PHYS-E0412 Computational Physics :: Homework 11

Ari Viitala 432568

In [473]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

a) Heat equation

Let's first make the time and space variables

In [474]:
```
#spatial grid
N = 100
h = 1 / (N - 1)
x = np.linspace(0,1, N)

#initial state
u0 = 1.831 * np.exp(-10 * (x - 0.5)**2)
u0 -= u0[0]

#time
t_steps = 100
time = np.linspace(0, 0.1, 100)[1:]
t_h = time[1] - time[0]
```

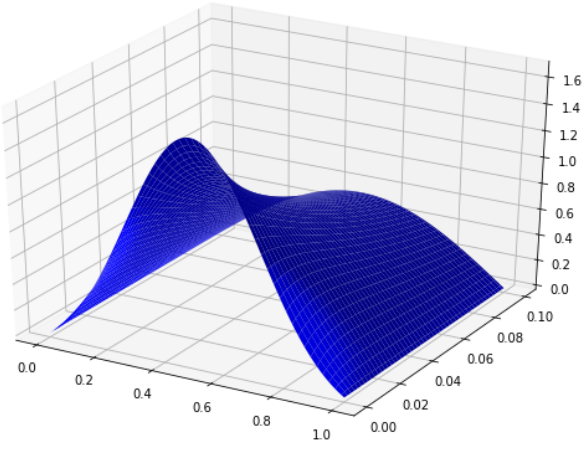
In [475]:
```
#the three point stencil
stencil =  -1 / h**2 * (2 * np.eye(N) - np.diag(np.ones(N-1), 1) - np.diag(np.ones(N-1), -1))

#boundary conditions
stencil[0] = 0
stencil[-1] = 0
stencil[0, 0] = 1
stencil[-1, -1] = 1
```

Implicit Euler

In [476]:
```
#vector to store the values
u = [u0]
#for each timestep
for i, t in enumerate(time):
    #solve the system for the next state
    u1 = np.linalg.solve(np.eye(N) - t_h * stencil, u[i])
    #append to the result vecotr
    u.append(u1)

#convert to numpy array
u= np.array(u)
```

Plotting the surface $u \times t$.

In [478]:
```
fig = plt.figure(1, (10, 7))
ax = fig.add_subplot(111, projection='3d')

time_ax = [0] + list(time)
X, Y = np.meshgrid(x, time_ax) # Plot the surface
ax.plot_surface(X, Y, u, color='b')
plt.show()
```

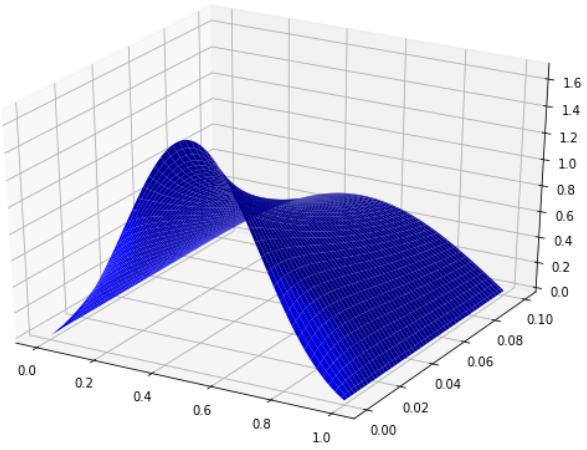


We see that implicit Euler gives us a pretty believeable solution. The heat distribution begins to shooth out as time goes on.

Crank-Nicolson

In [479]:
```
#result vector
u_cn = [u0]
#the implicit side
first = np.linalg.solve(np.eye(N) - 0.5 * t_h * stencil, np.eye(N))
#the explicit side
second = np.eye(N) + 0.5 * t_h * stencil
m = first @ second
for i, t in enumerate(time):
    #the next step is just matrix multiplication
    u1 = m.dot(u_cn[i])
    u_cn.append(u1)

u_cn = np.array(u_cn)
```

In [480]:
```
fig = plt.figure(1, (10, 7))
ax = fig.add_subplot(111, projection='3d')

time_ax = [0] + list(time)
X, Y = np.meshgrid(x, time_ax) # Plot the surface
ax.plot_surface(X, Y, u_cn, color='b')
plt.show()
```



We see that Crank-Nicolson gives us a really similar answer to the implicit Euler.

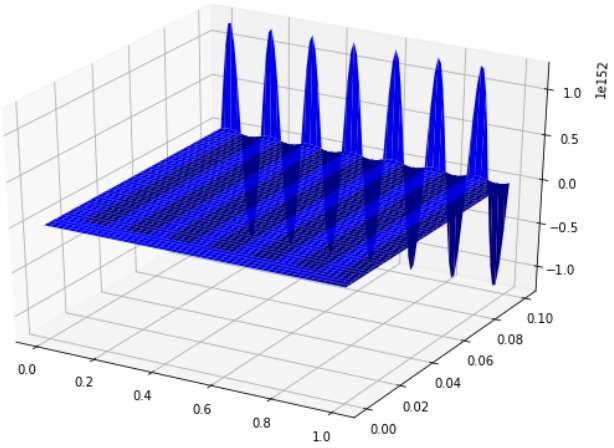
Reversing time

The time is reversed just by switching the sign of the time step so we are stepping back in time. Otherwise the methods are the same.

Crank-Nicolson

In [481]:
```
u_cn_r = [u_cn[-1]]
first = np.linalg.solve(np.eye(N) + 0.5 * t_h * stencil, np.eye(N))
second = np.eye(N) - 0.5 * t_h * stencil
m = first @ second
for i, t in enumerate(time):
    u1 = m.dot(u_cn_r[i])
    u_cn_r.append(u1)

u_cn_r = np.array(u_cn_r)
```

In [482]:
```
fig = plt.figure(1, (10, 7))
ax = fig.add_subplot(111, projection='3d')

time_ax = [0] + list(time)
X, Y = np.meshgrid(x, time_ax) # Plot the surface
ax.plot_surface(X, Y, u_cn_r, color='b')
plt.show()
```

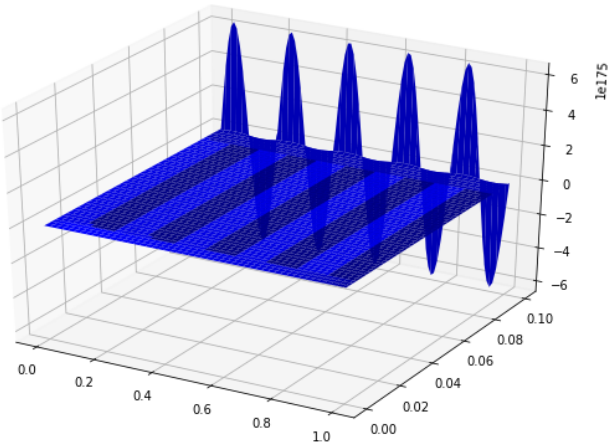


Implicit Euler

In [483]:
```
u_r = [u[-1]]
for i, t in enumerate(time):
    u1 = np.linalg.solve(np.eye(N) + t_h * stencil, u_r[i])
    u_r.append(u1)

u_r= np.array(u_r)
```

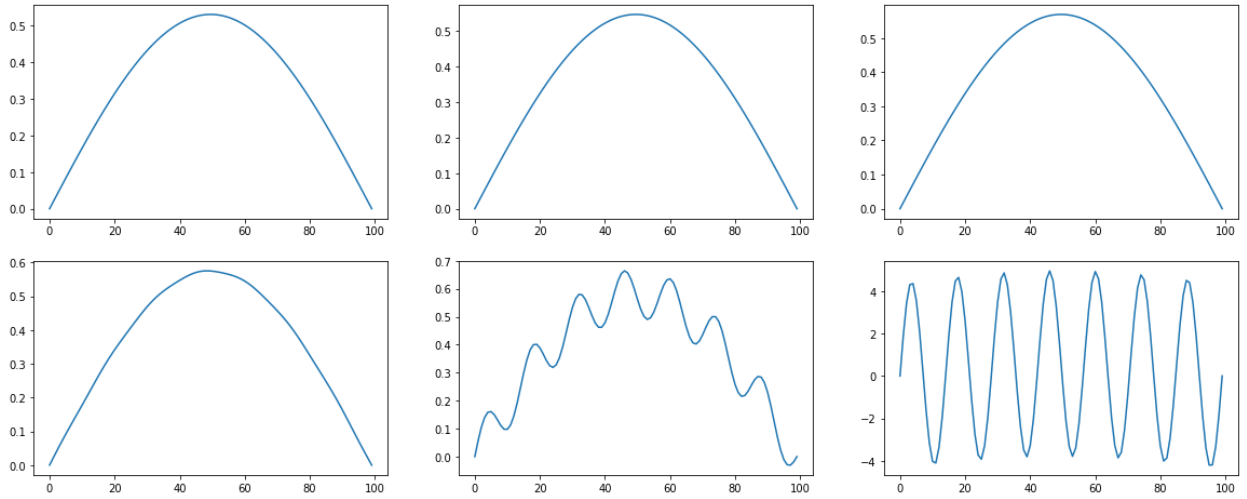
In [484]:
```
fig = plt.figure(1, (10, 7))
ax = fig.add_subplot(111, projection='3d')

time_ax = [0] + list(time)
X, Y = np.meshgrid(x, time_ax) # Plot the surface
ax.plot_surface(X, Y, u_r, color='b')
plt.show()
```

```
/home/ari/anaconda3/lib/python3.6/site-packages/mpl_toolkits/mplot3d/proj3d.py:73: RuntimeWarning: overflow encountered
in double_scalars
  return np.sqrt(v[0]**2+v[1]**2+v[2]**2)
```



Plotting the first steps of the Crank-Nicolson method.

In [485]:
```
plt.figure(1, (20, 8))
plt.subplot(2, 3, 1)
plt.plot(u_cn_r[0])
plt.subplot(2, 3, 2)
plt.plot(u_cn_r[3])
plt.subplot(2, 3, 3)
plt.plot(u_cn_r[7])
plt.subplot(2, 3, 4)
plt.plot(u_cn_r[8])
plt.subplot(2, 3, 5)
plt.plot(u_cn_r[9])
plt.subplot(2, 3, 6)
plt.plot(u_cn_r[10])
plt.show()
```

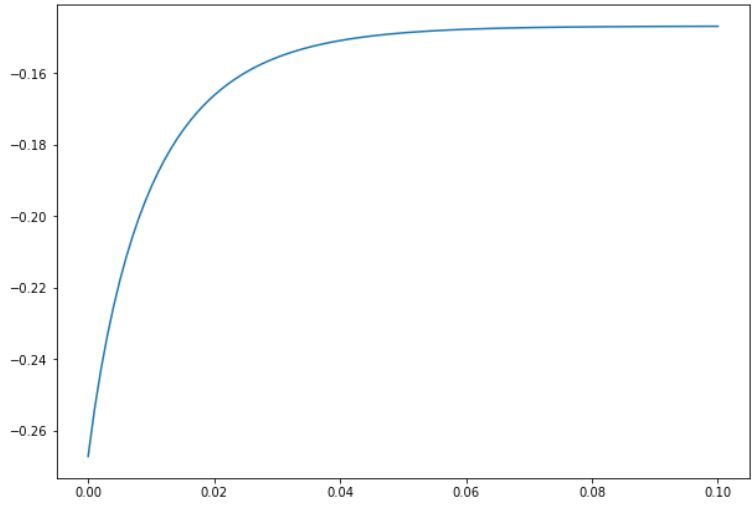


We see that the method goes correctly for about 4 time-steps before it starts to oscillate out of control. Based on the countour plots this is the case for both of the methods and we are unable to retrieve u_0. This is probably because the heat equation has on tendency to smooth out the function and if we reverse that effect we get big magnification for errors in the solution that accumulate as time goes on and make the solution explode.

Entropy

In [486]:
```
S = []

for i in range(t_steps):
    #calculating the normalized u for this step
    u_norm = (u[i] / np.trapz(u[i], x))[1:-1]
    #calculating the entropy
    S.append(-np.trapz(u_norm * np.log(u_norm), x[1:-1]))
```

In [487]:
```
plt.figure(1, (10, 7))
p_time = [0] + list(time)
plt.plot(np.linspace(0,0.1, t_steps), S)
plt.show()
```



As we could expect from the heat equation the entropy increases in the system as the time goes on.

b) Wave equation

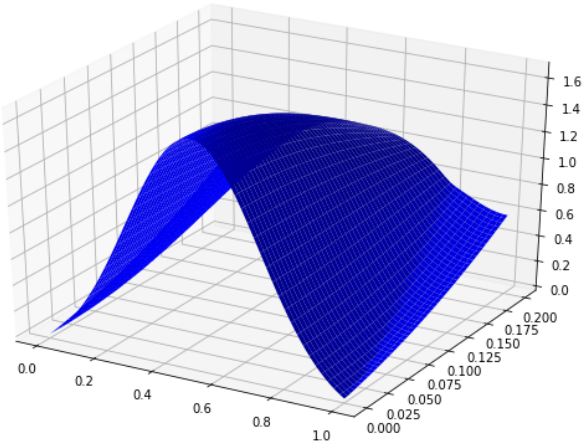
Time and space variabes for the system and the matrix used to solve the system

In [488]:
```
#space grid
N = 200
h = 1 / (N - 1)
x = np.linspace(0,1, N)

#initial state
u0 = 1.831 * np.exp(-10 * (x - 0.5)**2)
u0 -= u0[0]

#initial velocity
v0 = np.zeros_like(x)

#time
t_steps = 200
tmax = 0.2
time = np.linspace(0, tmax, t_steps)
t_h = time[1] - time[0]

#the stencil is the same except that the boundary conditions are continuous
stencil =   (2 * np.eye(N) - np.diag(np.ones(N-1), 1) - np.diag(np.ones(N-1), -1))

#boundary conditions
stencil[0, -1] = -1
stencil[-1, 0] = -1

stencil *= -1 / h**2

#creating the matrix from zero matrices, identity matrix and the stencil
right = np.vstack((np.eye(N), np.zeros_like(stencil)))
left = np.vstack((np.zeros_like(stencil), stencil))
wave_sten = np.hstack((left, right))
```

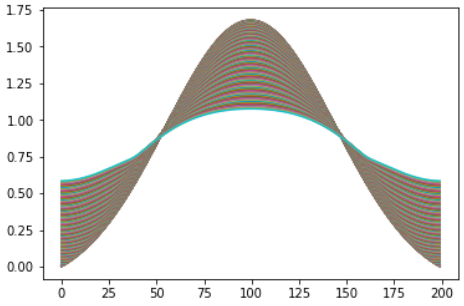
Implicit Euler

In [489]:
```
#initial value
w_u = [np.hstack((u0, v0))]
for i in range(t_steps - 1):
    #solving the system
    u1 = np.linalg.solve(np.eye(N*2) - t_h * wave_sten, w_u[i])
    w_u.append(u1)
w_u = np.array(w_u)[:,:N]
```

In [490]:
```
fig = plt.figure(1, (10, 7))
ax = fig.add_subplot(111, projection='3d')

X, Y = np.meshgrid(x, time) # Plot the surface
ax.plot_surface(X, Y, w_u, color='b')
plt.show()
```

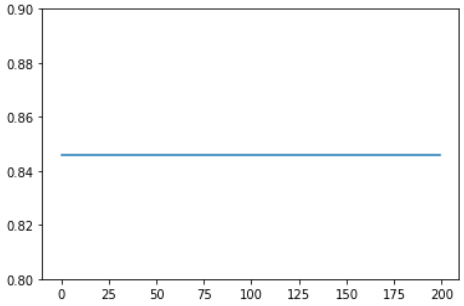


In [491]:
```
for i in range(t_steps):
    plt.plot(w_u[i])
```



Plotting the mean of the reasult vector for each step. This should be constant for each step.

In [492]:
```
plt.plot(np.mean(w_u, axis = 1))
plt.ylim(0.8, 0.9)
plt.show()
```

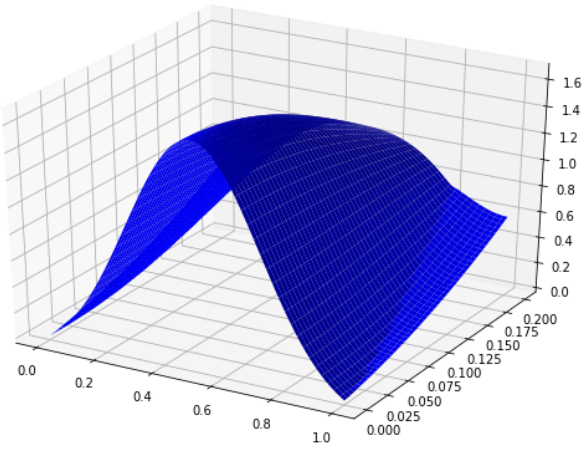


We see that the result seems like the wave equation.

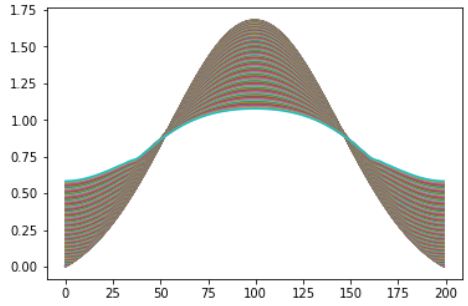
Crank-Nicolson

In [493]:
```
w_u_cn = [np.hstack((u0, v0))]
first = np.linalg.solve(np.eye(2*N) - 0.5 * t_h * wave_sten, np.eye(2*N))
second = np.eye(2*N) + 0.5 * t_h * wave_sten
m = first @ second
for i in range(t_steps - 1):
    w_u1 = m.dot(w_u_cn[i])
    w_u_cn.append(w_u1)

w_u_cn = np.array(w_u_cn)[:,:N]
```

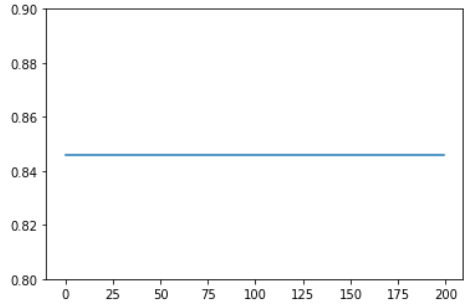
In [494]:
```
fig = plt.figure(1, (10, 7))
ax = fig.add_subplot(111, projection='3d')


X, Y = np.meshgrid(x, time) # Plot the surface
ax.plot_surface(X, Y, w_u_cn, color='b')
plt.show()
```



In [495]:
```
for i in range(t_steps):
    plt.plot(w_u_cn[i])
```



In [496]:
```
plt.plot(np.mean(w_u_cn, axis = 1))
plt.ylim(0.8, 0.9)
plt.show()
```



Again the solution seems correct.

Reversing time

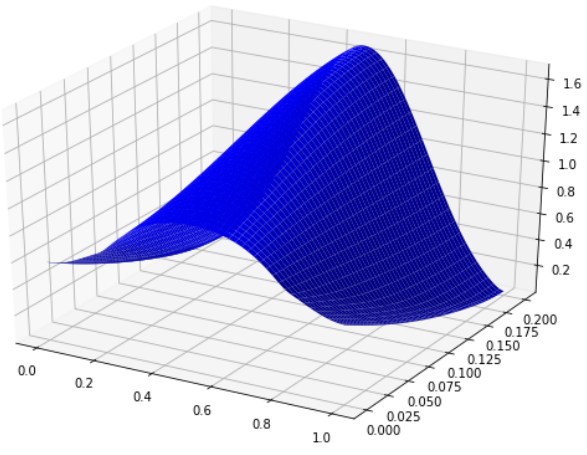
Let's first solve the forwards in time system first. The solutions for both methods seemd the same so here we test just the implicit Euler.

In [497]:
```
#forward system
w_u = [np.hstack((u0, v0))]
for i in range(t_steps - 1):
    u1 = np.linalg.solve(np.eye(N*2) - t_h * wave_sten, w_u[i])
    w_u.append(u1)
w_u = np.array(w_u)
```

In [498]:
```
#the initial value is the last state of the forward system
r_w_u = [w_u[-1]]
for i in range(t_steps - 1):
    #solving the system but the sign of t_h is changed from the forward solution
    u1 = np.linalg.solve(np.eye(N*2) + t_h * wave_sten, r_w_u[i])
    r_w_u.append(u1)
r_w_u = np.array(r_w_u)[:,:N]
```

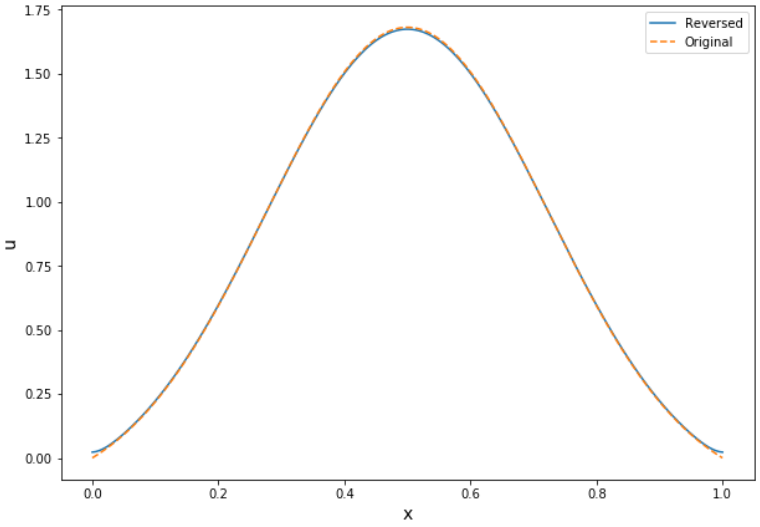
Plotting

In [499]:
```
fig = plt.figure(1, (10, 7))
ax = fig.add_subplot(111, projection='3d')

X, Y = np.meshgrid(x, time) # Plot the surface
ax.plot_surface(X, Y, r_w_u, color='b')
plt.show()
```



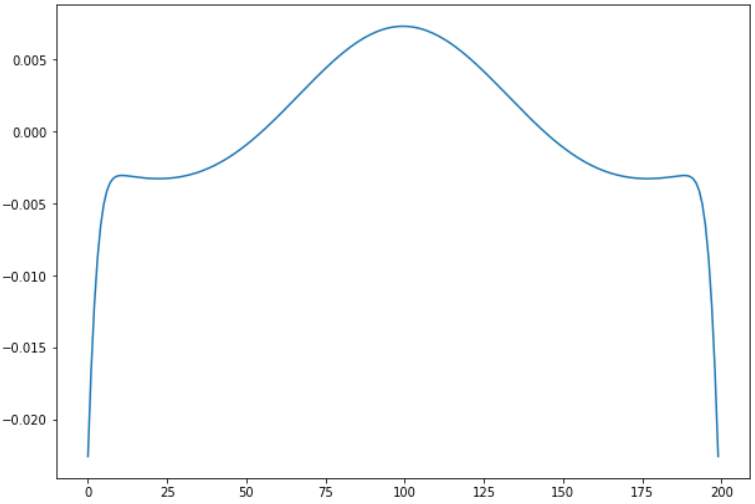
We seet that the solution goes back to a state that resembles the inial state.

In [500]:
```
plt.figure(1, (10, 7))
plt.plot(x, r_w_u[-1], label = "Reversed")
plt.plot(x, u0, label = "Original", linestyle = "--")
plt.xlabel("x", size = 14)
plt.ylabel("u", size = 14)
plt.legend()
plt.show()
```



If we plot the initial state of the system and the one we achieve going to $t = 0.2$ and back, we see that we arrive at about the same state that we left with.

In [501]:
```
plt.figure(1, (10, 7))
plt.plot(u0 - r_w_u[-1])
plt.show()
```



If we plot the difference of the two solutions we see that there is the most error on the edges as well as at the peak of the solution but these differences are rather small.

d) I used about 8 hours for this exercise