HW09

April 14, 2021

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

1 Problem 1

We will implement the Crank-Nicolson scheme for solving the Heat Equation which is given as follows:

$$\phi_t = \partial_x[a(x)\cdot\phi_x] + f(x,t) \tag{1}$$
$$\text{IC:}\quad \phi(x,0) = \phi_0(x) \tag{2}$$
$$\text{BC:}\quad \phi(0,t) = 0 \tag{3}$$
$$\text{BC:}\quad \phi(1,t) = 0 \tag{4}$$

In the code below we define this scheme where we assume the homogeneous Dirichlet boundary conditions, but provide flexibility for the initial conditions, variable conductivity, and power sources.

[2]:
```
'''
1 dimensional Heat Equation solver using the Crank-Nicolson scheme.
Assumes homogenous Dirichlet boundary conditions.

f -> f(x,t), power input to the system (i.e. RHS)
a -> a(x), conductivity
ic -> phi(x), initial condition for spatial dimension
d -> ((x0, xN), (t0, tN)) Tuple of spatial domain and temporal domain
hx -> Spatial step size (optional)
ht -> Temporal step size (optional)
'''
def heatEq(f, a, ic, d, hx=0.1, ht=0.1):
    #Unpack some stuff
    Dx = d[0]
    Dt = d[1]

    #Set up grid
    Nx = np.arange(Dx[0], Dx[1]+hx, hx)
    Nt = np.arange(Dt[0], Dt[1]+ht, ht)
    U = np.zeros((len(Nx), len(Nt))) #Space x Time

    #Set IC
    U[:, 0] = ic(Nx)

    #Check consistency
    assert U[0, 0]==0 and U[-1, 0]==0

    #Combine some terms
    h = (1/2)*(ht/hx**2)

    #Build A, B, C, D
    A = h*a(Nx-hx/2)
    B = 1 + h*(a(Nx-hx/2) + a(Nx+hx/2))
    C = h*a(Nx+hx/2)

    #Now we loop and solve
    #Construct E and F
    E, F = np.zeros(len(Nx)), np.zeros(len(Nx))

    for n in range(1, len(Nt)):
        for j in range(1, len(Nx)-1):
            D = A[j]*U[j-1,n-1] - B[j]*U[j,n-1] + \
                                C[j]*U[j+1,n-1] + 2*U[j,n-1] + \
                                ht*(f(Nx[j],Nt[n-1])+f(Nx[j],Nt[n]))/2

            E[j] = C[j]/(B[j]-A[j]*E[j-1])
            F[j] = (D+A[j]*F[j-1])/(B[j]-A[j]*E[j-1])

        #Build solution
        for j in range(len(Nx)-2, 0, -1):
            U[j,n] = E[j]*U[j+1,n] + F[j]

    return U, Nx, Nt
```

In order to visually examine the solutions we receive from our method defined above we will define a method that plots a temporal snapshot for the entire spatial domain.

[3]:
```
def spaceEvolution(U, Nx, Nt, n=4, t='Heat'):
    step = int(U.shape[1]/n)

    fig, ax = plt.subplots(1,1,figsize=(10,10))

    for i in range(n):
        ax.plot(Nx, U[:,step*i])

    ax.set_xlabel('Spatial dimension (x)')
    ax.set_ylabel('Solution (u)')
    ax.set_title(f'Space Evolution of {t} Equation Solution')
    ax.legend([f't = {Nt[step*i]}' for i in range(n)])
```

We will apply our scheme to several examples where we change the system input (f), the conductivity (a), and the initial conditions. First, we define a few terms that will be constant such as the domain of the solution and the stepsizes. Specifically, we consider $x \in [0, 1]$ and $t \in [0, 1]$ with a space step and time step of 0.01 which should be stable as per the theory.

[7]:
```
d = ((0,1), (0,10))
ht = 0.01
hx = 0.01
```

We first consider the following problem..

$$\phi_t = \frac{\phi_{xx}}{100} \tag{5}$$

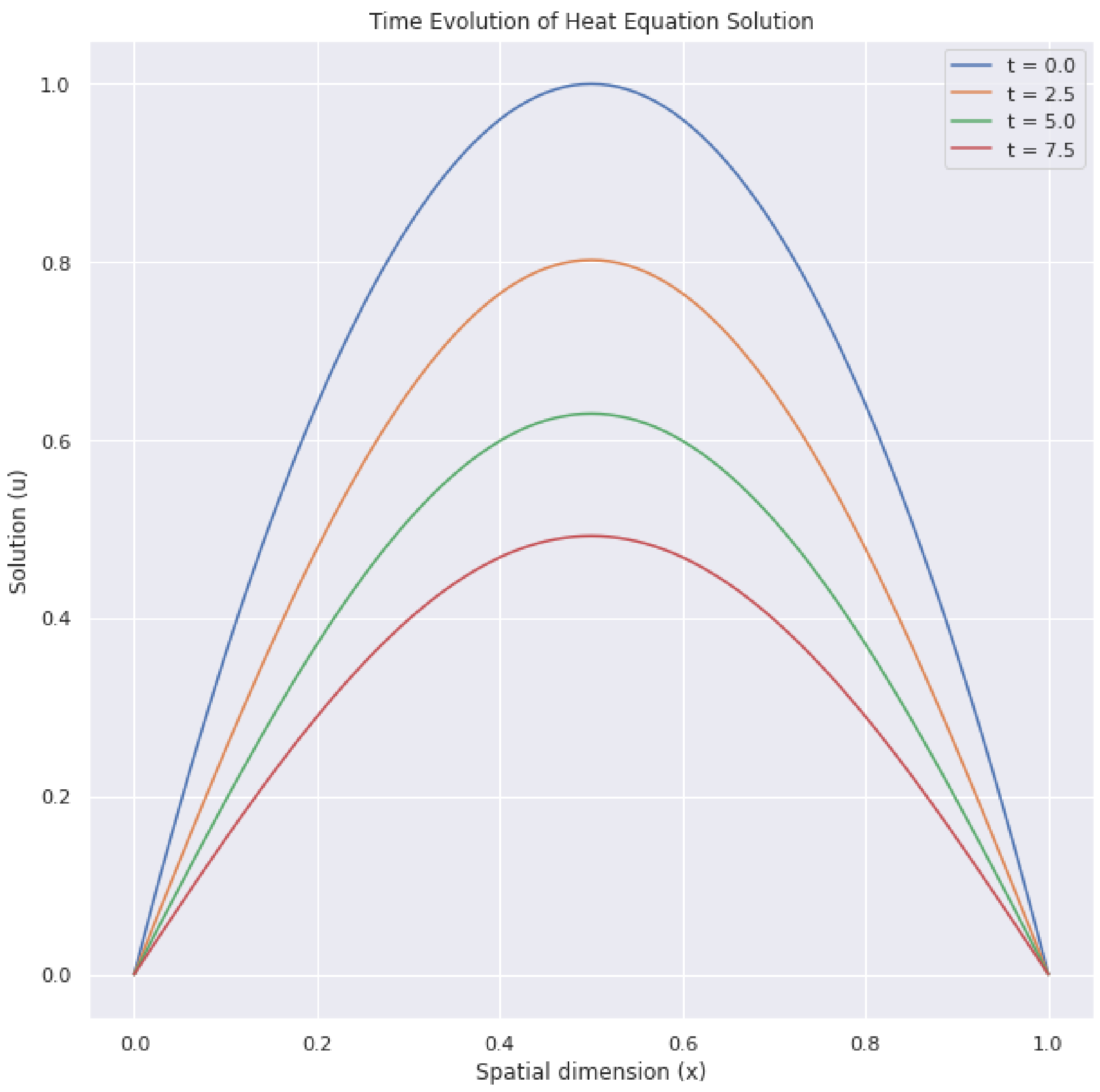
$$\text{IC:} \quad \phi(x, 0) = -4x(x-1) \tag{6}$$

Thus we consider constant conductivity of 0.01 and no system input, so this should be about the simplest form possible. The initial conditions are such that heat is located more towards the center of the domain.

[20]:
```
f = lambda x,t: 0
a = lambda x: 0.01*np.ones(len(x))
ic = lambda x: -4*x*(x-1)
```

[21]:
```
sol, Nx, Nt = heatEq(f, a, ic, d, hx, ht)
```

[22]:
```
spaceEvolution(sol, Nx, Nt)
```



The solution we receive is exactly as we would expect, without any source term the heat decays as time progresses.

Next, we consider the following…

$$\phi_t = \partial_x \Big[\frac{(1-x)}{10} \cdot \phi_x\Big] \tag{7}$$

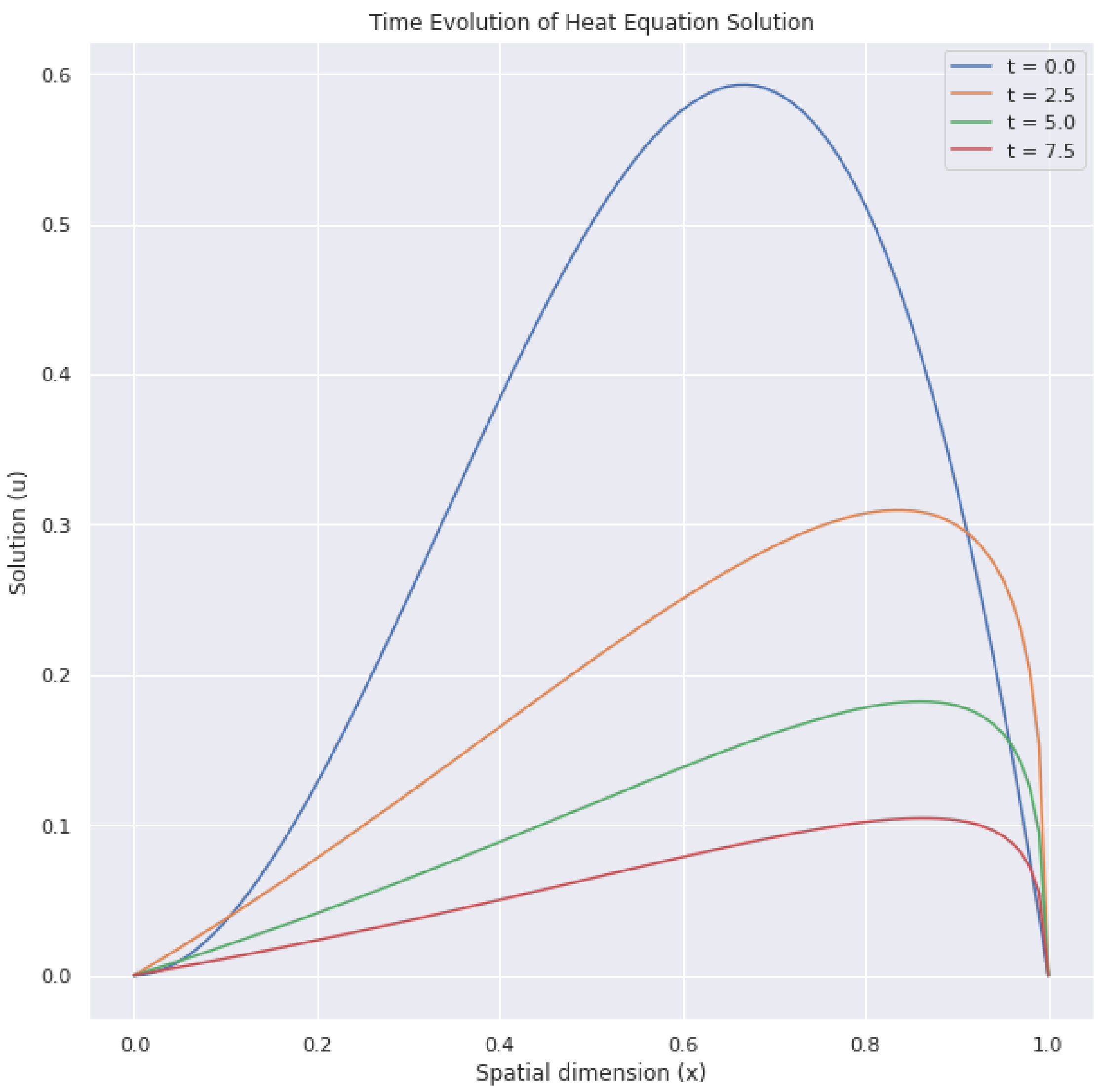
$$\text{IC:} \quad \phi(x,0) = -4x^2(x-1) \tag{8}$$

In this scenario we still do not have a source term, but now the conductivity decreases from 1 towards 0 as you move from left to right on the domain. As well, the initial conditions are such that heat is located more towards this righ side of the domain.

[14]:
```
f = lambda x,t: 0
a = lambda x: 0.1*(1-x)
ic = lambda x: -4*(x**2)*(x-1)
```

[15]:
```
sol, Nx, Nt = heatEq(f, a, ic, d, hx, ht)
```

[16]:
```
spaceEvolution(sol, Nx, Nt)
```



Again we see results that match our expectations. Because the conductivity is zero towards the right side of the domain, heat is unable to escape and has to travel to the left. This is what results in heat continuing to be located near the right end. However, because there is still no source the heat will continue to decay as time progresses.

Lastly, we consider the following...

$$\phi_t = \frac{\phi_{xx}}{100} + \sin(\pi x)e^{-t} \tag{9}$$

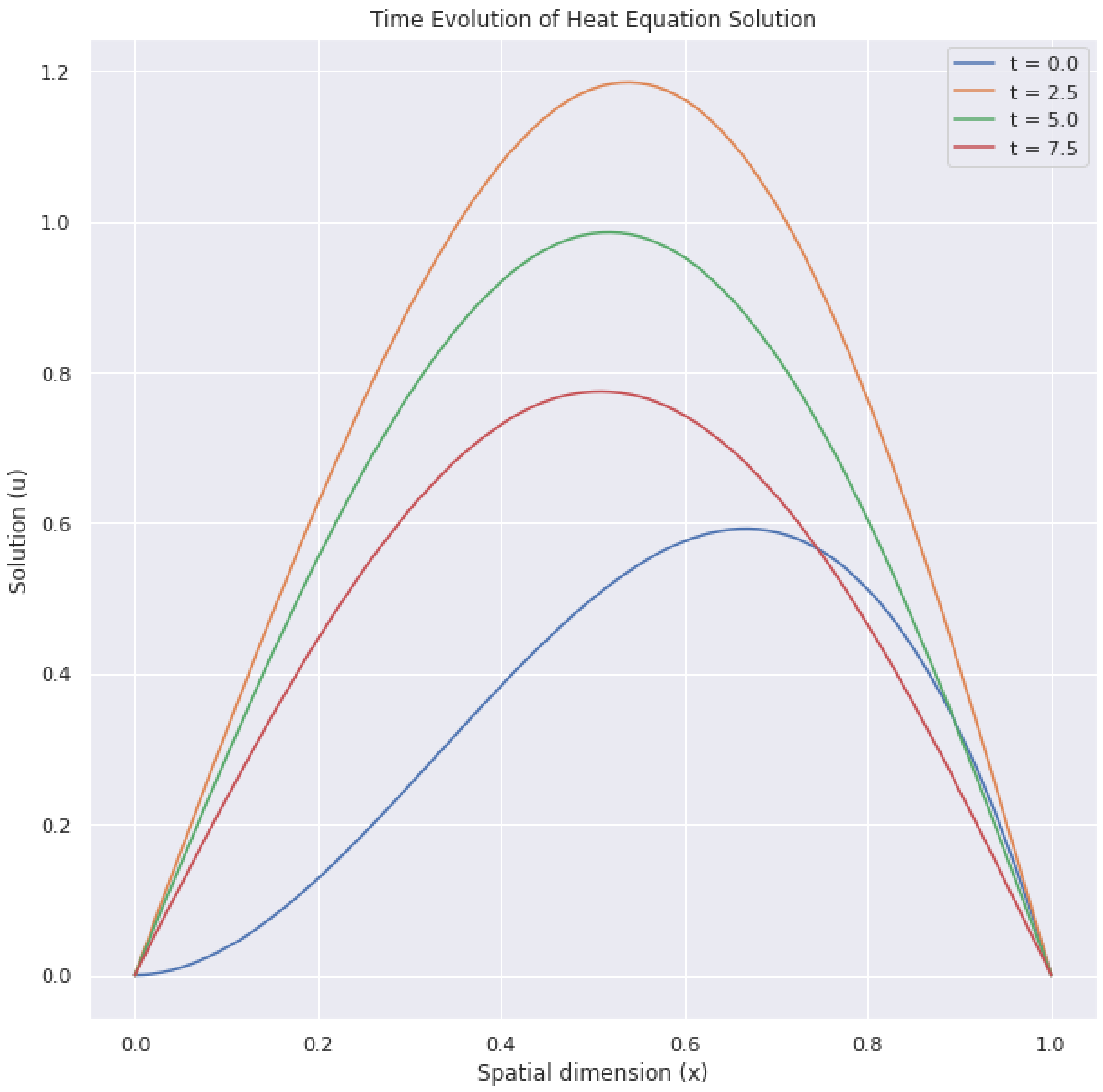
$$\text{IC:} \quad \phi(x,0) = -4x^2(x-1) \tag{10}$$

We have switched back to our constant conductivity and kept the right justified initial conditions, but now we have a source term that decays over time.

[32]:
```
f = lambda x,t: np.sin(np.pi*x)*np.exp(-t)
a = lambda x: 0.01*np.ones(len(x))
ic = lambda x: -4*(x**2)*(x-1)
```

[33]:
```
sol, Nx, Nt = heatEq(f, a, ic, d, hx, ht)
```

[34]:
```
spaceEvolution(sol, Nx, Nt)
```



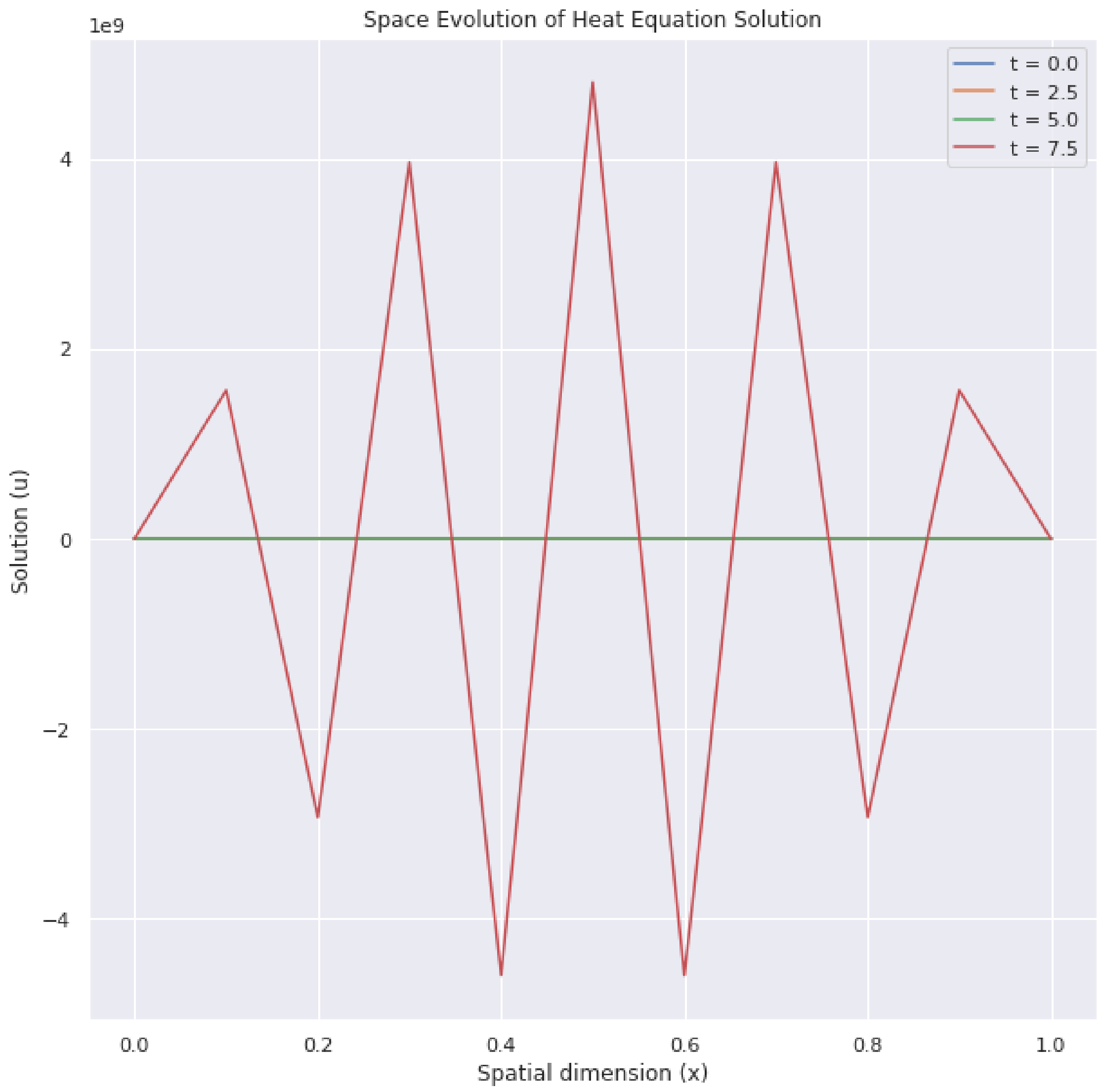
Some slightly more complicated dynamics are taking place, but it is still within our expectations. We note two things: the heat becomes more evenly distributed over time due to the source term and the constant conductivity, and the heat actually increases before decaying. The latter phenomenon occurs because the source term initially overpowers the loss of heat before itself decays to zero.

We can reverse the direction of time in the Heat Equation above by negating the RHS. The results of doing so are shown below.

```
[5]: f = lambda x,t: 0
     a = lambda x: -0.01*np.ones(len(x))
     ic = lambda x: -4*x*(x-1)
```

```
[10]: sol, Nx, Nt = heatEq(f, a, ic, d, .1, .1)
```

```
[11]: spaceEvolution(sol, Nx, Nt)
```



We can see that this causes severe problems as the solution blows up.

2 Problem 2

We will implement the explicit second-order central difference scheme for the wave equation given below:

$$\phi_{tt} = \partial_x[a(x) \cdot \phi_x] + f(x,t) \tag{11}$$
$$\text{IC:} \quad \phi(x,0) = \phi_0 \tag{12}$$
$$\text{IC:} \quad \phi_t(x,0) = \phi_1 \tag{13}$$

We assume that all functions are periodic in x with a period of 1, which gives us the following boundary condition assuming a target interval of $[0,1]$:

$$\text{BC:} \quad \phi(0,t) = \phi(1,t)$$

In the code below we define this central difference scheme where we only assume the functions are periodic over the specified spatial domain.

[6]:
```python
'''
1 dimensional Wave Equation solver using explicit central difference
scheme. We assume that all functions are periodic over the target
interval for the spatial dimension.

f -> f(x,t), input to the system (i.e. RHS)
a -> a(x), velocity
ic -> (phi(x), phi'(x)), initial conditions for spatial dimension
d -> ((x0, xN), (t0, tN)) Tuple of spatial domain and temporal domain
hx -> Spatial step size (optional)
ht -> Temporal step size (optional)
'''
def waveEq(f, a, ic, d, hx=0.1, ht=0.1):
    #Unpack some stuff
    Dx = d[0]
    Dt = d[1]

    #Set up grid
    Nx = np.arange(Dx[0], Dx[1]+hx, hx)
    Nt = np.arange(Dt[0], Dt[1]+ht, ht)

    U = np.zeros((len(Nx), len(Nt))) #Space x Time

    #Check consistency
    assert np.allclose(ic[0](np.array(Dx[0])), ic[0](np.array(Dx[1])))

    #Set IC
    U[:,0] = ic[0](Nx)

    #Condensing terms
    h = (ht/hx)**2

    #Build the matrices A and B
    am, ap = a(Nx-hx/2), a(Nx+hx/2)

    A = np.diag(-(am + ap)) + np.diag(am[1:], 1) + np.diag(am[1:], -1)
    A[0,-1] = ap[-1]
    A[-1,0] = ap[-1]

    B = np.eye(len(Nx)) + (h/2)*A

    #Then we seed our scheme
    U[:,1] = B@U[:,0] + ht*ic[1](Nx) + (ht**2)*(f(Nx, Nt[0]))/2

    for n in range(2, len(Nt)):
        U[:,n] = 2*B@U[:,n-1] - U[:,n-2] + (ht**2)*(f(Nx, Nt[n]))

    return U, Nx, Nt
```

In addition to our temporal snapshot visualization we defined earlier, we will also define a spatial variation that plots a spatial snapshot over the time domain.

[7]:
```
def timeEvolution(U, Nx, Nt, t='Wave'):
    fig, ax = plt.subplots(1,1,figsize=(10,10))

    rng = np.random.default_rng()
    loc = rng.integers(0,len(Nx))

    ax.plot(Nt, U[loc,:])

    ax.set_xlabel('Time dimension (t)')
    ax.set_ylabel('Solution (u)')
    ax.set_title(f'Time Evolution of {t} Equation Solution')
    ax.legend([f'x = {Nx[loc]:.3f}'])
```

We will now verify that our method works as expected by applying it to a number of examples where we can vary the source term, the wave speed, and the initial conditions. The only thing we will specify for all problems is the domain, specifically $x \in [0,1]$ and $t \in [0,10]$.

[8]:
```
d = ((0,1), (0,10))
```

We first consider the following problem..

$$\phi_{tt} = \phi_{xx} \tag{14}$$
$$\text{IC:} \quad \phi(x,0) = \sin(2\pi x) \tag{15}$$
$$\text{IC:} \quad \phi_t(x,0) = 0 \tag{16}$$

Here we have no source term, constant unit speed, a sinusoidal initial position, and no initial velocity. This is the simplest setup, and one that should produce two standing waves. We also note that we take the time and space step sizes to be 0.001 which should be stable given we have constant wave speed of 1.

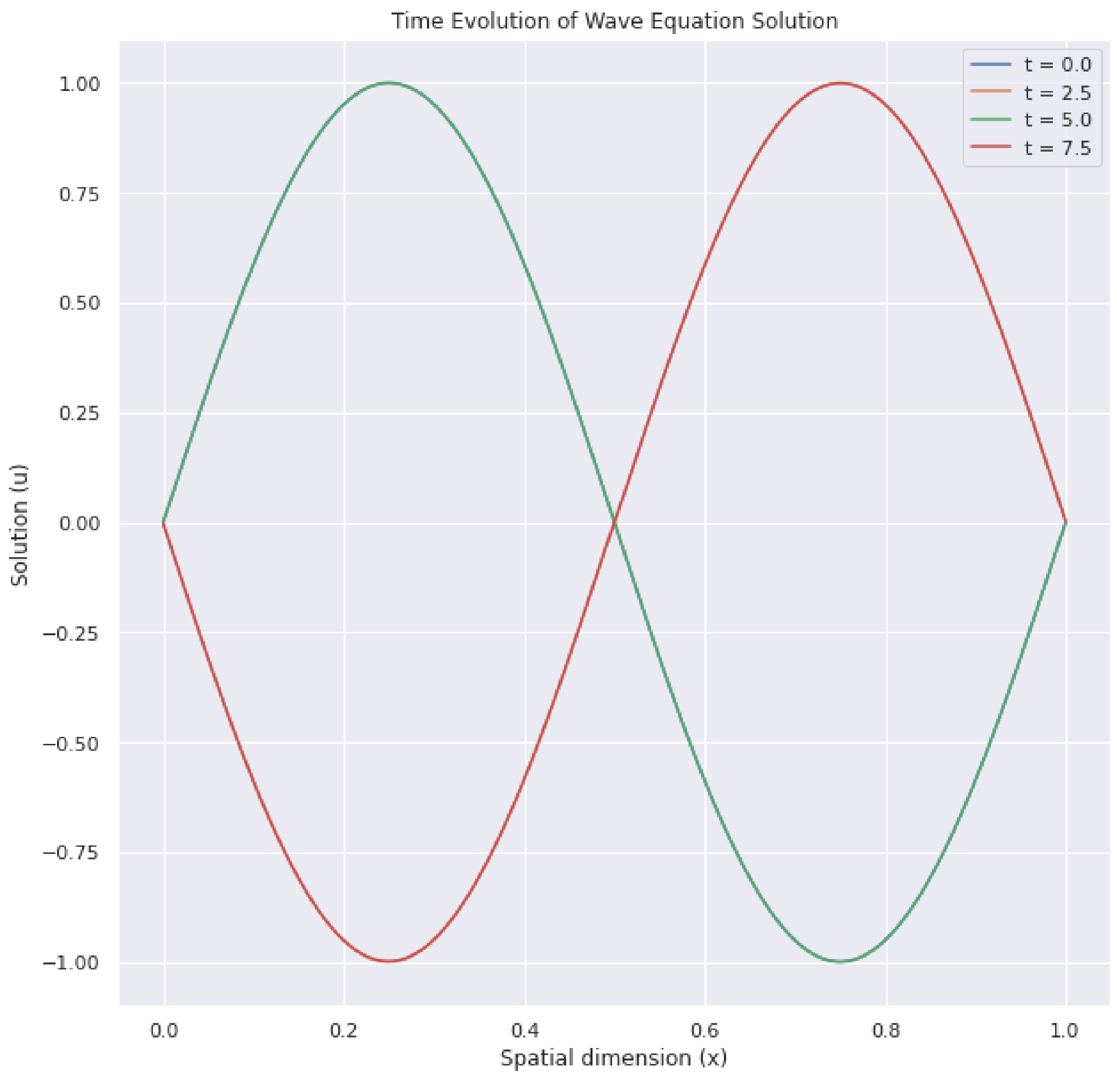
[37]:
```
ht, hx = 0.001, 0.001

f = lambda x,t: np.zeros(x.shape)
```

```
a = lambda x: np.ones(x.shape)
ic1 = lambda x: np.sin(np.pi*x*2)
ic2 = lambda x: np.zeros(x.shape)
```

[38]:
```
sol, Nx, Nt = waveEq(f, a, (ic1, ic2), d, hx, ht)
```

[39]:
```
spaceEvolution(sol, Nx, Nt, t='Wave')
```



There are the two standing waves that we expect. In fact, we cannot even see all the curves because some are hiding behind the others.

Next, we consider the following problem..

$$\phi_{tt} = \frac{\phi_{xx}}{10} \tag{17}$$
$$\text{IC:} \quad \phi(x,0) = \text{Gaussian}(0.5, 0.1) \tag{18}$$
$$\text{IC:} \quad \phi_t(x,0) = 0 \tag{19}$$

We maintain the constant wave speed (but reduce the magnitude) and the initial velocity, but now we have a Gaussian initial condition.

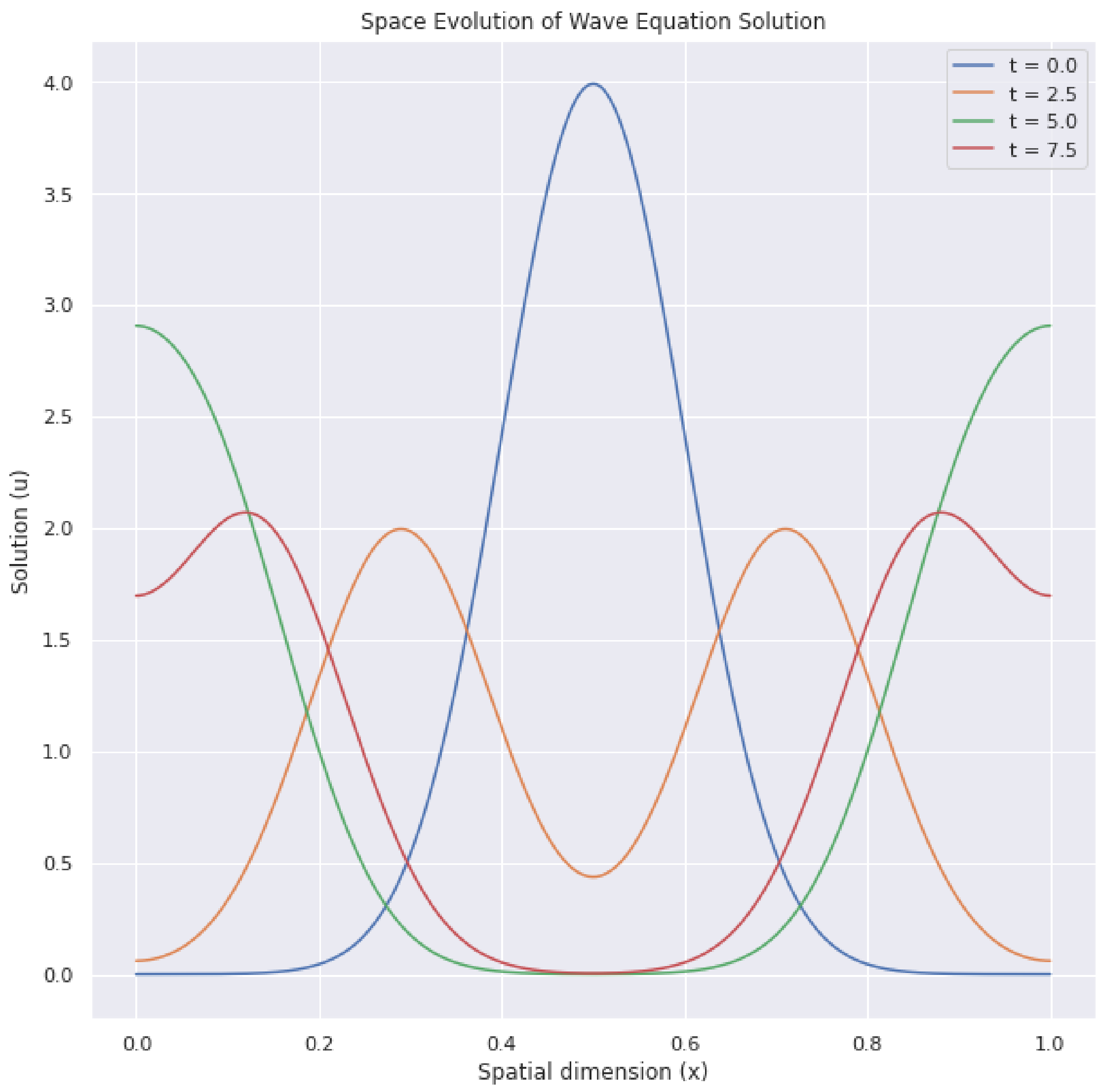
[64]:
```
from scipy.stats import norm

ht, hx = 0.001, 0.001

f = lambda x,t: np.zeros(x.shape)
a = lambda x: 0.1*np.ones(x.shape)
ic1 = lambda x: norm.pdf(x, loc=0.5, scale=0.1) #Gaussian
ic2 = lambda x: np.zeros(x.shape)
```

[65]:
```
sol, Nx, Nt = waveEq(f, a, (ic1, ic2), d, hx, ht)
```

[69]:
```
spaceEvolution(sol, Nx, Nt, t='Wave')
```



Finally, we consider the following problem..

$$\phi_{tt} = \phi_{xx} + 5\cos(t) I_{x=0} \tag{20}$$
$$\text{IC:} \quad \phi(x,0) = 0 \tag{21}$$
$$\text{IC:} \quad \phi_t(x,0) = 0 \tag{22}$$

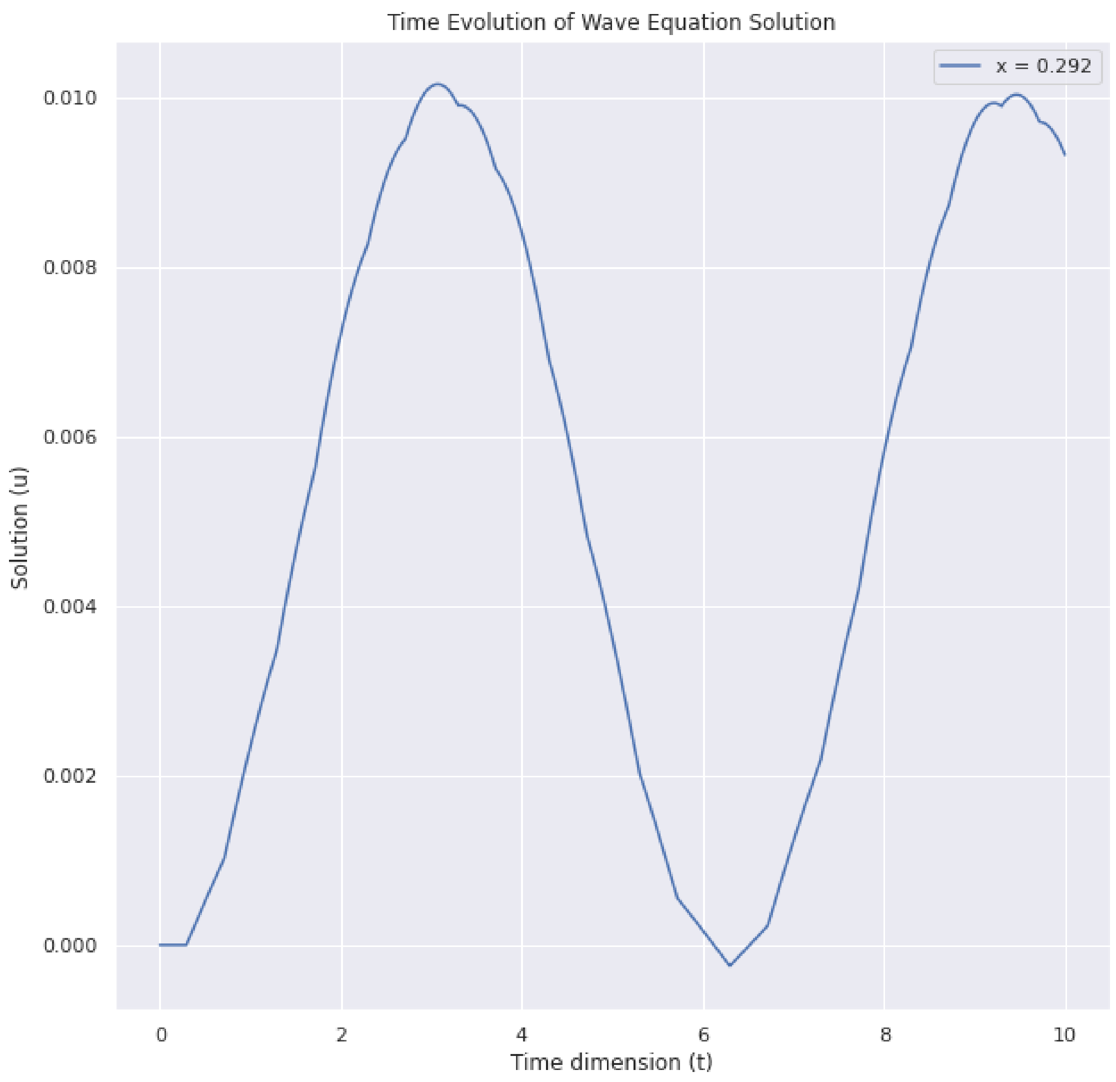
We have a point source and a zero initial conditions.

[123]:
```
ht, hx = 0.001, 0.001

f = lambda x,t: 5*np.array([1 if xi==0 else 0 for xi in x])*np.cos(t)
a = lambda x: np.ones(x.shape)
ic1 = lambda x: np.zeros(x.shape)
ic2 = lambda x: np.zeros(x.shape)
```

[124]:
```
sol, Nx, Nt = waveEq(f, a, (ic1, ic2), d, hx, ht)
```

[125]:
```
timeEvolution(sol, Nx, Nt, t='Wave')
```



The results are what we would expect but they don't look great. This is most likely down to numerical imprecision given the more difficult setup.

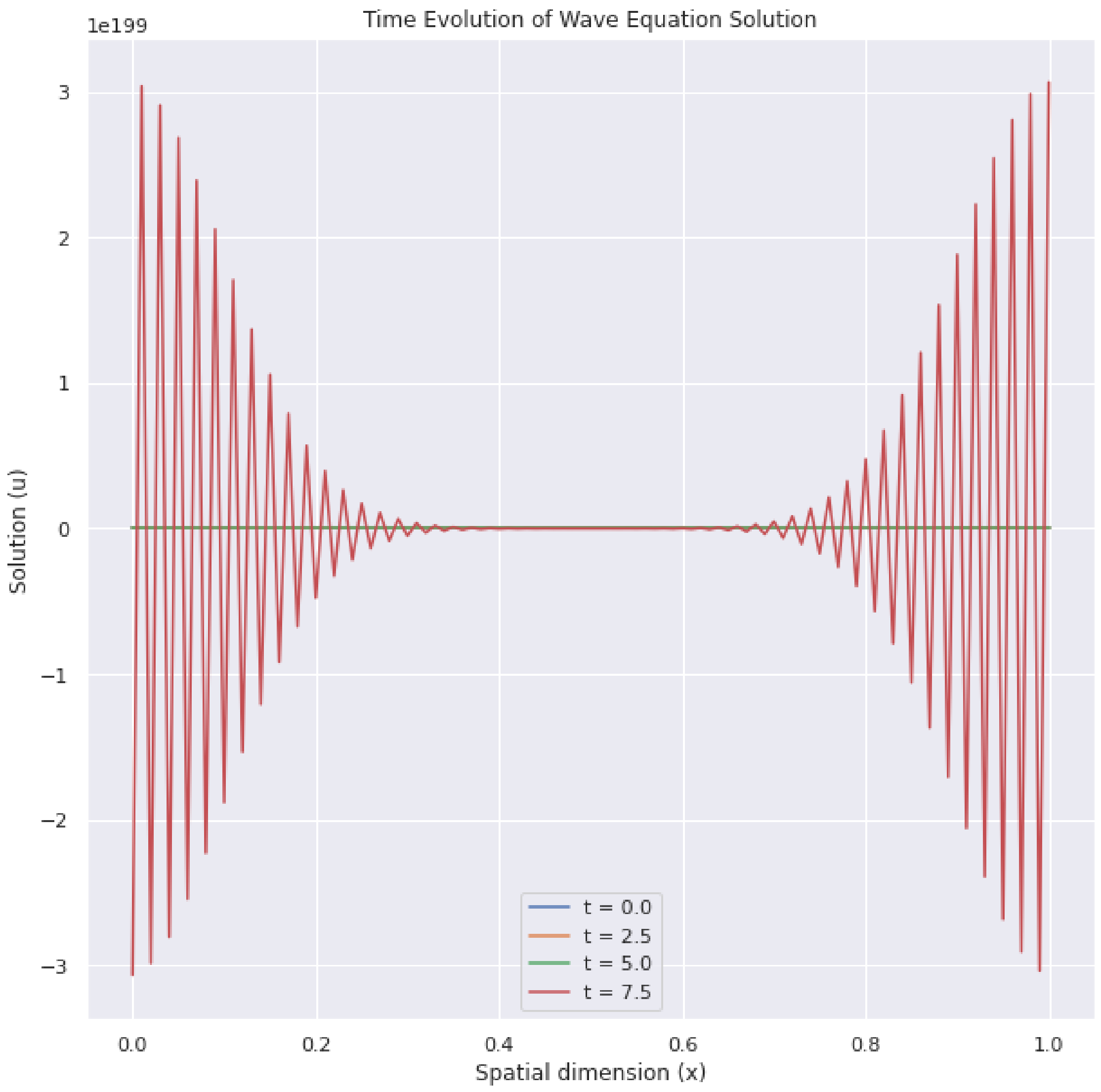
We can change the sign of the RHS of the Wave Equation above just like we did for the Heat Equation and observe the following results.

[129]:
```
ht, hx = 0.01, 0.01

f = lambda x,t: np.zeros(x.shape)
a = lambda x: -0.1*np.ones(x.shape)
ic1 = lambda x: np.sin(np.pi*x*2)
ic2 = lambda x: np.zeros(x.shape)
```

```
[130]: sol, Nx, Nt = waveEq(f, a, (ic1, ic2), d, hx, ht)
```

```
[131]: timeEvolution(sol, Nx, Nt, t='Wave')
```



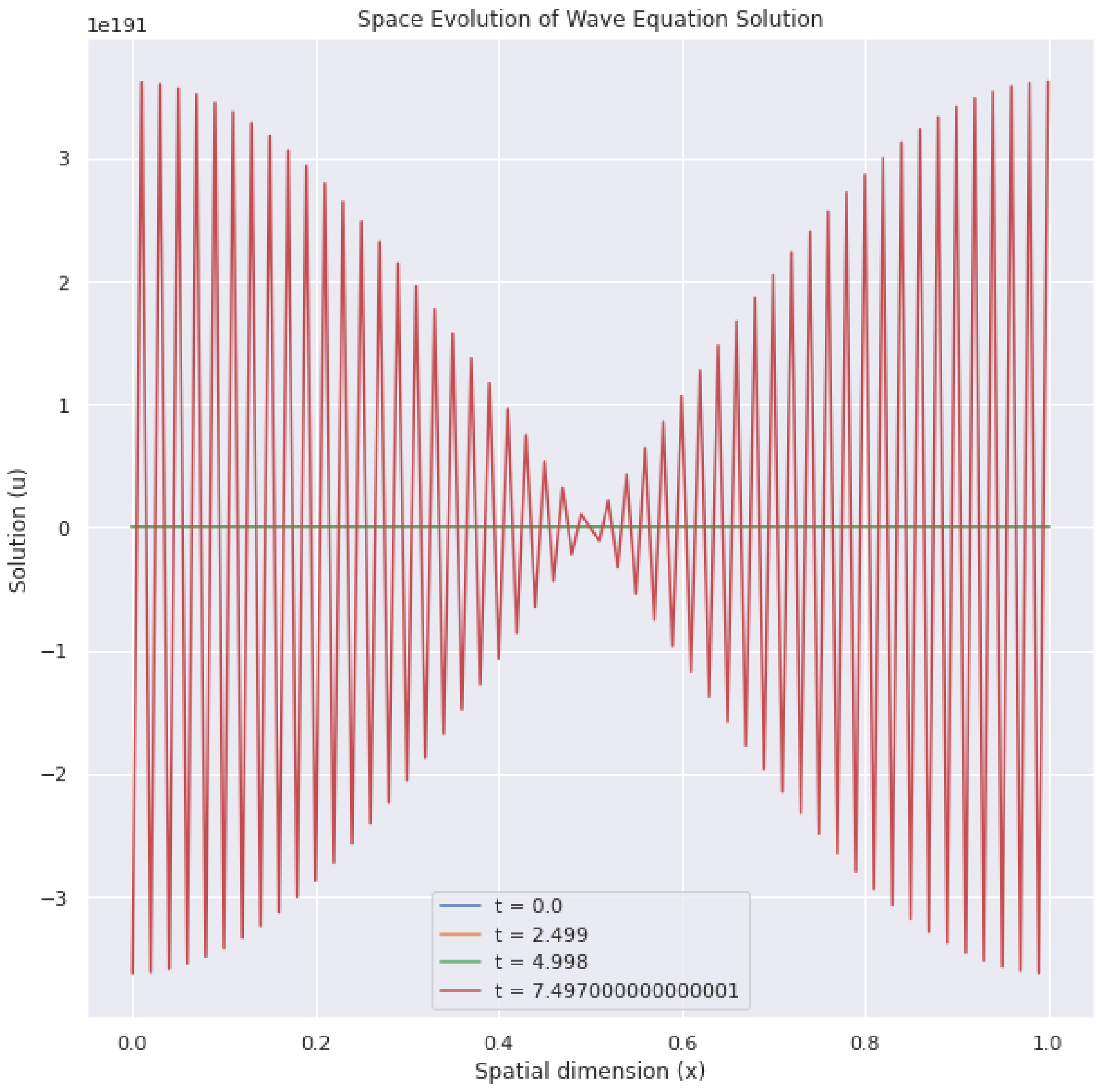
Similarly the solution blows up. We can also violate the step size criterion by taking the time step to be 0.0005 larger the space step. We do this for the original problem and observe the results.

```
[24]: ht, hx = 0.0105, 0.01

      f = lambda x,t: np.zeros(x.shape)
      a = lambda x: np.ones(x.shape)
      ic1 = lambda x: np.sin(np.pi*x*2)
      ic2 = lambda x: np.zeros(x.shape)
```

```
[25]: sol, Nx, Nt = waveEq(f, a, (ic1, ic2), d, hx, ht)
```

[26]:
```
spaceEvolution(sol, Nx, Nt, t='Wave')
```



We can see that even just barely violating this criterion results in an unbounded solution.