Assignment 1 - Advanced Computational Science

Liam O'Sullivan – 10309537

March 19, 2014

Question 1

(1)

We are interested in solving the 1-d wave equation with $c = 1$,

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} \tag{1}$$

using central differences,

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{\delta_t^2 u_j^n}{\tau^2}, \qquad \frac{\partial^2 u}{\partial x^2} \approx \frac{\delta_x^2 u_j^n}{h^2} \tag{2}$$

so our equation becomes

$$\frac{\delta_t^2 u_j^n}{\tau^2} = \frac{\delta_x^2 u_j^n}{h^2} \tag{3}$$

where τ is the time step size, and h is the x step size.

The operator δ_x^2, when applied to u_j^n, gives

$$\delta_x^2 = u_j^{n-1} - 2u_j^n + u_j^{n+1}$$

allowing us to expand out for both spatial and temporal derivatives and rearrange for u_j^{n+1}, giving

$$u_j^{n+1} = \nu^2 \left[u_{j-1}^n + u_{j+1}^n\right] - u_j^{n-1} + 2(1 - \nu^2)u_j^n \tag{4}$$

with $\nu = \frac{\tau}{h}$.

A program was written in Python 3.3 to solve this problem with Dirichlet boundary conditions, $u(-7, t) = u(7, t) = 0$, on the intervals $x \in [-7, 7]$, $t \in [0, 14]$. The boundaries are imposed by only computing from $j = 1$ to $j = j_{\max} - 1$, neglecting both endpoints in the array, as the entire array is initialised to 0. The initial condition is given by $u(x, 0) = e^{-x^2}$. The program is included in appendix A. Values for h, τ were taken to be $h = 0.1$ and $\tau = \frac{1}{10\sqrt{2}}$, with τ set by h as discussed in section 1.2, and h justified in section 1.3.

The solutions at various times are shown in figure 1. The wave is observed to split in two, propagating half in either direction. At the boundaries, the wave is inverted, travelling back to superimpose into a mirror image of the initial condition at $t = 14$.

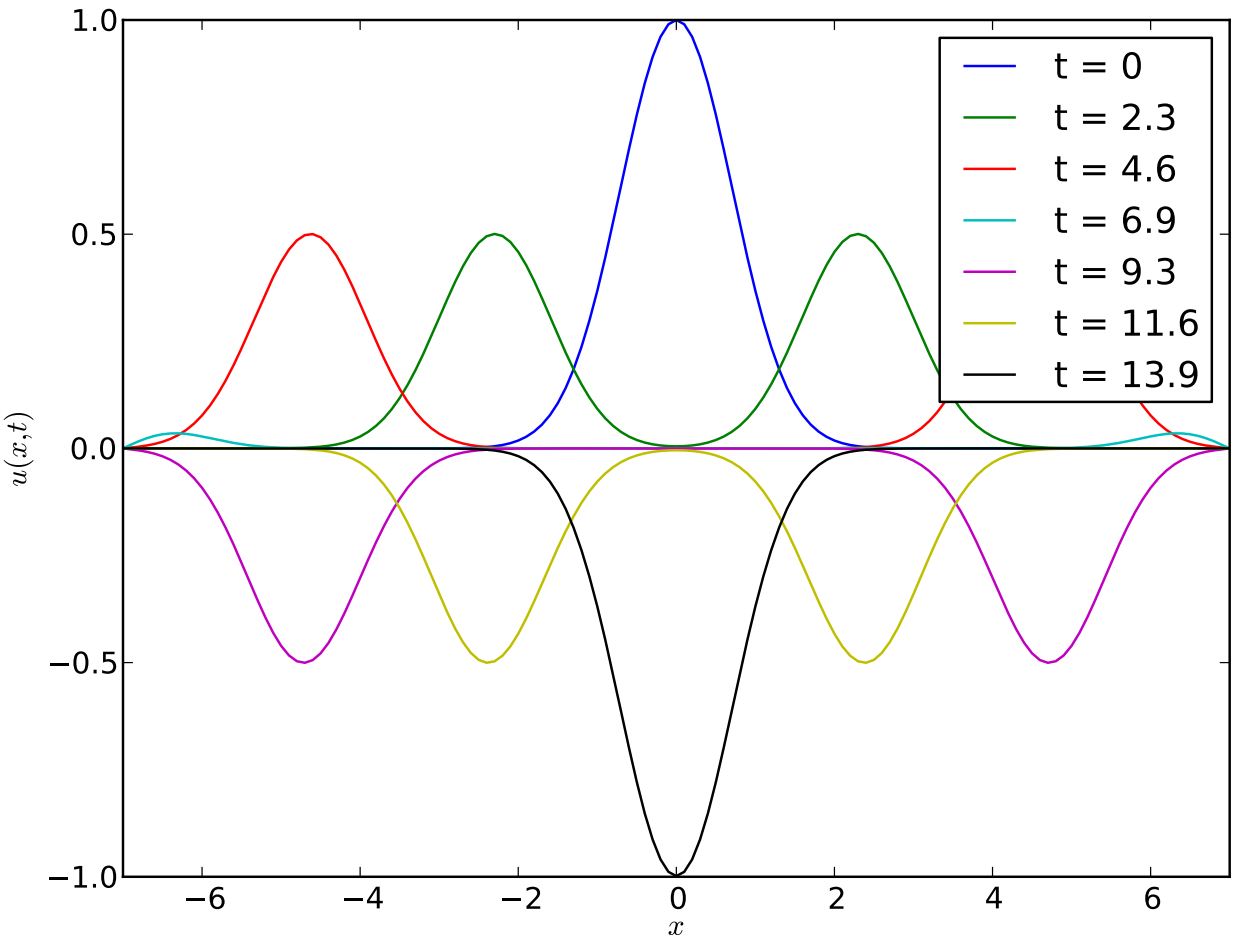

Figure 1: Solution to the wave equation with Gaussian initial condition, $h = 0.1$.

(2)

Our expression for u_j^n using Central Differences is given in equation 4. To perform Fourier/von Neumann stability analysis, we take $u_j^n = \xi^n e^{ikhj}$ as a single Fourier mode solution, and see how our *amplification factor*, ξ, behaves as a function of τ, h, k. Substituting this into equation 4 gives

$$\xi^{n+1}e^{ikhj} = \nu^2\left[\xi^n e^{ikh(j-1)} + \xi^n e^{ikh(j+1)}\right] - \xi^{n-1}e^{ikhj} + 2(1-\nu^2)\xi^n e^{ikhj}.$$

Dividing across by $\xi^{n-1}e^{ikhj}$ and cleaning up gives

$$\xi^2 + \alpha\xi + 1 = 0 \tag{5}$$

with

$$\begin{aligned}\alpha &= -\nu^2\left[e^{-ikh} + e^{ikh} - 2\left(1 - \frac{1}{\nu^2}\right)\right] = -2\nu^2\left[\cos(kh) - \left(1 - \frac{1}{\nu^2}\right)\right] \\ &= -2\nu^2\left[1 - \sin^2\left(\frac{kh}{2}\right) - \left(1 - \frac{1}{\nu^2}\right)\right] = 2\nu^2\left[\sin^2\left(\frac{kh}{2}\right) - \frac{1}{\nu^2}\right]\end{aligned}$$

This is a quadratic equation with roots

$$\xi_\pm = \frac{1}{2}\left(-\alpha \pm \sqrt{\alpha^2 - 4}\right) = \frac{1}{2}\left(-\alpha \pm i\sqrt{(2-\alpha)(2+\alpha)}\right).$$

Inserting α into the above gives

$$\begin{aligned}\xi_\pm &= -\frac{1}{2}\left[2\nu^2\sin^2\left(\frac{kh}{2}\right) - 1\right] \pm \frac{1}{2}\sqrt{\left(4\nu^2\sin^2\left(\frac{kh}{2}\right) - 2\right)^2 - 4} \\ &= -\left[2\nu^2\sin^2\left(\frac{kh}{2}\right) - 1\right] \pm \nu\sin\left(\frac{kh}{2}\right)\sqrt{4\nu^2\sin^2\left(\frac{kh}{2}\right) - 2}\end{aligned}$$

We consider $\tau > 0$ and $h > 0$, and by extension $\nu > 0$. From here, it is immediately apparent that if $\nu > 1$, both terms will be greater than 1 for some values of k, making the method unstable. For $\nu = 1$, $|\xi_-|$ can for some values of k, be greater than one. A good choice seems to be $\nu = \frac{1}{\sqrt{2}}$, as then the above reduces to

$$\begin{aligned}\xi_\pm &= -\left[\sin^2\left(\frac{kh}{2}\right) - 1\right] \pm \sin\left(\frac{kh}{2}\right)\sqrt{\sin^2\left(\frac{kh}{2}\right) - 1} \\ &= -\left[\sin^2\left(\frac{kh}{2}\right) - 1\right] \pm \sin\left(\frac{kh}{2}\right)\sqrt{-\cos^2\left(\frac{kh}{2}\right)} \\ &= -\left[\sin^2\left(\frac{kh}{2}\right) - 1\right] \pm i\sin\left(\frac{kh}{2}\right)\cos\left(\frac{kh}{2}\right)\end{aligned}$$

In the limits of $h \to 0$ or $k \to 0$, the sin terms go to 0, and we are left with $\xi_\pm = 1$. Elsewhere, the function moves between 0 and 1 as a function of k. Otherwise, when considering $|\xi_\pm|$, it is calculated by

$$\xi_\pm = \sqrt{\mathrm{Re}\left(\xi_\pm\right)^2 + \mathrm{Im}\left(\xi_\pm\right)^2}.$$

This demonstrates that ξ_+ and ξ_- are equivalent for $\nu = \frac{1}{\sqrt{2}}$. It is worth noting that in the above workings on the expression for $\xi_\pm$, many $\pm$s were absorbed by the initial $\pm$ due to working with the square root. This leads me to believe that both $\xi_\pm$ are not physically significant, or correspond to the amplifications forward and backward in time. In the quadratic form, we consider what happens when we require $\xi \in [-1, 1]$. It is clear that α as defined earlier must be bound to the region $[-2, 2]$, which sets a clear requirement that $\nu \in (0, 1]$[1] is a limit beyond which stability will not be achieved. This constrains $\tau \leq h$ for stability. However, I have chosen to take $\nu = \frac{1}{\sqrt{2}}$ for the computations in this report, a situation where $\xi \in [0, 1]$ is guaranteed, as this avoids the possibility of one of $\xi_\pm$ blowing the solution up, should I be wrong.

Note: $\nu = 1$ was tested in these exercises and did not lead to the solution blowing up in the time intervals used.

[1] ν could be negative, but we have previously constrained it to be positive.

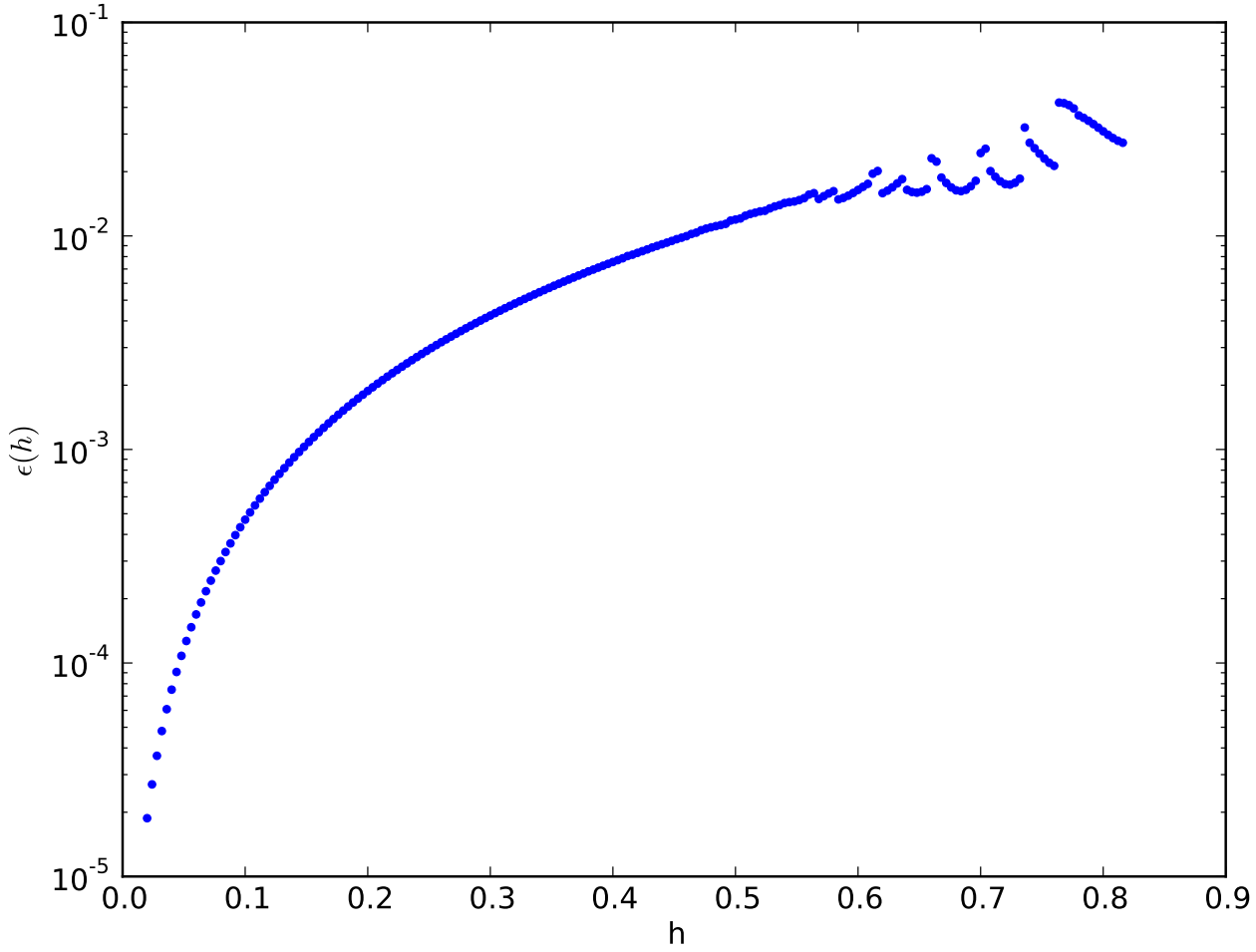

Figure 2: Relative error as a function of h.

(3)

In order to find an optimal value of h for our solutions, we define the *relative error*,

$$\epsilon(h) = 1 - \frac{L_{h/2}}{L_h} \tag{6}$$

where L_h is the L^2-norm of the solution computed with h, and $L_{\frac{h}{2}}$ with $\frac{h}{2}$. The L^2-norm is computed numerically as

$$L_h = \sqrt{h \cdot \sum_{j=0}^{j_{\max}} u_j^{t=10.5}} \tag{7}$$

at $t = 10.5$.

A program was written in Python 3.3 to compute this function for various values of h, available in appendix C. The program defines a function which takes the h value as an argument, performs the integration, and then returns the L^2-norm at $t = 10.5$. As such, for each h value, this is called twice; once for h and once for $\frac{h}{2}$. Plotted in figure 2 is the function $\epsilon(h)$ for various values of h. We are told to choose a value of h such that the relative error is $\approx 10^{-3}$. As such, I chose 0.1, as this is a round number giving relative error $\epsilon \approx 3 \times 10^{-4}$.

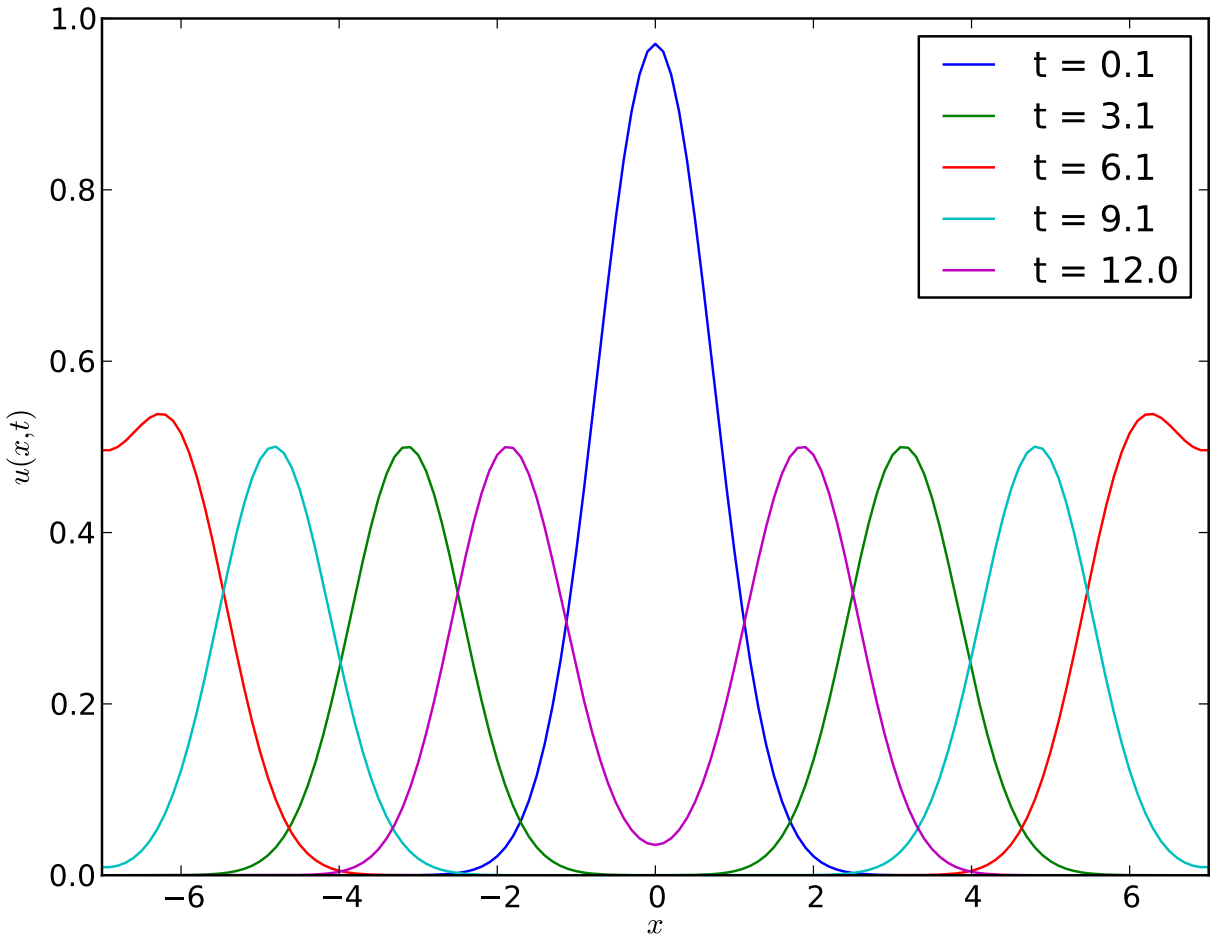

Figure 3: Solution to the wave equation with Gaussian initial condition with Neumann boundaries.

(4)

This exercise is a repeat of the first, but using Neumann boundary conditions; $\frac{\partial}{\partial x}(x_{\min}) = \frac{\partial}{\partial x}(x_{\max}) = 0$. These were imposed by setting $u_0^n = u_1^n$ and equivalent on the max boundary. The program used is included in appendix C. The solutions to the problem at various times are plotted in figure 3, showing the wave initially propagate as in section 1.1, but proceeding to reflect from the walls without inverting as before.

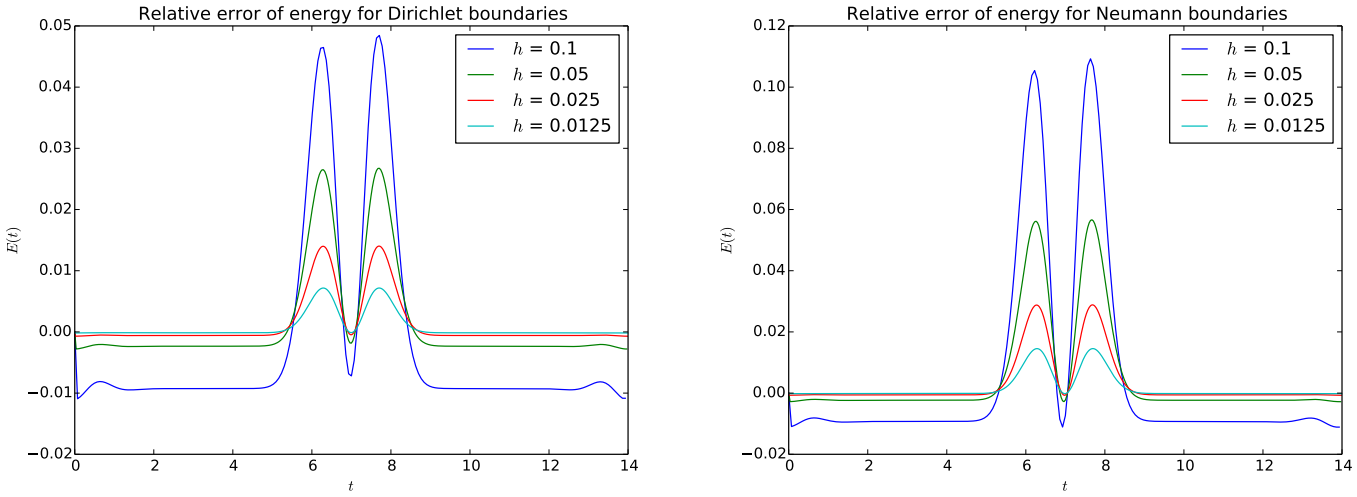

Figure 4: Relative error for Dirichlet(left) and Neumann(Right) boundary conditions.

(5)

Analytical Energy Conservation

$$\mathcal{H}(x,t) = \frac{1}{2}\left(\frac{\partial u}{\partial t}\right)^2 + \frac{1}{2}\left(\frac{\partial u}{\partial x}\right)^2 \tag{8}$$

$$E(t) = \int_{x_{\min}}^{x_{\max}} \mathcal{H}(x,t)dx \tag{9}$$

Numerical Results

The Hamiltonian density is defined in equation 8. As such, we compute an array of values, $\mathcal{H}_j^n$. At the boundaries of space or time, forward/backward differences are used in that quantity, with Double-Interval central differences used elsewhere, for example

$$\mathcal{H}_j^n = \frac{1}{2}\left[\frac{1}{2\tau}\left(u_j^{n+1} - u_j^{n-1}\right)\right]^2 + \frac{1}{2}\left[\frac{1}{2h}\left(u_{j+1}^n - u_{j-1}^n\right)\right]^2$$

generally, with variations where $j = 0, j_{\max}$, etc. As in the case for L^2-norms, we perform naive integration

$$E^n = h\sum_j \mathcal{H}_j^n.$$

To quantify the conservation of energy, we define the *relative error*,

$$f_r(t) = \frac{E(t) - E(0)}{E(0)}. \tag{10}$$

This is implemented in the programs presented in appendix D, written in Python 3.3. Figure 4 shows this quantity as a function of time for the problem posed in sections 1.1 and 1.4.

For both boundary conditions, we see that the solution is bounded above and below. There are peaks in the relative error for both cases when the wave is interacting with the boundary points, and the amplitude of these peaks is slightly over double when Neumann boundary conditions are used. The peaks are bounded by a number that appears to scale linearly with h, in the sense that halving h halves the amplitude of these peaks, and so on. The lower continuum of the relative error appears to scale with h^{-2}, showing a different behaviour in the relative energy when the wave is interacting with a boundary.

For the majority of the interval $t \in [0, 14]$, we see that $f_r(t)$ is bounded below by a small number, which is h^{-2}. Given the h^{-2}, τ^{-2} dependence of $\mathcal{H}_j^n$ from our integration, this makes sense. As $t \to 14$ however, this bound is broken.

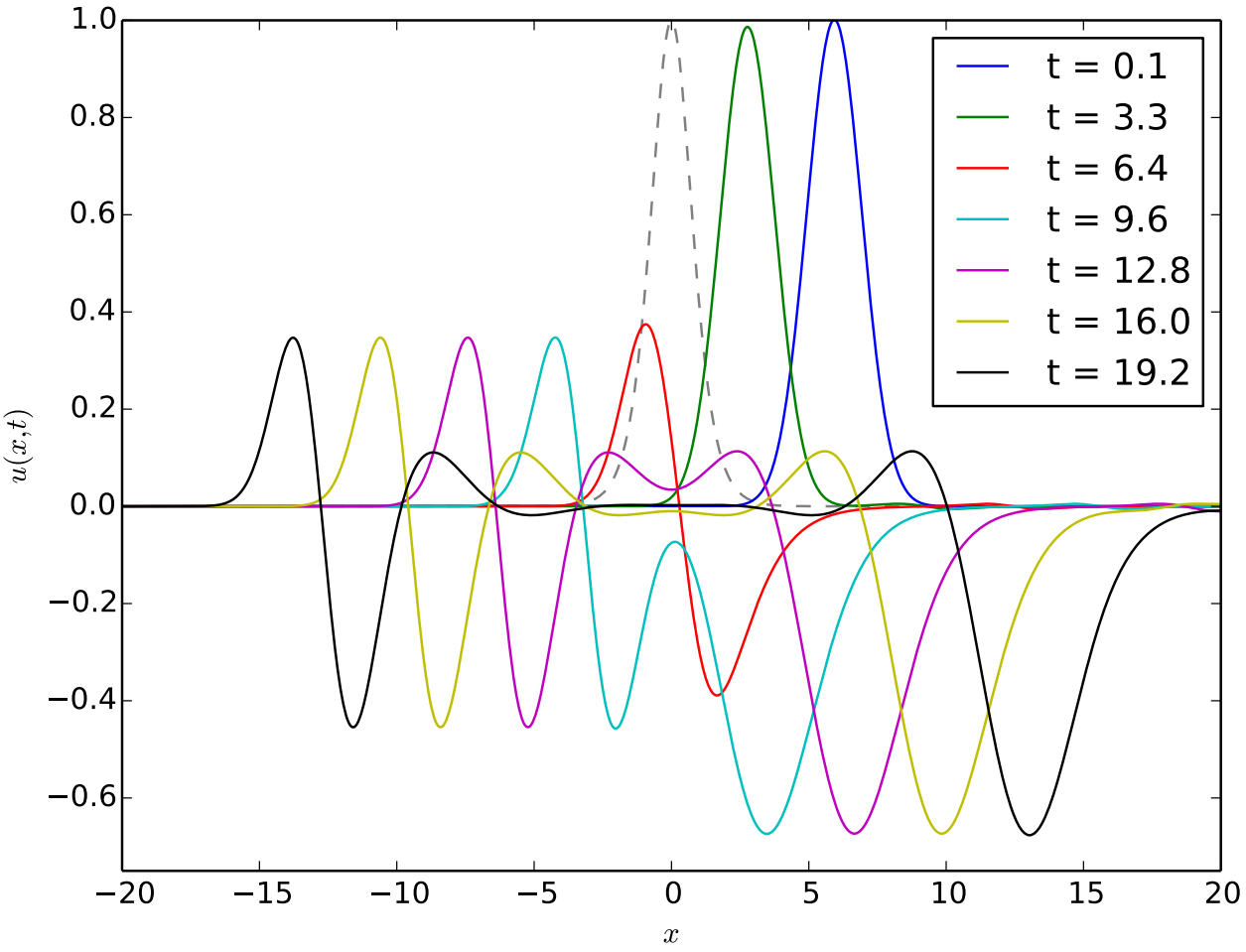

Figure 5: The wave at various times, with the potential plotted in the dashed grey line, $h = 0.05$.

Question 2

(1)

We are now asked to solve the wave equation with $c = 1$ and external potential $V(x) = \frac{1}{\cosh^2(x)}$. I chose to use central differences – equation 2 – once again as it is simple to program and showed no significant shortcomings in the previous section which cannot be fixed by computational power. By the same method as for question 1.1, we can use central differences for this equation. This gives

$$u_j^{n+1} = \nu^2 \left[u_{j-1}^n + u_{j+1}^n\right] - u_j^{n-1} + 2(1-\nu^2)u_j^n - \tau^2 V_j u_j^n \tag{11}$$

where V_j is the potential corresponding to the x-value j. The effect of V can be seen to scale with the time step τ, which is intuitive.

The program was written in Python 3 again, and is in appendix E. A plot of the waves behaviour is shown in figure 5. After travelling left to the potential barrier, the wave interacts with the barrier, and some is transmitted, while more is reflected.

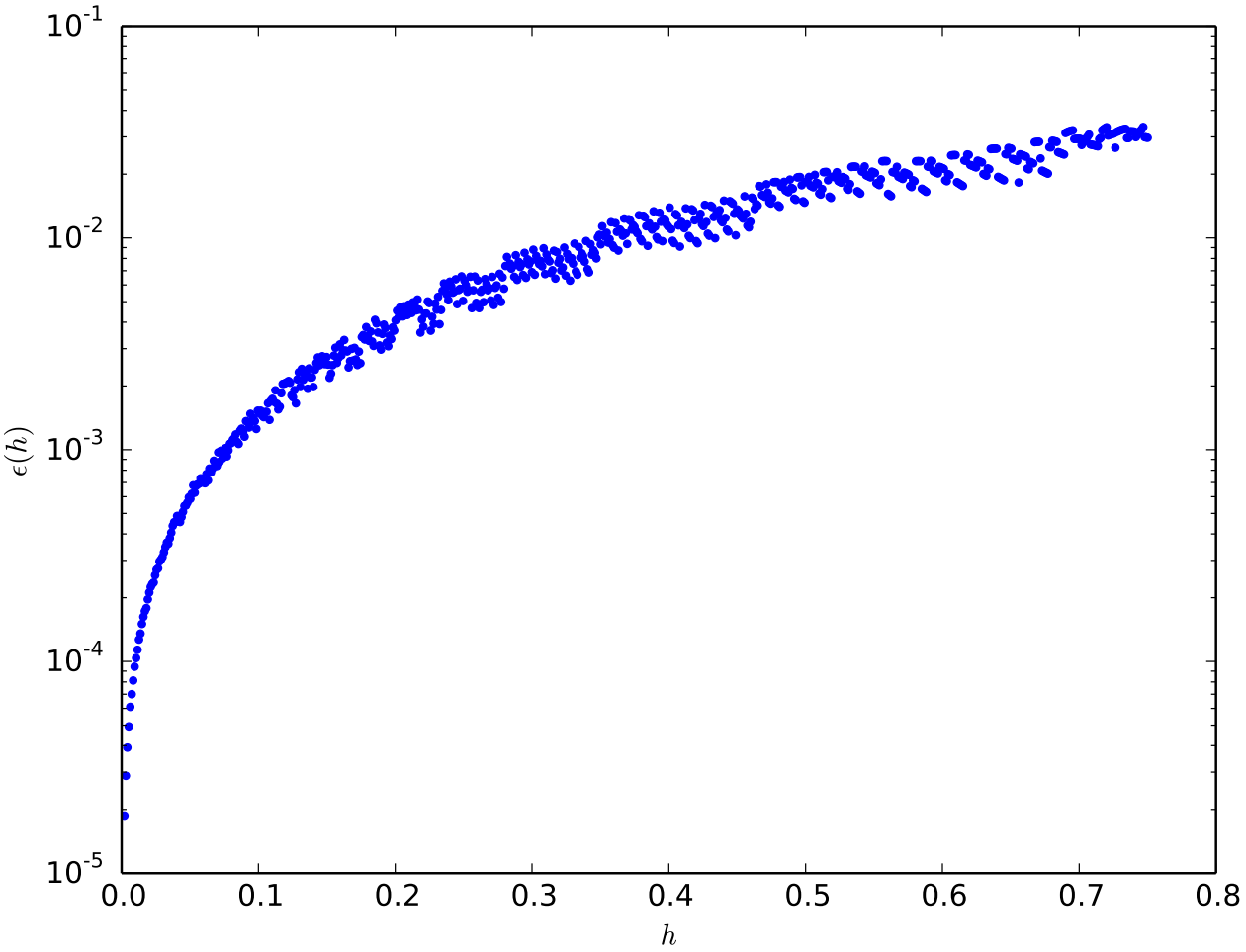

Figure 6: Relative error in L^2-norms for varios values of h.

(2)

To validate our choice of h in the previous section, we perform another study using the relative error defined in equation 6. The program which does this operates in the same manner as the previous resolution study, and is presented in appendix F. Figure 6 shows the relative error as a function of h, showing an optimal value for $h \sim 0.075$. This value is however lowered in the next section.

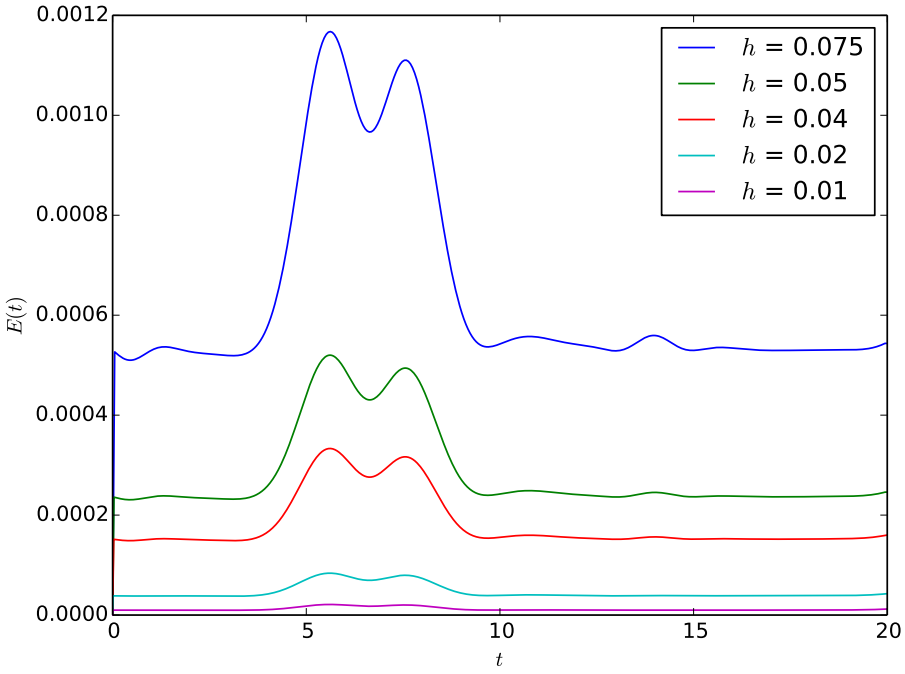

Figure 7: Relative error in energy as a function of time.

(3)

Analytical Energy Conservation

We define our Hamiltonian density as in section 1.5, but with a term for the external potential, giving

$$\mathcal{H}(x,t) = \frac{1}{2}\left(\frac{\partial u}{\partial t}\right)^2 + \frac{1}{2}\left(\frac{\partial u}{\partial x}\right)^2 + \frac{1}{2}V(x)u(x,t)^2, \tag{12}$$

To show that energy is conserved, we consider the time derivative of the energy and show it will be zero.

$$\frac{d}{dt}\int_{x_{\min}}^{x_{\max}} \mathcal{H}(x,t)dx = \int_{x_{\min}}^{x_{\max}} \frac{\partial}{\partial t}\mathcal{H}(x,t)dx =$$

$$\int_{x_{\min}}^{x_{\max}} \frac{\partial}{\partial t}\left[\frac{1}{2}\left(\frac{\partial u}{\partial t}\right)^2 + \frac{1}{2}\left(\frac{\partial u}{\partial x}\right)^2 + \frac{1}{2}V(x)u(x,t)^2\right]dx \tag{13}$$

The wave eqution can be rearranged to give an expression for the time derivatives, remembering $c = 1$,

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} - V(x)u(x,t)$$

and substituting this into equation 14 gives

(14)

Numerical Results

We now compute the total energy of the wave as a function of time. As in section 1.5. Taking our expression for $\mathcal{H}$ from equation 12, we obtain an expression for the numerical $\mathcal{H}_j^n$

$$\mathcal{H}_j^n = \frac{1}{2}\left[\frac{1}{2\tau}\left(u_j^{n+1} - u_j^{n-1}\right)\right]^2 + \frac{1}{2}\left[\frac{1}{2h}\left(u_{j+1}^n - u_{j-1}^n\right)\right]^2 + \frac{1}{2}V_j\left(u_j^n\right)^2.$$

This is integrated as before, and the results for various values of h are shown in figure 7.

We see large peaks in the error during the interaction of the wave with the potential, after which it smooths out. Again, the baseline relative error, and peak height, seem to scale as h^{-2}. In the previous section, 2.2, our optimal value for h is ~ 0.075. This value however gives peaks in the relative energy here greater than 10^{-3}, so our optimal value for h is now taken to be 0.05, as this divides our range evenly and provides relative errors for the L^2-norm and energy better than 10^{-3}.

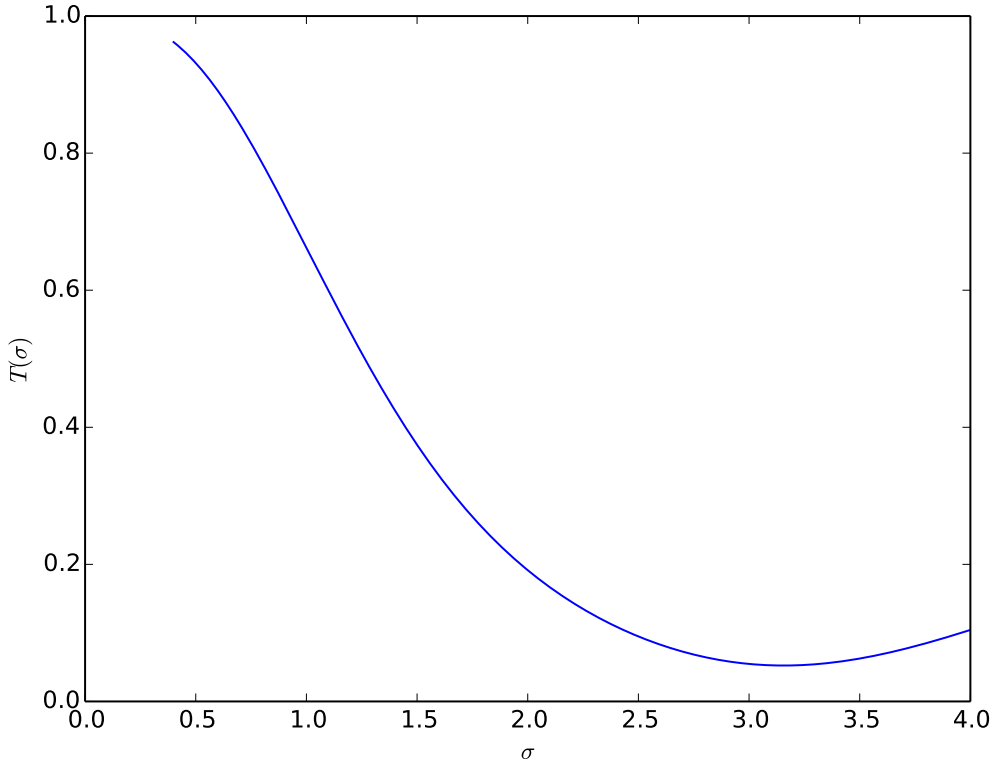

Figure 8: Dependence of T on σ.

(4)

We are now interested in quantifying how much of the pulse is transmitted through the potential barrier. We define

$$T_\sigma(t_{\max}) = \frac{E_<(t_{\max}) - E(t_{\max})}{E(t_{\max})} \tag{15}$$

as the *transmission coefficient* for a given σ, where $E_<(t)$ is the energy of the wave left of $x = 0$, i.e. the transmitted wave. This transmission coefficient, $T \in [0, 1]$, provides a measure of how much of the pulse is transmitted for a given $t_{\max}$ and σ. $t_{\max}$ is chosen so that the wave is sufficiently far from both the potential barrier and the boundaries to be interacting with neither, and in this case is taken as $t_{\max} = 18$.

Figure 8 shows the dependence of T on σ. There is a clear increase in T as σ decreases, below $\sigma \sim 3$. Above ~ 3, the evaluation of T becomes unrepresentative of what is happening, as the initial Gaussian is spread very wide, with a significant portion already beyond the barrier, as is seen in figure 9, with $\sigma = 4$. This can be rectified by changing the value of μ in the Gaussian to start the wave further away, but the range of σ values presented here is sufficient to quantify the behaviour of the system.

Our initial condition, $\frac{\partial u}{\partial t}(x, 0) = \frac{\partial u}{\partial x}(x, 0)$, is largely responsible for the effect of σ on T. Since we know

$$u(x, 0) = e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

we can analytically compute $\frac{\partial u}{\partial t}(x, 0)$, the initial velocity of the wave, to be

$$\frac{\partial u}{\partial t}(x, 0) = -\frac{(x - \mu)}{\sigma^2} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Clearly, the initial velocity of the wave is increased as σ is decreased, resulting in a more energetic wave better able to overcome the barrier.

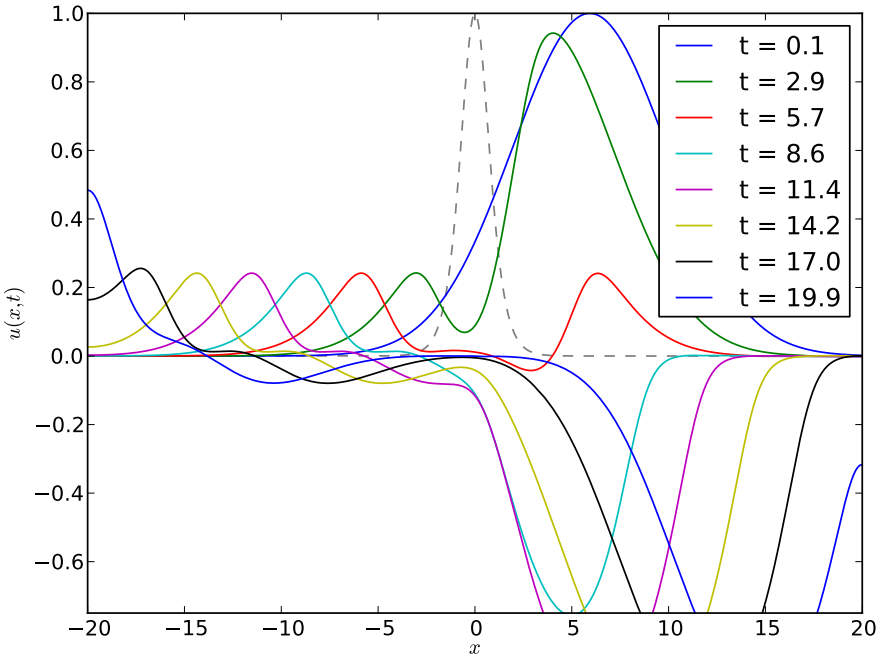

Figure 9: Plot of the system for $\sigma = 4$.

A Program for 1.1

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 using central differences.
# Dirichlet boundaries are used, and imposed implicitly, by allocating an array
# of zeros and not writing to the edge points.
# Liam O'Sullivan

from matplotlib.pyplot import *
from numpy import *

# Define our space and time step sizes.
dx = 0.1
dt = dx/sqrt(2)

# Define our Temporal and Spatial lengths.
Time = [0,14]
Dist = [-7,7]

# Define variables with the number of space/time steps for ease later.
# They must be cast as ints for use with range().
nt = int((Time[1] - Time[0])/dt) + 1
nx = int((Dist[1] - Dist[0])/dx) + 1

# Define our initial arrays.
U = zeros((nt, nx), dtype=float64)
t = zeros(nt, dtype=float64)
x = zeros(nx, dtype=float64)
# Initialise the tand x arrays.
for n in range(0,nt):
    t[n] = Time[0] + n*dt
for j in range(0,nx):
    x[j] = Dist[0] + j*dx

# Initialise the t_0 part of the array.
for j in range(1,nx-1):
```

```
    U[0][j] = e**(-x[j]*x[j])
plot(x,U[0],label="t = 0")

# c is not c from the wave equation:
c = (dt/dx)**2

# Since U_j^n-1 is unknown, I take is equivalent to U_j^n for the first run.
for j in range(1,nx-1):
    U[1][j] = c*(U[0][j-1] - 2*U[0][j] + U[0][j+1]) + U[0][j]

# Now perform our integration.
for n in range(2,nt):
    for j in range(1,nx-1):
        U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j]
    if ((n+1)%33) is 0:
        plot(x,U[n],label="t = "+str(round(14*(n/nt),1)))
xlabel("$x$")
ylabel("$u(x,t)$")
xlim([-7,7])
ylim([-1,1])
legend()
savefig("dirichlet.pdf",format="pdf")
show()
```

B Program for 1.3

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 using central differences.
# Dirichlet boundaries are used, and imposed implicitly, by allocating an array
# of zeros and not writing to the edge points. A variety of h values are used.
# Liam O'Sullivan
from matplotlib.pyplot import *
from numpy import *

def run(dx,T,D):
    # Define variables with the number of space/time steps for ease later.
    # They must be cast as ints for use with range(), but should be integers
    # anyway, since I'll only be setting dx as a nice value, I swear...
    dt = dx/sqrt(2)
    nt = int((T[1] - T[0])/dt)
    nx = int((D[1] - D[0])/dx)
    # Define our initial arrays.
    U = zeros((nt, nx), dtype=float64)
    t = zeros(nt, dtype=float64)
    x = zeros(nx, dtype=float64)
    # Initialise the tand x arrays.
    for n in range(0,nt):
        t[n] = Time[0] + n*dt
    for j in range(0,nx):
        x[j] = Dist[0] + j*dx

    # Initialise the t_0 part of the array.
    for j in range(1,nx-1):
        U[0][j] = e**(-x[j]*x[j])

    # c is not c from the wave equation, just a useful constant.
    c = (dt/dx)**2

    # Since U_j^n-1 is unknown, I take is equivalent to U_j^n for the first run.
    for j in range(1,nx-1):
        U[1][j] = c*(U[0][j-1] - 2*U[0][j] + U[0][j+1]) + U[0][j]

    # Now perform our norm computation.
    for n in range(2,nt):
        for j in range(1,nx-1):
            U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j]

    Ltime = nt*10.5/(T[1] - T[0])
```

```
    norm = 0
    for k in U[Ltime]:
        norm += k**2
    norm = sqrt(dx*norm)
    return norm


# Define our Temporal and Spatial lengths.
Time = [0,14]
Dist = [-7,7]

# Define our space and time step sizes.
dxs = [(0.001 + o/2500) for o in range(0,2000)]
err = []

for i in range(0,len(dxs)):
    Lfull = run(dxs[i],Time,Dist)
    Lhalf = run(dxs[i]/2,Time,Dist)
    err.append(1 - (Lhalf/Lfull))
    print("h = "+str(round(dxs[i],4))+", err = "+str(err[-1]))

xlabel("$h$")
ylabel("$\epsilon(h)$")
semilogy(dxs,err,'.')
savefig("Error.pdf",format='PDF')
show()
```

C Program for 1.4

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 using central differences.
# Neumann boundaries are used, and imposed by eqating the endpoints to one in.
# of zeros and not writing to the edge points.
# Liam O'Sullivan

from matplotlib.pyplot import *
from numpy import *

# Define our space and time step sizes.
dx = 0.1
dt = dx/sqrt(2)

# Define our Temporal and Spatial lengths.
Time = [0,14]
Dist = [-7,7]

# Define variables with the number of space/time steps for ease later.
# They must be cast as ints for use with range(), but should be integers
# anyway, since I'll only be setting dx as a nice value, I swear...
nt = int((Time[1] - Time[0])/dt) + 1
nx = int((Dist[1] - Dist[0])/dx) + 1

# Define our initial arrays.
U = zeros((nt, nx), dtype=float64)
t = zeros(nt, dtype=float64)
x = zeros(nx, dtype=float64)
# Initialise the tand x arrays.
for n in range(0,nt):
    t[n] = Time[0] + n*dt
for j in range(0,nx):
    x[j] = Dist[0] + j*dx

# Initialise the t_0 part of the array.
for j in range(1,nx-1):
    U[0][j] = e**(-x[j]*x[j])

# c is not c from the wave equation:
c = (dt/dx)**2
```

```
# Since U_j^n-1 is unknown, I take is equivalent to U_j^n for the first run.
for j in range(1,nx-1):
    U[1][j] = c*(U[0][j-1] - 2*U[0][j] + U[0][j+1]) + U[0][j]
U[1][0] = U[1][1]
U[1][nx-1] = U[1][nx-2]

# Now perform our integration.
for n in range(2,nt):
    for j in range(1,nx-1):
        U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j]
    U[n][0] = U[n][1]
    U[n][nx-1] = U[n][nx-2]
    if ((n-2)%42) is 0:
        plot(x,U[n],label="t = "+str(round(14*(n/nt),1)))
xlabel("$x$")
ylabel("$u(x,t)$")
xlim([-7,7])
legend()
savefig("neumann.pdf",format="pdf")
show()
```

D Programs for 1.5

Dirichlet Boundaries:

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 using central differences.
# Dirichlet boundaries are used, and imposed implicitly, by allocating an array
# of zeros and not writing to the edge points.
# This program proceeds to caltulate the energy as a function of time and h.
# Liam O'Sullivan

from matplotlib.pyplot import *
from numpy import *

def run(dx,Time,Dist):
    # Define our space and time step sizes.
    dt = dx/sqrt(2)

    # Define variables with the number of space/time steps for ease later.
    # They must be cast as ints for use with range().
    nt = int((Time[1] - Time[0])/dt) + 1
    nx = int((Dist[1] - Dist[0])/dx) + 1

    # Define our initial arrays.
    U = zeros((nt, nx), dtype=float64)
    H = zeros((nt, nx), dtype=float64)
    E = zeros(nt, dtype=float64)
    t = zeros(nt, dtype=float64)
    x = zeros(nx, dtype=float64)
    # Initialise the tand x arrays.
    for n in range(0,nt):
        t[n] = Time[0] + n*dt
    for j in range(0,nx):
        x[j] = Dist[0] + j*dx

    # Initialise the t_0 part of the array.
    for j in range(1,nx-1):
        U[0][j] = e**(-x[j]*x[j])

    # c is not c from the wave equation:
    c = (dt/dx)**2

    # Since U_j^n-1 is unknown, I take is equivalent to U_j^n for the first run.
    for j in range(1,nx-1):
        U[1][j] = c*(U[0][j-1] - 2*U[0][j] + U[0][j+1]) + U[0][j]

    # Now perform our integration.
```

```
    for n in range(2,nt):
        for j in range(1,nx-1):
            U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j]

    # From here we compute H(x,t). The boundary terms are messy, but most
    # of this corresponds exactly to the analytical form.
    # Sort out the t = 0 boundary.
    H[0][0] = 0.5*(((1/dt)*(U[1][0] - U[0][0]))**2) + 0.5*(((1/dx)*(U[0][1] - U[0][0]))**2)
    H[0][nx-1] = 0.5*(((1/dt)*(U[1][nx-1] - U[0][nx-1]))**2) + 0.5*(((1/dx)*(U[0][nx-1] - U[0][nx-2]))
        **2)
    for j in range(1,nx-1):
        H[0][j] = 0.5*(((1/(dt))*(U[1][j]-U[0][j]))**2) + 0.5*(((1/(2*dx))*(U[0][j+1]-U[0][j-1]))**2)
    E[0] += dx*sum(H[0])
    # Now start compuating H more generally.
    for n in range(1,nt-1):
        # Sort out the j boundaries.
        H[n][0] = 0.5*(((1/(2*dt))*(U[n+1][0] - U[n-1][0]))**2) + 0.5*(((1/dx)*(U[n][1] - U[n][0]))**2)
        H[n][nx-1] = 0.5*(((1/(2*dt))*(U[n+1][nx-1]-U[n-1][nx-1]))**2)+0.5*(((1/dx)*(U[n][nx-1]-U[n][nx
            -2]))**2)
        for j in range(1,nx-1):
            H[n][j] = 0.5*(((1/(2*dt))*(U[n+1][j]-U[n-1][j]))**2) + 0.5*(((1/(2*dx))*(U[n][j+1]-U[n][j
                -1]))**2)
        E[n] += dx*sum(H[n])
    # Sort the t_max boundaries.
    H[nt-1][0] = 0.5*(((1/dt)*(U[nt-1][0] - U[nt-2][0]))**2) + 0.5*(((1/dx)*(U[nt-1][1] - U[nt-1][0]))
        **2)
    H[nt-1][nx-1]=0.5*(((1/dt)*(U[nt-1][nx-1]-U[nt-2][nx-1]))**2)+0.5*(((1/dx)*(U[nt-1][nx-1]-U[nt-1][nx
        -2]))**2)
    for j in range(1,nx-1):
        H[nt-1][j]=0.5*(((1/(dt))*(U[nt-1][j]-U[nt-1][j]))**2)+0.5*(((1/(2*dx))*(U[nt-1][j+1]-U[nt-1][j
            -1]))**2)
    E[nt-1] += dx*sum(H[nt-1])

    # F is the relative error of the energy.
    F = array([(e/E[0] - 1) for e in E])
    F[-1] = F[-2] # Due to the boundaries the final term is not well constrained.
    return array([t,F])

# Try four different values of h.
a1 = run(0.1,[0,14],[-7,7])
a2 = run(0.05,[0,14],[-7,7])
a3 = run(0.025,[0,14],[-7,7])
a4 = run(0.0125,[0,14],[-7,7])

# Now we plot everything.
plot(a1[0],a1[1],'-',label="$h$ = 0.1")
plot(a2[0],a2[1],'-',label="$h$ = 0.05")
plot(a3[0],a3[1],'-',label="$h$ = 0.025")
plot(a4[0],a4[1],'-',label="$h$ = 0.0125")
xlabel("$t$")
ylabel("$E(t)$")
legend()
title("Relative error of energy for Dirichlet boundaries")
savefig("Ed.pdf",format="pdf")
show()
```

Neumann Boundaries:

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 using central differences.
# Neumann boundaries are used, and imposed explicitly.
# This program proceeds to caltulate the energy as a function of time and h.
# Liam O'Sullivan

from matplotlib.pyplot import *
from numpy import *

def run(dx,Time,Dist):
    # Define our space and time step sizes.
    dt = dx/sqrt(2)

    # Define variables with the number of space/time steps for ease later.
```

```
    # They must be cast as ints for use with range().
    nt = int((Time[1] - Time[0])/dt) + 1
    nx = int((Dist[1] - Dist[0])/dx) + 1

    # Define our initial arrays.
    U = zeros((nt, nx), dtype=float64)
    H = zeros((nt, nx), dtype=float64)
    E = zeros(nt, dtype=float64)
    t = zeros(nt, dtype=float64)
    x = zeros(nx, dtype=float64)
    # Initialise the tand x arrays.
    for n in range(0,nt):
        t[n] = Time[0] + n*dt
    for j in range(0,nx):
        x[j] = Dist[0] + j*dx

    # Initialise the t_0 part of the array.
    for j in range(1,nx-1):
        U[0][j] = e**(-x[j]*x[j])

    # c is not c from the wave equation:
    c = (dt/dx)**2

    # Since U_j^n-1 is unknown, I take is equivalent to U_j^n for the first run.
    for j in range(1,nx-1):
        U[1][j] = c*(U[0][j-1] - 2*U[0][j] + U[0][j+1]) + U[0][j]
    U[1][0] = U[1][1]
    U[1][nx-1] = U[1][nx-2]

    # Now perform our integration.
    for n in range(2,nt):
        for j in range(1,nx-1):
            U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j]
        U[n][0] = U[n][1]
        U[n][nx-1] = U[n][nx-2]

    # From here we compute H(x,t). The boundary terms are messy, but most
    # of this corresponds exactly to the analytical form.
    # Sort out the t = 0 boundary.
    H[0][0] = 0.5*(((1/dt)*(U[1][0] - U[0][0]))**2) + 0.5*(((1/dx)*(U[0][1] - U[0][0]))**2)
    H[0][nx-1] = 0.5*(((1/dt)*(U[1][nx-1] - U[0][nx-1]))**2) + 0.5*(((1/dx)*(U[0][nx-1] - U[0][nx-2]))
        **2)
    for j in range(1,nx-1):
        H[0][j] = 0.5*(((1/(dt))*(U[1][j]-U[0][j]))**2) + 0.5*(((1/(2*dx))*(U[0][j+1]-U[0][j-1]))**2)
    E[0] += dx*sum(H[0])
    # Now start compuating H more generally.
    for n in range(1,nt-1):
        # Sort out the j boundaries.
        H[n][0] = 0.5*(((1/(2*dt))*(U[n+1][0] - U[n-1][0]))**2) + 0.5*(((1/dx)*(U[n][1] - U[n][0]))**2)
        H[n][nx-1] = 0.5*(((1/(2*dt))*(U[n+1][nx-1]-U[n-1][nx-1]))**2)+0.5*(((1/dx)*(U[n][nx-1]-U[n][nx
            -2]))**2)
        for j in range(1,nx-1):
            H[n][j] = 0.5*(((1/(2*dt))*(U[n+1][j]-U[n-1][j]))**2) + 0.5*(((1/(2*dx))*(U[n][j+1]-U[n][j
                -1]))**2)
        E[n] += dx*sum(H[n])
    # Sort the t_max boundaries.
    H[nt-1][0] = 0.5*(((1/dt)*(U[nt-1][0] - U[nt-2][0]))**2) + 0.5*(((1/dx)*(U[nt-1][1] - U[nt-1][0]))
        **2)
    H[nt-1][nx-1]=0.5*(((1/dt)*(U[nt-1][nx-1]-U[nt-2][nx-1]))**2)+0.5*(((1/dx)*(U[nt-1][nx-1]-U[nt-1][nx
        -2]))**2)
    for j in range(1,nx-1):
        H[nt-1][j]=0.5*(((1/(dt))*(U[nt-1][j]-U[nt-1][j]))**2)+0.5*(((1/(2*dx))*(U[nt-1][j+1]-U[nt-1][j
            -1]))**2)
    E[nt-1] += dx*sum(H[nt-1])

    # F is the relative error of energy, as defined in the report.
    F = array([(e/E[0] - 1) for e in E])
    F[-1] = F[-2] # Due to the boundaries the final term is not well constrained.
    return array([t,F])

# Perform four runs of the above function, to display for various h.
```

```
a1 = run(0.1,[0,14],[-7,7])
a2 = run(0.05,[0,14],[-7,7])
a3 = run(0.025,[0,14],[-7,7])
a4 = run(0.0125,[0,14],[-7,7])

# Now plot everything.
plot(a1[0],a1[1],'-',label="$h$ = 0.1")
plot(a2[0],a2[1],'-',label="$h$ = 0.05")
plot(a3[0],a3[1],'-',label="$h$ = 0.025")
plot(a4[0],a4[1],'-',label="$h$ = 0.0125")
xlabel("$t$")
ylabel("$E(t)$")
legend()
title("Relative error of energy for Neumann boundaries")
savefig("En.pdf",format="pdf")
show()
```

E Program for 2.1

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 and external potential using
# central differences. Neumann boundaries are used.
# Liam O'Sullivan
from matplotlib.pyplot import *
from numpy import *
#sigma for gaussian
sig = 1
# Define our space and time step sizes.
dx = 0.05
dt = dx/sqrt(2)
# Define our Temporal and Spatial lengths.
Time = [0,20]
Dist = [-20,20]

# Define variables with the number of space/time steps for ease later.
# They must be cast as ints for use with range().
nt = int((Time[1] - Time[0])/dt) + 1
nx = int((Dist[1] - Dist[0])/dx) + 1
# Define our initial arrays.
U = zeros((nt, nx), dtype=float64)
V = zeros(nx, dtype=float64)
t = zeros(nt, dtype=float64)
x = zeros(nx, dtype=float64)

# Initialise the tand x arrays.
for n in range(0,nt):
    t[n] = Time[0] + n*dt
for j in range(0,nx):
    x[j] = Dist[0] + j*dx
# Initialise the t_0 part of the array; Gaussian with sigma=1 and mu=6.
for j in range(1,nx-1):
    U[0][j] = e**(-((x[j]-6)**2)/(2*(sig**2)))
# Initialise our potential array.
for j in range(0,nx):
    V[j] = 1/(cosh(x[j])**2)
# Plot the potential with a broken line for visual purposes.
plot(x,V,'--',color='grey')
# c is not c from the wave equation, but nu^2.
c = (dt/dx)**2
# I manually impose the initial derivative condition, by computing the second
# time step as d/dx of the analytic u_0, using the Euler method(Forgive me).
for j in range(1,nx-1):
    U[1][j] = U[0][j] + dt*(-((x[j]-6)/(sig**2))*(e**(-((x[j]-6)**2)/(2*(sig**2)))))

# Now perform our integration.
for n in range(2,nt):
    for j in range(1,nx-1):
```

```
        U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j] - (dt**2)*V[j]*U[
            n-1][j]
        # Impose the Neumann condition here:
    U[n][0] = U[n][1]
    U[n][nx-1] = U[n][nx-2]
    if ((n-2)%90) is 0:
        plot(x,U[n],label="t = "+str(round(Time[1]*(n/nt),1)))
xlabel("$x$")
ylabel("$u(x,t)$")
xlim([-20,20])
ylim([-0.75,1])
legend()
savefig("cosh.pdf",format="pdf")
show()
```

F Program for 2.2

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 and external potential using
# central differences. Neumann boundaries are used.
# Liam O'Sullivan
from matplotlib.pyplot import *
from numpy import *

def run(dx,Time,Dist):
    dt = dx/sqrt(2)
    # Define variables with the number of space/time steps for ease later.
    # They must be cast as ints for use with range().
    nt = int((Time[1] - Time[0])/dt)
    nx = int((Dist[1] - Dist[0])/dx)
    # Define our initial arrays.
    U = zeros((nt, nx), dtype=float64)
    V = zeros(nx, dtype=float64)
    t = zeros(nt, dtype=float64)
    x = zeros(nx, dtype=float64)
    # Initialise the tand x arrays.
    for n in range(0,nt):
        t[n] = Time[0] + n*dt
    for j in range(0,nx):
        x[j] = Dist[0] + j*dx
    # Initialise the t_0 part of the array; Gaussian with sigma=1 and mu=6.
    for j in range(1,nx-1):
        U[0][j] = e**(-((x[j]-6)**2)/2)
    # Initialise our potential array.
    for j in range(0,nx):
        V[j] = 1/(cosh(x[j])**2)

    # c is not c from the wave equation, but nu^2.
    c = (dt/dx)**2
    # I manually impose the initial derivative condition, by computing the second
    # time step as d/dx of the analytic u_0, using the Euler method(Forgive me).
    for j in range(1,nx-1):
        U[1][j] = U[0][j] + dt*(-(x[j]-6)*(e**(-((x[j]-6)**2)/2)))
        U[1][0] = U[1][1]
        U[1][nx-1] = U[1][nx-2]
    # Now perform our integration.
    for n in range(2,nt):
        for j in range(1,nx-1):
            U[n][j] = c*(U[n-1][j-1]-2*U[n-1][j]+U[n-1][j+1])-U[n-2][j]+2*U[n-1][j]-(dt**2)*V[j]*U[n-1][j
                ]
        U[n][0] = U[n][1]
        U[n][nx-1] = U[n][nx-2]
    Ltime = 18.*nt/(Time[1] - Time[0]) # The time t_{max}.
    norm = 0
    for b in U[Ltime]:
        norm += b**2
    norm = sqrt(dx*norm)
    return norm
```

```
# Define our space and time step sizes.
dxs = linspace(0.750,0.002,699)
# Define our Temporal and Spatial lengths.
aTime = [0,19]
aDist = [-20,20]
hs = []
for i in range(0,len(dxs)):
    n1 = run(dxs[i],aTime,aDist)
    n2 = run(dxs[i]/2.,aTime,aDist)
    hs.append(1 - n2/n1)
    print("h = "+str(round(dxs[i],4))+", err = "+str(hs[-1]))
semilogy(dxs,hs,'.')
xlabel("$h$")
ylabel("$\epsilon(h)$")
savefig("epsilon.pdf",format="pdf")
show()
```

G Program for 2.3

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 and external potential using
# central differences. Neumann boundaries are used.
# Liam O'Sullivan
from matplotlib.pyplot import *
from numpy import *
#sigma for gaussian
sig = 1
# Define our space and time step sizes.
dx = 0.05
dt = dx/sqrt(2)
# Define our Temporal and Spatial lengths.
Time = [0,20]
Dist = [-20,20]

# Define variables with the number of space/time steps for ease later.
# They must be cast as ints for use with range().
nt = int((Time[1] - Time[0])/dt) + 1
nx = int((Dist[1] - Dist[0])/dx) + 1
# Define our initial arrays.
U = zeros((nt, nx), dtype=float64)
V = zeros(nx, dtype=float64)
t = zeros(nt, dtype=float64)
x = zeros(nx, dtype=float64)

# Initialise the tand x arrays.
for n in range(0,nt):
    t[n] = Time[0] + n*dt
for j in range(0,nx):
    x[j] = Dist[0] + j*dx
# Initialise the t_0 part of the array; Gaussian with sigma=1 and mu=6.
for j in range(1,nx-1):
    U[0][j] = e**(-((x[j]-6)**2)/(2*(sig**2)))
# Initialise our potential array.
for j in range(0,nx):
    V[j] = 1/(cosh(x[j])**2)
# Plot the potential with a broken line for visual purposes.
plot(x,V,'--',color='grey')
# c is not c from the wave equation, but nu^2.
c = (dt/dx)**2
# I manually impose the initial derivative condition, by computing the second
# time step as d/dx of the analytic u_0, using the Euler method(Forgive me).
for j in range(1,nx-1):
    U[1][j] = U[0][j] + dt*(-((x[j]-6)/(sig**2))*(e**(-((x[j]-6)**2)/(2*(sig**2)))))

# Now perform our integration.
for n in range(2,nt):
    for j in range(1,nx-1):
```

```
        U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j] - (dt**2)*V[j]*U[
            n-1][j]
        # Impose the Neumann condition here:
    U[n][0] = U[n][1]
    U[n][nx-1] = U[n][nx-2]
    if ((n-2)%90) is 0:
        plot(x,U[n],label="t = "+str(round(Time[1]*(n/nt),1)))
xlabel("$x$")
ylabel("$u(x,t)$")
xlim([-20,20])
ylim([-0.75,1])
legend()
savefig("cosh.pdf",format="pdf")
show()
```

H Program for 2.4

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 and external potential using
# central differences. Neumann boundaries are used.
# This program computes the ratio of transmitted to reflected wave.
# Liam O'Sullivan
from matplotlib.pyplot import *
from numpy import *

def run(sig,dx,Time,Dist):
    # Define our space and time step sizes.
    dt = dx/sqrt(2)
    # Define variables with the number of space/time steps for ease later.
    # They must be cast as ints for use with range().
    nt = int((Time[1] - Time[0])/dt) + 1
    nx = int((Dist[1] - Dist[0])/dx) + 1
    # Define our initial arrays.
    U = zeros((nt, nx), dtype=float64)
    H = zeros((nt, nx), dtype=float64)
    V = zeros(nx, dtype=float64)
    t = zeros(nt, dtype=float64)
    x = zeros(nx, dtype=float64)

    # Initialise the tand x arrays.
    for n in range(0,nt):
        t[n] = Time[0] + n*dt
    for j in range(0,nx):
        x[j] = Dist[0] + j*dx
    # Initialise the t_0 part of the array; Gaussian with sigma=sig and mu=6.
    for j in range(1,nx-1):
        U[0][j] = e**(-((x[j]-6)**2)/(2*(sig**2)))
    # Initialise our potential array.
    for j in range(0,nx):
        V[j] = 1/(cosh(x[j])**2)
    # c is not c from the wave equation, but nu^2.
    c = (dt/dx)**2
    # I manually impose the initial derivative condition, by computing the second
    # time step as d/dx of the analytic u_0, using the Euler method(Forgive me).
    for j in range(1,nx-1):
        U[1][j] = U[0][j] + dt*(-((x[j]-6)/(sig**2))*(e**(-((x[j]-6)**2)/(2*(sig**2)))))
    # Now perform our integration.
    for n in range(2,nt):
        for j in range(1,nx-1):
            U[n][j] = c*(U[n-1][j-1]-2*U[n-1][j]+U[n-1][j+1])-U[n-2][j]+2*U[n-1][j]-(dt**2)*V[j]*U[n-1][j
                ]
            # Impose the Neumann condition here:
        U[n][0] = U[n][1]
        U[n][nx-1] = U[n][nx-2]
    # No need to worry about t=0,tmax boundaries, start compuating H generally.
    for n in range(1,nt-1):
        # Sort out the j boundaries, ignore potential as it's really small.
        H[n][0] = 0.5*(((1/(2*dt))*(U[n+1][0] - U[n-1][0]))**2) + 0.5*(((1/dx)*(U[n][1] - U[n][0]))**2)
```

```
        H[n][nx-1] = 0.5*(((1/(2*dt))*(U[n+1][nx-1]-U[n-1][nx-1]))**2)+0.5*(((1/dx)*(U[n][nx-1]-U[n][nx
            -2]))**2)
        for j in range(1,nx-1):
            H[n][j] = 0.5*(((1/(2*dt))*(U[n+1][j]-U[n-1][j]))**2) + 0.5*(((1/(2*dx))*(U[n][j+1]-U[n][j
                -1]))**2) + 0.5*V[j]*(U[n][j]**2)
    Tmax = 18.*nt/(Time[1] - Time[0]) # Find the index for t_max.
    jmid = int(nx/2)
    T = dx*sum(H[Tmax][0:jmid])
    E = dx*sum(H[Tmax])
    return T/E

aTime = [0,19]
aDist = [-20,20]
sigmas = []
Ts = []
for s in linspace(4,0.4,200):
     sigmas.append(s)
     Ts.append(run(s,0.05,aTime,aDist))
     print("Sigma = "+str(round(s,5))+", T = "+str(round(Ts[-1],5)))

plot(sigmas,Ts,'-')
ylabel("$T(\sigma)$")
xlabel("$\sigma$")
savefig("sigma.pdf",format="pdf")
show()
```