FYS2130 Prosjekt

Kadnr.: 15070

7. mai 2017

1 Exercise 1)

We are here going to find the expression for a mass point y_i of a string from its neighbours and previous position. From the exercise text we are given the force on the a mass point as

$$F_i = F_{i,l} + F_{i,r} = -(k_{i-1} + k_i)y_i + k_{i-1}y_{i-1} + k_i y_{i+1} \tag{1}$$

Where $F_{i,l}$ is the force acting on the mass point from the left, and $F_{i,r}$ is the force acting on it from the right. k_i is the spring constant of the i'th spring; y_{i-1} and y_{i+1} are the left and right neighbour, respectively; and k_{i-1} the $i-1$'th spring.

From Newton's second law we get that

$$F_i = m_i \ddot{y}_i \tag{2}$$

And from the exercise text we lastly get the numerical approximation of the second derivative:

$$\ddot{y}_i = \frac{d^2 y_i}{dt^2} \approx \frac{y_i^+ - 2y_i^0 + y_i^-}{(\Delta t)^2} \tag{3}$$

Where y_i^+ is what are interested in finding, namely the position of the mass point y_i in the next time step. y_i^0 is the current position of the mass point, and y_i^- the previous position. We are only going to look at the y-position of each mass point. In reality the mass point would be able to move in the x-direction as well(and z-direction in a 3D case), but we are simplifying the system by only looking at the y-position.

We can now use the information above to find a expression for the position of a mass point in the next time step y_i^+. We begin by inserting equation (1) and (3) into (2):

$$-(k_{i-1} + k_i)y_i + k_{i-1}y_{i-1} + k_i y_{i+1} = m_i \frac{y_i^+ - 2y_i^0 + y_i^-}{(\Delta t)^2} \tag{4}$$

We can now simply solve for y_i^+. We then get the expression for the next y-position/amplitude of the i'th mass point:

$$\boxed{y_i^+ = \frac{\Delta t^2}{m_i}\left(k_{i-1}y_{i-1} - (k_{i-1} + k_i)y_i + k_i y_{i+1}\right) + 2y_i^0 - y_i^-} \tag{5}$$

Notice that this equation is dependent on the current and previous position o the mass point, meaning that we need an initial position *and* a pre-initial position. Finding the pre-initial conditions will depend on the behaviour we want the system to have, and will be discussed below for some of the later cases.

We also have to be careful of the neighbouring mass points. In the middle of the string this posses no problem, but and the ends we have to impose some special rules on the mass points. We are going to look at two methods of dealing with the endpoints:

Open ends:

For the open ends we want the mass points to only feel the force of one of its neighbours – the right neighbour for the first point, and the left of the last point –. This means that we can write Newton's second law for the first and last mass point as:

$$m_0 \ddot{y}_0 = F_{0,r} \qquad m_{N-1} \ddot{y}_{N-1} = F_{N-1,l} \tag{6}$$

We can thus rewrite expression (5) explicitly for the endpoints:

$$y_0^+ = \frac{\Delta t^2}{m_0} \left(-k_0 y_0 + k_0 y_1\right) + 2y_0^0 - y_0^- \tag{7}$$

$$y_{N-1}^+ = \frac{\Delta t^2}{m_{N-1}} \left(k_{N-2} y_{N-2} - k_{N-2} y_{N-1} +\right) + 2y_{N-1}^0 - y_{N-1}^- \tag{8}$$

Reflective ends:

The second type of ends are the reflective ends. These are ends that don't move from their initial position. This means that every wave hitting this point will be reflected. There are two ways of implementing this. The first and simplest method is to just force the position of these mass points to be the initial condition:

$$y_{end}^+ = y_{end}^0 = y_{end}^- \tag{9}$$

The second method is more like reality. A total reflection will happen when a wave hits a domain with a very large impedance, given as

$$z_i = \sqrt{k_i m_i} \tag{10}$$

or a wave on string. We can there for give the endpoints very large impedance. In our case we can give the point a mass $M_i >> m_i$. This will ensure a reflective end. But again we have to find a way to treat the neighbours at the endpoint. The way I did this in my simulation was to simply have open ends with mass M_i, this means that we don't have to think about the neighbours and we get reflective ends. So for the reflective ends y_{end}^+ is given as:

$$y_0^+ = \frac{\Delta t^2}{M_0} \left(-k_0 y_0 + k_0 y_1\right) + 2y_0^0 - y_0^- \tag{11}$$

$$y_{N-1}^+ = \frac{\Delta t^2}{M_{N-1}} \left(k_{N-2} y_{N-2} - k_{N-2} y_{N-1} +\right) + 2y_{N-1}^0 - y_{N-1}^- \tag{12}$$

2 Exercise 2)

We are here going to show that the above way of describing a wave on a string reduces to the 1D wave equation.

We are going to start be introducing a mass density $\mu = m/\Delta x$ and the constant spring stiffness $\kappa = k\Delta x$, here Δx is the distance between the mass points. We are going to assume the spring constants k_i and masses m_i to be constant along the whole spring [1]. This means that we can use that

$$k = \frac{\kappa}{\Delta x}, \qquad m = \mu\Delta x \tag{13}$$

We can now insert these into equation (1) and (2)

$$m_i\ddot{y}_i = \mu\Delta x\ddot{y}_i = F_i = \frac{\kappa}{\Delta x}(y_{i-1} - 2y_i + y_{i+1}) \tag{14}$$

$$\Rightarrow \mu\Delta x\ddot{y}_i = \frac{\kappa}{\Delta x}(y_{i-1} - 2y_i + y_{i+1}) \tag{15}$$

In the expression for F_i I pulled out $\kappa/\Delta x$ and rearranged the y-terms. We then move $\mu\Delta x$ to the right side, and get

$$\ddot{y}_i = \frac{\kappa}{\mu} \cdot \frac{y_{i-1} - 2y_i + y_{i+1}}{(\Delta x)^2} \tag{16}$$

If we remember back to how discretized the second derivative of y with respect to time in (3) we see that the last term on the right side look eerily similar. This is infact the discretized of the second derivative of y with respect to x

$$\frac{d^2y_i}{dx^2} \approx \frac{y_{i-1} - 2y_i + y_{i+1}}{(\Delta x)^2} \tag{17}$$

We are going to use that this go to a equivalence as Δx goes to zero. So if we insert this into (16) we get

$$\ddot{y}_i = \frac{\kappa}{\mu}\frac{d^2y_i}{dx^2} \tag{18}$$

This means that a wave on a string obeys the differential equation:

$$\boxed{\frac{d^2y_i}{dt^2} = v_B^2\frac{d^2y_i}{dx^2}} \tag{19}$$

This is the 1D wave equation. Comparing this with (18), we get that a wave on a string propagate with the velocity v_B given as

$$\boxed{v_B^2 = \frac{\kappa}{\mu}} \tag{20}$$

[1]This is not necessary to for the end points, and especially for reflective ends. But for a solution of the wave equation, the imposed boundary conditions will ensure the right behaviour of the ends. We can therefore ignore the ends for the derivation of the wave equation.

3 Exercise 3)

When we are simulating the wave, we need to ensure that the program is numerically stable. The requirement for this to be true is that

$$\frac{\Delta x}{\Delta t} \geq v_B \tag{21}$$

This comes form the Courant–Friedrichs–Lewy condition for converges of a numerical solution for partial differential equations like the wave equation. Broadly this condition says that the time step has to be smaller that the time it takes the wave to move the next mass point, which is equivalent to the above (21).

To see what kind of limitations this means or our simulation, we are going to rewrite v_B with k and m:

$$v_B = \sqrt{\frac{\kappa}{\mu}} = \sqrt{\frac{k\Delta x}{m/\Delta x}} = \Delta x\sqrt{\frac{k}{m}} \tag{22}$$

We insert this into (21)

$$\frac{\Delta x}{\Delta t} \geq \Delta x\sqrt{\frac{k}{m}} \tag{23}$$

Doing the algebra we get the convergence condition:

$$\boxed{\Delta t \leq \sqrt{\frac{m}{k}}} \tag{24}$$

As we can see, the convergence condition no longer involve Δx. If we look at the expression for y_i^+ (5) we see that there is no mention of Δx, meaning that both the stability and precision of the simulation is wholly independent of Δx.

The conclusion is thus that while the value of Δt is highly important for the accuracy of the simulation, Δx is just a scaling factor which we are free to set as 1 for our simulation.

4 Exercise 4)

5 Exercise 8)

We are going to make a triangle centred around mass point 30. This means the our system starts with a initial condition

$$y_i^0 = \begin{cases} \frac{i}{30} & 0 \leq i \leq 29 \\ \frac{59-i}{30} & 30 \leq i \leq 58 \\ 0 & \text{else} \end{cases} \tag{25}$$

We are then going to make it move to the right. To do make this happen we have to find specific pre-initial conditions y_i^-. There are two different methods to make this happen. One method where we set Δt to some specific value, where we can just center the triangle one mass point/grid point backwards for y_i^-. The second method uses interpolation to find have the triangle looked one time step back.

Method 1:

We are going to start with the general equation for a wave travelling to the right $F(x - vt)$. Since we have a discretized system, we know that in one time step Δt, the wave has to have travelled some distance $n\Delta x$, where $n \in \mathbb{N}$. This gives us that

$$F(x - v\Delta t) = F(x - n\Delta x) \Rightarrow v\Delta t = n\Delta x \tag{26}$$

Using the definition of v we get

$$\sqrt{\frac{k}{m}}\Delta x\Delta t = n\Delta x \Leftrightarrow \Delta t = n\sqrt{\frac{m}{k}} \tag{27}$$

Comparing this to the convergence condition (24) we see that we must have $n = 1$. So we get that

$$\boxed{\Delta t = \sqrt{\frac{m}{k}}} \tag{28}$$

For this specific value of Δt(the largest possible) we can just center the triangle one mass point to the left to get a triangle moving right.

Method 2:

If we instead want a method which we can use for every possible value of Δt we have to use interpolation. During one time step the a mass point a moved some Δy either up or down(or not at all). We can use dimensional analysis to find the value of Δy. We take the derivative of the function of the initial condition with respect to the number of mass points((27) is dependent on i, not x), y', we know that this is the amount the triangle has move from one mass point to the next Δi[2]

$$y' = \frac{\Delta y}{\Delta i} \tag{29}$$

The velocity v_B can be written as $v_B = \sqrt{k/m}\Delta x$ meaning that $\sqrt{k/m}$ is the velocity the wave moves from one mass point to the next during a time step.

$$\sqrt{\frac{k}{m}} = \frac{\Delta i}{\Delta t} \tag{30}$$

[2] Δi is trivially equals to 1, but is used here as a useful marker from the dimensional analysis

If we combine these we get that

$$\Delta y = y' \sqrt{\frac{k}{m}} \Delta t \tag{31}$$

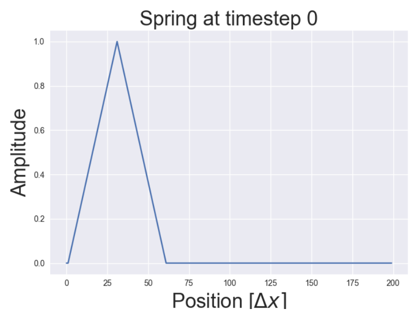
And from this we get the function for the pre-initial conditions:

$$\boxed{y_i^- = y_i^0 - \Delta y_{i+1} = y_i^0 - y'_{i+1} \sqrt{\frac{k}{m}} \Delta t} \tag{32}$$

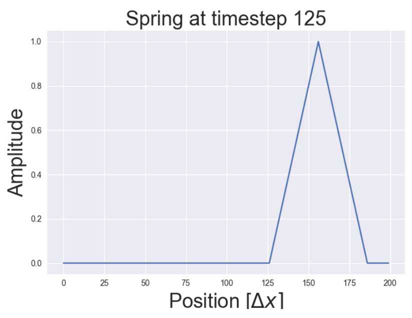
Notice that this is dependent on y'_{i+1}, the derivative one step to the right. We can so look at the results:

Results Method 1:

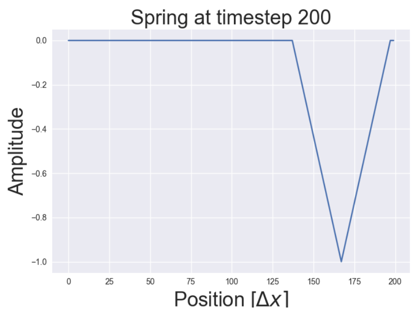
For the following results I have chosen to center the triangle at 31 instead of 30, due to have to move the triangle one mass point to the left for the pre-initial conditions. I have also done this in the *method 2*-results, to have comparable results for the two methods



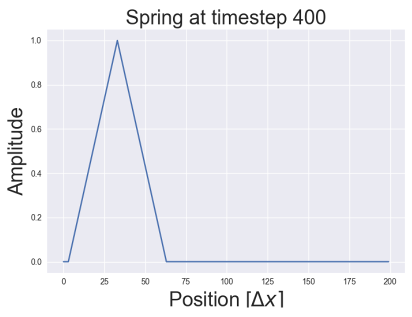
(a) The initial triangular wave.



(b) The wave after 125 time steps. It has move almost across the string.



(c) The wave after 200 time steps. The wave has just been reflected of the right wall.

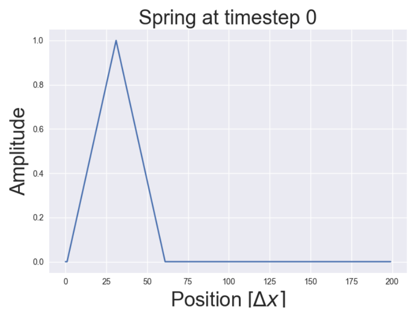


(d) The wave after 400 time steps. The wave has just been reflected of the left wall.

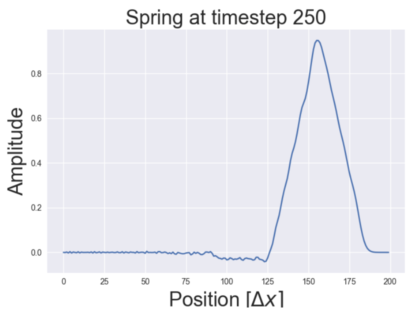
Figur 1: A triangular wave moving to the right using $\Delta t = \sqrt{\frac{m}{k}}$. The triangle holds its shape, moves to the right, reflects off the right wall, moves back and reflects off left wall. As we can see the wave behaves exactly as we expects.

We can see from the above figure that the wave moves how it suppose to. This method of making the wave move to the right does not depend on the size or initial condition, so it is very versatile, and very exact. The only problem is the set size of Δt. If we for some reason want to lower the time step, we have to use the interpolation. But as we will see below this creates imperfections in the simulation.

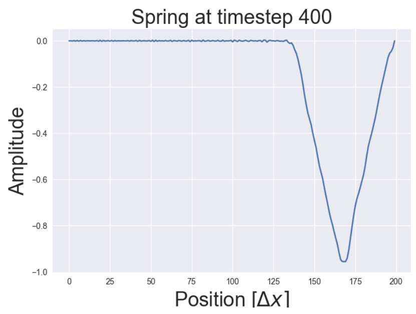
Results Method 2:



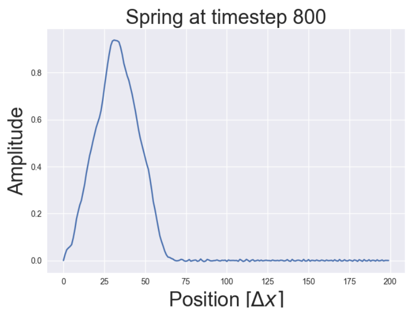
(a) The initial triangular wave.



(b) The wave after 250 time steps. It has move almost across the string.



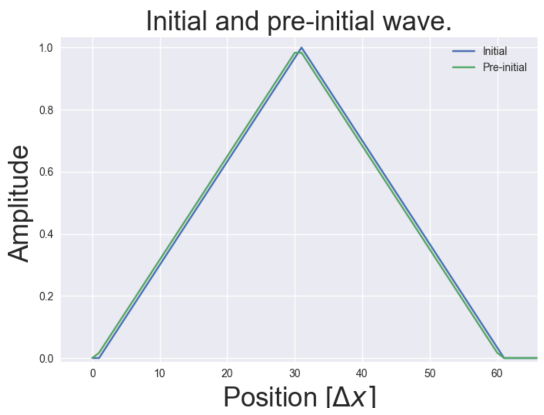
(c) The wave after 400 time steps. The wave has just been reflected of the right wall.



(d) The wave after 800 time steps. The wave has just been reflected of the left wall.

Figur 2: A triangular wave moving to the right using interpolation. We can see that the moving wave is not a perfect triangle. The triangle has after the initial time steps no longer amplitude 1, and there are ripples following it. But other than that, the wave behaves as we expect. This is simulated with $\Delta t = 0.5 \cdot \sqrt{\frac{m}{k}}$

As we see in the figure above, the triangle moves to the right, reflects of the right wall. And due to the near infinite impedance of the end point its amplitude changes sign. The reflected wave moves back and reflects of the left wall. We can see that we have a amplitude less than the original, and ripples behind the wave. This is due to us trying to impose continuity on a discrete system. We can look at the initial and the pre-initial wave:



Figur 3: The initial and pre-initial triangle. They are almost identical (and shifted) but for the top and the endpoints.

We can see that because we are interpolate, the top of the triangle is missing(it only exist if the mass point had been $1 \cdot v\Delta t$ away), and endpoints are a kink. These are the reasons we are getting the imperfections in the triangular wave.

But other then the above mentioned effects, the triangle wave behaves as we expected. Moving to the right and reflecting of the walls.

A Code for exercise 8:

```
# -*- coding: utf-8 -*-
"""
Created on Sun May 07 15:06:57 2017

@author: dulte
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim
import seaborn
from scipy.signal import argrelextrema


class waveSolver:

    def __init__(self, edges, m, k, T,N, dt,dx):

        """Checks if legal option for ending is given"""
        if edges != "open" and edges != "reflective":
            print "That is not a valid type of boundary condition!"
            exit()

        """Checks if dt fulfills the convergence conditions"""
        if dt > np.sqrt(float(m)/float(k)):
            print "m,k and dt are chosen in a way that makes dt to large!"
            exit()

        self.edges = edges
        self.m = m
        self.k = k
        self.N = N
        self.dx = dx
        self.dt = dt
        self.T = T

        self.y = np.zeros((N,T))
        self.ms = np.ones(N)*self.m #Array of masses. Constant mass set as
            diffult



    def solve(self,y,y_prev,plot = True):


        M =1e10 #'infinite' mass of the endpoints

        self.y[:,0] = np.copy(y)

        """ Im setting y_i^- for the first timestep in the last timestep,
        this way we dont need any special treatment of this step
        (since '-1' is the last position of an array in python)"""
        self.y[:,-1] = np.copy(y_prev)

        if self.edges == "reflective":
            self.ms[0] = float(M) #Givs the endpoint there 'infinite' mass
            self.ms[-1] = float(M)

        """This factor is predefined to save computational time"""
        self.factor = (self.dt**2/(self.ms))*self.k


        for t in xrange(self.T-1):

            self.y[0,t+1] = 0#(self.dt**2/float(self.ms[0]))*(self.k*self.y
                [1,t] - self.k*self.y[0,t] )+2*self.y[0,t] - self.y[0,t-1]
            self.y[self.N-1,t+1] = 0#(self.dt**2/float(self.ms[self.N-1]))
                *(-self.k*self.y[self.N-1,t] + self.k*self.y[self.N-2,t])+2*
                self.y[self.N-1,t] - self.y[self.N-1,t-1]

            self.y[1:-1,t+1] =    self.factor[1:-1]*(self.y[2:,t] - 2*self.y
                [1:-1,t] + self.y[0:-2,t])+2*self.y[1:-1,t] - self.y[1:-1,t
                -1]




    """Method takes either a list of frames one wants to plot, or just a
        single frame.
    Set animate to true if you want to animate(is somewhat unstable, so may
        get errors.
    And dont seem to work on iOS"""
    def plotWave(self,frames,animate = False):
        if animate:
            fig = plt.figure()
            animationImages = []
            for t in range(T):
                animationImages.append(plt.plot(self.y[:,t], 'r'))

            ani = anim.ArtistAnimation(fig, animationImages,interval = self.
                dt*1000, blit = True)
            #self.fig.show()
            plt.show()


        try:
            for f in frames:
                if f > self.T:
                    continue
                plt.plot(self.y[:,f])
                plt.title("Spring at timestep %g" %f, fontsize = 25)
                plt.xlabel("Position [$\Delta x]$", fontsize = 25)
                plt.ylabel("Amplitude", fontsize = 25)
                plt.show()
        except:
            if frames > self.T:
                return
            plt.plot(self.y[:,frames])
            plt.title("Spring at timestep %g" %frames, fontsize = 25)
            plt.xlabel("Position [$\Delta x]$", fontsize = 25)
            plt.ylabel("Amplitude ", fontsize = 25)
            plt.show()

def intialConditions(y,N):

    for i in xrange(N):

        if i >= 2 and i <= 31:
            y[i] = (i-1)/30.
        elif i >= 32 and i <= 61:
            y[i] = (61- i)/30.
        else:
            y[i] = 0

def prevIntialConditions(y,yp,dt,k,m,N):

    for i in xrange(N):

        if i >= 1 and i <= 30:
            yp[i] = y[i] + 1/30.*dt*np.sqrt(k/m)
        elif i >= 31 and i <= 60:
            yp[i] = y[i] - 1/30.*dt*np.sqrt(k/m)
        else:
            y[i] = 0




if __name__ == "__main__":
    """Declare variables"""
    m = 0.02 #kg
    k = 10. #kg/s^2
```

```
dt = .5*np.sqrt(m/k)#Use for method 2
dt = 1*np.sqrt(m/k) #Use for method 1
dx = 1

N = 200

T = 1200
edges = "reflective"

"""Makes the initial- and pre-initial conditions"""
y0 = np.zeros(N)
y_m = np.zeros(N)

intialConditions(y0,N)
prevIntialConditions(y0,y_m,dt,k,m,N)

"""For plotting the initial and pre-initial conditions"""
plt.plot(y0)
plt.plot(y_m)
plt.title("Initial and pre-initial wave.", fontsize = 25)
plt.xlabel("Position [$\Delta x]$", fontsize = 25)
plt.ylabel("Amplitude ", fontsize = 25)
plt.legend(["Initial","Pre-initial"])
plt.show()


"""Solves the wave equation and analyse the resulats"""
wave = waveSolver(edges,m,k,T,N,dt,dx)

wave.solve(y0,y_m)
wave.plotWave(frames = [0,125,200,400], animate = False) #Use for method
    1
#wave.plotWave(frames = [0,250,400,800], animate = False) #Use for
    method 2
```