a time, leave the boundary conditions
    # fixed and fill indices 1 through -2
    for n in range (len(t) - 1):
            u[n + 1, 1:-1, 1:-1] = (k1*u[n, 1:-1, 1:-1]*(1-(u[n, 1:-1, 1:-1]/k2)) + \
            D*((u[n, 2:, 1:-1] - 2*u[n, 1:-1, 1:-1] + u[n, :-2, 1:-1]) / dx**2) + \
            ((u[n, 1:-1, 2:] - 2*u[n, 1:-1, 1:-1] + u[n, 1:-1, :-2])/dy**2))*dt + \
            u[n, 1:-1, 1:-1]
    # plot the solution
    wrapper (x, y, u, t, X, Y)

main()
```

```
dt =  0.001 , dx =  0.1
```

Frame 6

```
/var/folders/sx/gw6n5mnj28x21kcmksnp_p240000gn/T/ipykernel_55877/292827910.py:4:
MatplotlibDeprecationWarning: Calling gca() with keyword arguments was deprecated
in Matplotlib 3.4. Starting two minor releases later, gca() will take no keyword
arguments. The gca() function should only be used to get the current axes, or if
no axes exist, create new axes with default keyword arguments. To create a new ax
es with non-default arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```

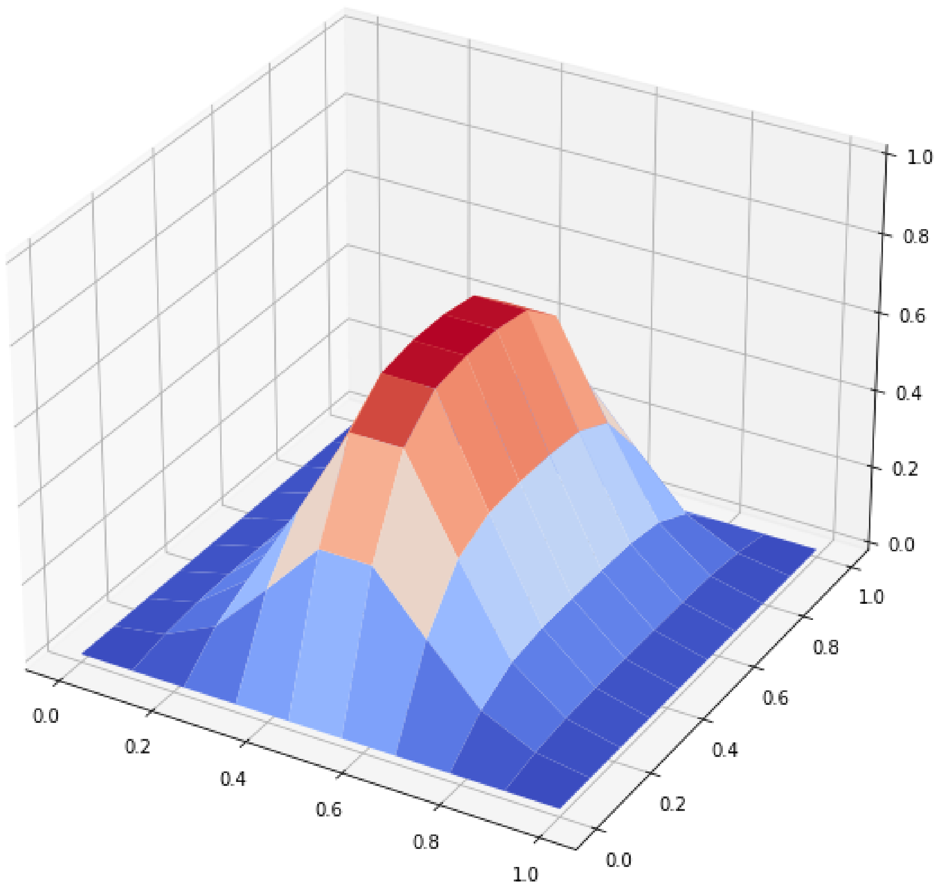


In [67]:
```
def wrapper (x, y, u, t, X, Y):
    def plotter(Frame):
        fig = plt.figure(figsize=(12,10))
        ax = fig.gca(projection='3d')
        ax.plot_surface(X,Y,u[Frame,:,:], cmap=cm.coolwarm)
        ax.set_zlim(0,1)
        plt.show()
    interactive_plot = interactive(plotter, Frame=(0,len(t)))
    display (interactive_plot)

def main():
    # create an array for t starting at t = 0 and ending at t = 1
    # divide this interval into 10 equal parts
    t = np.linspace(0,1,10)
    dt = 1e-3

    # create an array for x and y starting at x = 0 and ending at x = 1
    # divide this interval into 10 equal parts
    x = np.linspace(0,1,10)
    y = np.linspace(0,1,10)

    dx = 0.1
    dy = 0.1

    X, Y = np.meshgrid(x,y)

    # define the wave velocity
    D = 2

    k1 = 4
    k2 = 8

    # print the values
    print ("dt = ", dt, ", dx = ", dx)

    # build array U to store all approximations at all times and
    # at all spatial points
    u = np.zeros ((len(t), len(x), len(y)))

    # enforce the left boundary condition
    u[:,:, 0] = 0 # y initial
    u[:,0, :] = 0 # x initial

    # enforce the right boundary condition
    u[:, :, -1] = 0 # y initial
    u[:, -1, :] = 0 # x initial


    # enforce the initial condition
    u[0,:, :] = np.exp(-((x-0.5)**2 + (y-0.5)**2)/0.05)

    # fill the array U one row at a time, leave the boundary conditions
    # fixed and fill indices 1 through -2
    for n in range (len(t) - 1):
            u[n + 1, 1:-1, 1:-1] = (k1*u[n, 1:-1, 1:-1]*(1-(u[n, 1:-1, 1:-1]/k2)) + \
            D*((u[n, 2:, 1:-1] - 2*u[n, 1:-1, 1:-1] + u[n, :-2, 1:-1]) / dx**2) + \
            ((u[n, 1:-1, 2:] - 2*u[n, 1:-1, 1:-1] + u[n, 1:-1, :-2])/dy**2))*dt + \
            u[n, 1:-1, 1:-1]
    # plot the solution
    wrapper (x, y, u, t, X, Y)

main()
```

```
dt =  0.001 , dx =  0.1
```

Frame 5

```
/var/folders/sx/gw6n5mnj28x21kcmksnp_p240000gn/T/ipykernel_55877/1408654499.py:4:
MatplotlibDeprecationWarning: Calling gca() with keyword arguments was deprecated
in Matplotlib 3.4. Starting two minor releases later, gca() will take no keyword
arguments. The gca() function should only be used to get the current axes, or if
no axes exist, create new axes with default keyword arguments. To create a new ax
es with non-default arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```

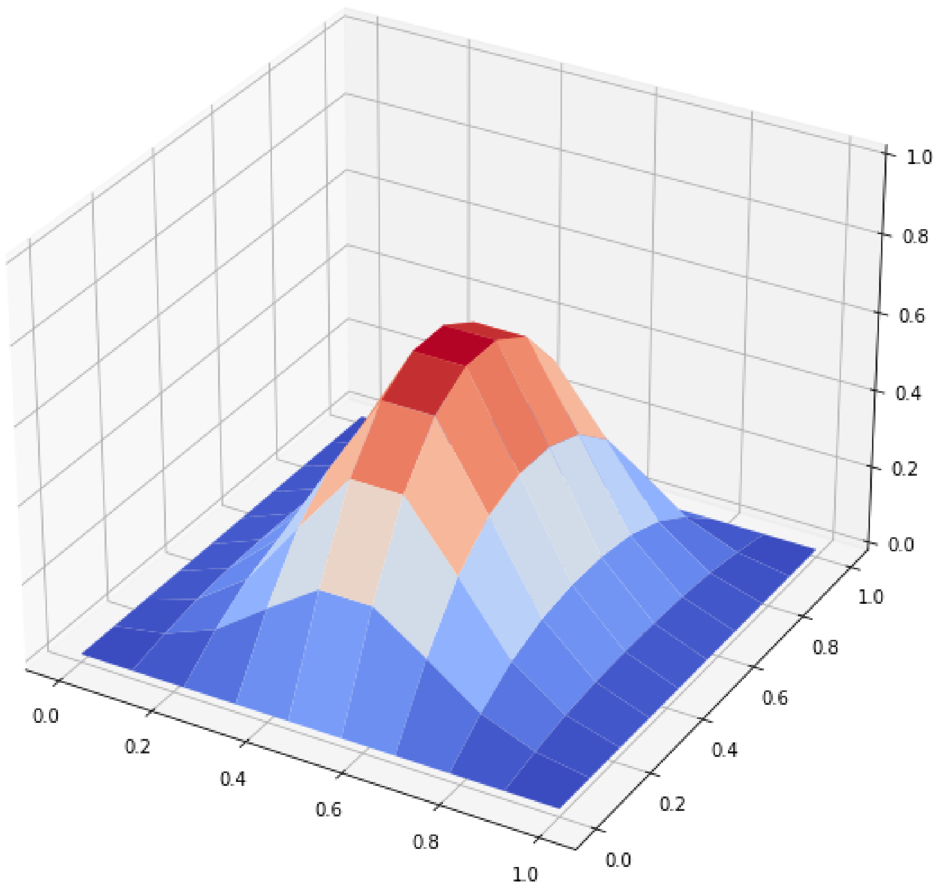


In [68]:
```
def wrapper (x, y, u, t, X, Y):
    def plotter(Frame):
        fig = plt.figure(figsize=(12,10))
        ax = fig.gca(projection='3d')
        ax.plot_surface(X,Y,u[Frame,:,:], cmap=cm.coolwarm)
        ax.set_zlim(0,1)
        plt.show()
    interactive_plot = interactive(plotter, Frame=(0,len(t)))
    display (interactive_plot)

def main():
    # create an array for t starting at t = 0 and ending at t = 1
    # divide this interval into 10 equal parts
    t = np.linspace(0,1,10)
    dt = 1e-3

    # create an array for x and y starting at x = 0 and ending at x = 1
    # divide this interval into 10 equal parts
    x = np.linspace(0,1,10)
    y = np.linspace(0,1,10)

    dx = 0.1
    dy = 0.1

    X, Y = np.meshgrid(x,y)

    # define the wave velocity
    D = 3

    k1 = 15
    k2 = 30

    # print the values
    print ("dt = ", dt, ", dx = ", dx)

    # build array U to store all approximations at all times and
    # at all spatial points
    u = np.zeros ((len(t), len(x), len(y)))

    # enforce the left boundary condition
    u[:,:, 0] = 0 # y initial
    u[:,0, :] = 0 # x initial

    # enforce the right boundary condition
    u[:, :, -1] = 0 # y initial
    u[:, -1, :] = 0 # x initial


    # enforce the initial condition
    u[0,:, :] = np.exp(-((x-0.5)**2 + (y-0.5)**2)/0.05)

    # fill the array U one row at a time, leave the boundary conditions
    # fixed and fill indices 1 through -2
    for n in range (len(t) - 1):
            u[n + 1, 1:-1, 1:-1] = (k1*u[n, 1:-1, 1:-1]*(1-(u[n, 1:-1, 1:-1]/k2)) + \
            D*((u[n, 2:, 1:-1] - 2*u[n, 1:-1, 1:-1] + u[n, :-2, 1:-1]) / dx**2) + \
            ((u[n, 1:-1, 2:] - 2*u[n, 1:-1, 1:-1] + u[n, 1:-1, :-2])/dy**2))*dt + \
            u[n, 1:-1, 1:-1]
    # plot the solution
    wrapper (x, y, u, t, X, Y)

main()
```

```
dt =  0.001 , dx =  0.1
```

Frame 7

```
/var/folders/sx/gw6n5mnj28x21kcmksnp_p240000gn/T/ipykernel_55877/1353815954.py:4:
MatplotlibDeprecationWarning: Calling gca() with keyword arguments was deprecated
in Matplotlib 3.4. Starting two minor releases later, gca() will take no keyword
arguments. The gca() function should only be used to get the current axes, or if
no axes exist, create new axes with default keyword arguments. To create a new ax
es with non-default arguments, use plt.axes() or plt.subplot().
  ax = fig.gca(projection='3d')
```



Exercise 6.92 In Exercise 6.72 you solved the Poisson equation, $u_{xx} + u_{yy} = f(x, y)$, on the unit square with homogenous Dirichlet boundary conditions and a forcing function $f(x, y) = -20\exp\left(-\frac{(x-0.5)^2+(y-0.5)^2}{0.05}\right)$. Use a 10×10 grid of points to solve the Poisson equation on the same domain with the same forcing function but with boundary conditions

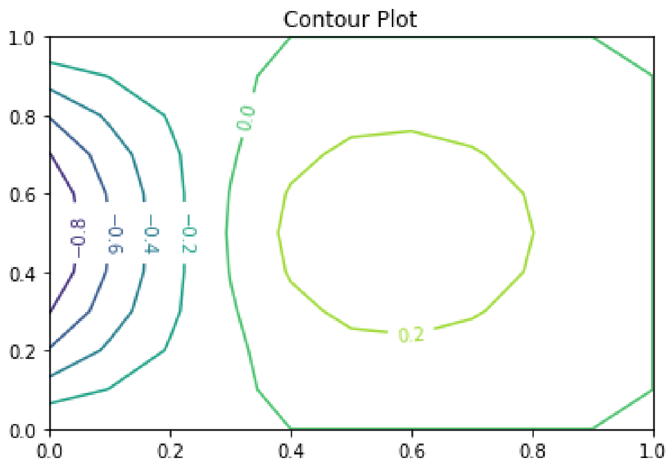
$$u(0, y) = 0, \quad u(1, y) = 0, \quad u(x, 0) = -\sin(\pi x), \quad u(x, 1) = 0.$$

Show a contour plot of your solution.

In [69]:
```
# number of square on a side
M = 10

# tolerance
tol = 1e-3

# establish arrays
phi = np.zeros([M+1, M+1], float)
phi_prime = np.empty([M+1, M+1], float)

# initialize boundaries
phi[0, :] = 0
phi[1, :] = 0
phi[:, 1] = 0

h = 0.1 #domain/M
delta = 1.0

# initial array to store function
f = np.zeros([M+1, M+1], float)

# function loop
for i in range(M+1):
    for j in range(M+1):
        f[i, j] = -20 * np.exp(-((h * i - 0.5)**2 + (h * j - 0.5)**2)/0.05)

# main loop
while(delta > tol):
    for i in range(M+1):
        for j in range(M+1):
            if (i == 0) or (i == M):
                phi_prime[i, j] = 0
            elif(j == 0):
                phi_prime[i, j] = -np.sin(np.pi * i * h)
            elif (j == M):
                phi_prime[i, j] = 0
            else:
                phi_prime[i, j] = 0.25 * (phi[i+1, j] + phi[i-1, j] + phi[i, j+1] + p

    diff = abs(phi-phi_prime)
    delta = np.amax(diff)

    phi, phi_prime = phi_prime, phi
```

In [70]:
```
import numpy as np
import matplotlib.cm as cm
import matplotlib.pyplot as plt

x = np.arange(0, 1.1, 0.1)
y = np.arange(0, 1.1, 0.1)
X, Y = np.meshgrid(x, y)
Z = phi

fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z)
ax.clabel(CS, inline=True, fontsize=10)
ax.set_title('Contour Plot')
```

Out[70]:
```
Text(0.5, 1.0, 'Contour Plot')
```


Contour Plot