Project 2
Numerical methods for Differential Equations

Daniel Holm, Erik Olsson

November 2019

1 Two point boundary value problems

The differential equation of interest here is the two point boundary value problem (2pBVP)

$$\begin{aligned} &y'' = f(x,\, y), \\ &y(0) = \alpha,\ y(L) = \beta. \end{aligned} \tag{1}$$

The problem can be discretized by first introducing an equidistant grid of N interior points along $[0,\, L]$ with $\Delta x = \frac{L}{N+1}$ and then approximating the second derivative with a symmetric finite difference quotient.

$$\begin{aligned} &\frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} = f(x_i,\, y_i), \\ &y_0 = \alpha,\ y_{N+1} = \beta. \end{aligned} \tag{2}$$

This will give an equation system of N equations for the N unknowns $y_1, y_2, ..., y_N$ that approximate the true values $y(x_1), y(x_2), ..., y(x_N)$. For our purposes we will only consider functions of the type $f(x,\, y) \equiv f(x)$, which correspond to simple linear problems where the right hand side is independent of y. Thus the equation system can be expressed as a matrix equation

$$\frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & \dots & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & & \ddots & 1 \\ & \dots & 0 & 1 & -2 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \\ y_N \end{pmatrix} = \begin{pmatrix} -\alpha/\Delta x^2 + f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{N-1}) \\ -\beta/\Delta x^2 + f(x_N) \end{pmatrix}. \tag{3}$$

Because the matrix is sparse the system has much lower complexity compared to a non-sparse matrix. This allows for more efficient computing methods.

1.1 Writing the 2pBVP solver

First, we simply created a vector, $fvec$, consisting of the values $f(x_1), f(x_2), ..., f(x_N)$. Then we fixed the boundaries according to right hand side of (3). Lastly we solved the equation for y and added the boundary points y_0, y_{N+1}. To verify we tested $f(x) = -\sin x$, with $\alpha = \beta = 0$ and $L = \pi$, where we know that $y = \sin x$. The resulting approximation can be seen in Figure 1.

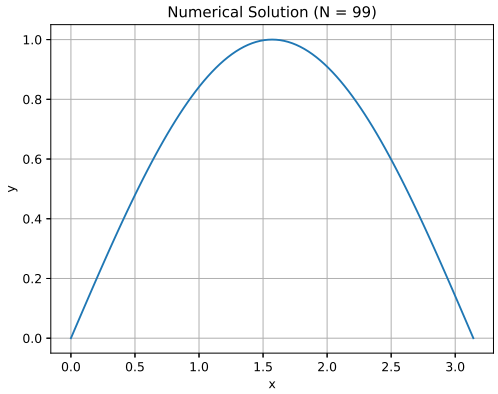

Figure 1: Approximation of y, with $f(x) = -\sin x$ and $\alpha = \beta = 0$

Because we know y, we can calculate the error, and investigate how the error is dependent on the stepsize.

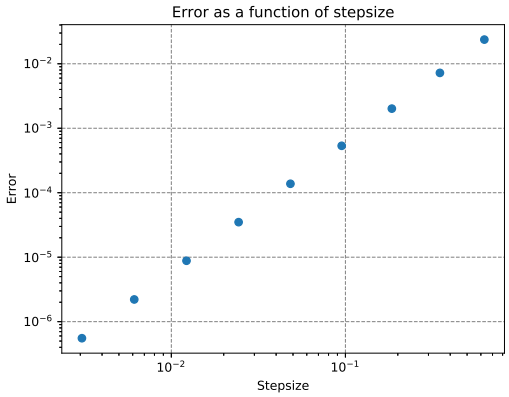

Figure 2: Error as a function of Stepsize, for $f(x) = -\sin x$

We can see in Figure 2 that the error is indeed $O(h^2)$, which is to be expected as the solver is a 2nd order method.

1.2 The Beam Equation

A force is applied to an elastic beam, and to calculate the deflection, we use the following model:

$$\begin{aligned} M'' &= q(x) \\ u'' &= M(x)/(EI) \end{aligned} \tag{4}$$

In (4), u is the beam's deflection, M is the bending moment and q is the load density. E is a constant (Young's modulus of elasticity) and I is the beam's cross-section moment inertia. We know that

$u(0) = u(L) = 0$, because the beam is fixed at it's ends. We also know that $M(0) = M(L) = 0$ for the same reasons. Using this knowledge we can use the solver from the previous task to approximate M, and then u. With $E = 1.9 \cdot 10^8$, $I(x) = 10^{-3} \cdot (3 - 2\cos^{12}\frac{\pi x}{L})$ and $q(x) = -50 \cdot 10^3$, u is approximated in Figure 3. The deflection of the beam's midpoint is -0.011741059 meters.

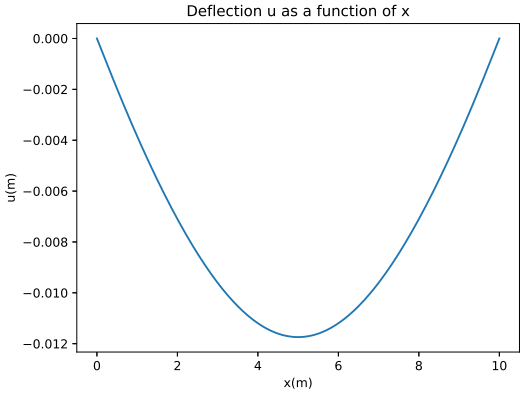

Figure 3: Deflection of a 10 m long construction steel beam, when applied with 5 metric tonnes per meter

2 Sturm-Liouville eigenvalue problems

The most common way of expressing the eigenvalue problem for a Sturm-Liouville differential operator is

$$\begin{aligned} &\frac{d}{dx}\left(p(x)\frac{dy}{dx}\right) - q(x)y = \lambda y, \\ &y(a) = 0, \ y(b) = 0, \end{aligned} \tag{5}$$

where $p(x) > 0$ and $q(x) \geq 0$. The problem is to find eigenvalues λ and their corresponding eigenfunctions $y(x)$ or simply "modes" that satisfy the above equation (5).

The problem is discretized by standard symmetric second order difference quotients similiar to (2) to obtain the matrix eigenvalue problem

$$T_{\Delta x} y = \lambda_{\Delta x} y, \tag{6}$$

where $T_{\Delta x}$ is a symmetric tridiagonal matrix of size $N \times N$ that depends on p, q and Δx. The discretization will have exactly N real eigenvalues $\lambda_{\Delta x} = \lambda + O(\Delta x^2)$, due to its symmetry. Of course the analytical problem has an infinite sequence of real eigenvalues and orthogonal eigenfunctions because the operator is self-adjoint.

2.1 A simple problem

Consider the simple problem

$$u'' = \lambda u, \tag{7}$$

with boundary conditions $u(0) = u(1) = 0$. The analytical solutions to this are of course

$$u(x) = \sin\sqrt{-\lambda}x = \sin k\pi x, \;\; \lambda = -k^2\pi^2, \tag{8}$$

for positive integers k (note that the eigenvalues are negative).

In order to construct a numerical solver for this problem we discretize and obtain the matrix problem

$$\frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & \dots & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & & \ddots & 1 \\ & \dots & 0 & 1 & -2 \end{pmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \\ u_N \end{pmatrix} = \lambda_{\Delta x}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \\ u_N \end{pmatrix}, \tag{9}$$

(note that the matrix is the same as in (3)). We can now write a solver method `slv(N, k)` which given the number of interior grid points `N` returns the first `k` eigenvalues and their corresponding eigenvectors with the boundary values included for plotting purposes. The method was implemented by first calculating Δx, creating the $T_{\Delta x}$ matrix and then using the built-in `scipy.sparse.linalg.eigsh` command. The boundary values (both zero) are inserted and appended to all eigenvectors before returning the results.

The solver was then used to plot the error $\lambda_{\Delta x} - \lambda$ for the first three eigenvalues as a function of N using various choices of N.

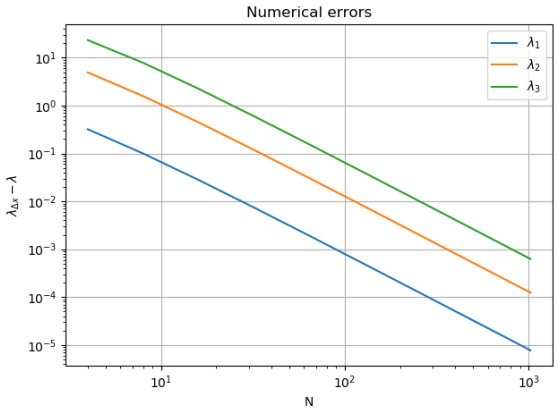

Figure 4: Numerical errors for the first three eigenvalues as a function of N

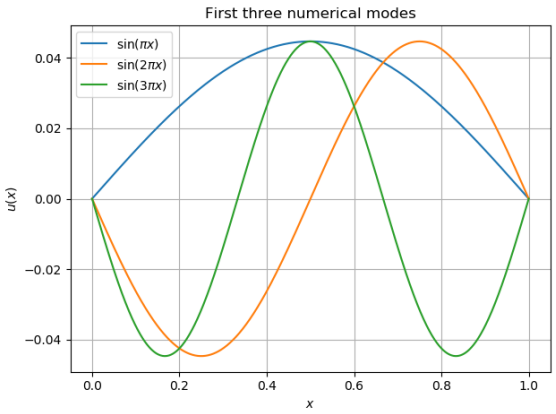

Figure 5: The numerical first three eigenmodes

As we can see in figure 4 the errors for all three eigenvalues is decreasing with logarithmic slope -2. This means that $\lambda_{\Delta x} = \lambda + O(\Delta x^2)$ as expected. In Figure 5 we see the first three numerically computed eigenmodes that do indeed look like the respective sin-waves they are supposed to

approximate.

A specific calculation for the first three eigenvalues where $N = 499$ was made by simply running the command `slv(499, 3)`. The results are presented in Table 1 below.

λ_1	λ_2	λ_3
-9.8695719	-39.477898	-88.823810

Table 1: Numerically computed eigenvalues to eight digits

2.2 The Schrödinger equation

A quantum particle trapped in a one-dimensional potential $V(x) \geq 0$ has a stationary wave function $\psi(x)$ that satisfies the Schrödinger equation

$$\psi'' - V(x)\psi = -E\psi, \tag{10}$$

where E is the energy level of the particle. If we assume the potential is infinite outside the interval [0, 1], the boundary conditions are $\psi(0) = \psi(1) = 0$. In this case it becomes a Sturm-Liouville problem for determining wave functions and energy levels. We will also study the corresponding probability densities $|\psi|^2$ which give the probability of finding the particle at a certain position x.

By discretizing and doing some algebra we obtain the matrix problem

$$\left[\begin{pmatrix} V(\Delta x) & 0 & \dots & & \\ 0 & V(2\Delta x) & 0 & & \\ \vdots & 0 & \ddots & & \\ & & & 0 & \\ & & & 0 & V(N\Delta x) \end{pmatrix} - \frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & \dots & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & & \ddots & 1 \\ & \dots & 0 & 1 & -2 \end{pmatrix}\right]\begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \\ \psi_N \end{pmatrix} = E\begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \\ \psi_N \end{pmatrix}. \tag{11}$$

Note that the right matrix is the same as in (3) and (9). The difference is of course that the system also contains the diagonal matrix $V_{\Delta x}$ with $V(x)$ evaluated at the interior points.

A numerical solver method `schr(N, v, k=6)` can now be constructed where `N` is the number of interior grid points, `v` is an array of the function $V(x)$ evaluated at these interior points, and `k` is the amount of eigenvalues and corresponding eigenvectors that should be returned. The implementation was very similiar to `slv`. First we calculate Δx and create both the matrices $V_{\Delta x}$ and $T_{\Delta x}$. Then we can compute the eigenvalues and eigenvectors using `scipy.sparse.linalg.eigsh` and of course insert and append the boundary values. Finally we plot all the eigenvectors (wave functions) and the probability densities at their respective "energy levels" in two separate figures as well as returning the eigenvalues. The plots are also scaled slightly to make them nicer.

2.2.1 Using the solver on a simple problem

In order to test the solver it was first used to to solve the simplest case where $V(x) = 0$ in the interval [0, 1] and $V(x) = \infty$ outside the interval. This is usually referred to as a particle in a potential

box. The boundary values have to both be zero since the probability of finding the particle outside the box is zero. The Schrödinger equation in this case reduces to the Sturm-Liouville problem

$$\psi'' = -E\psi.$$

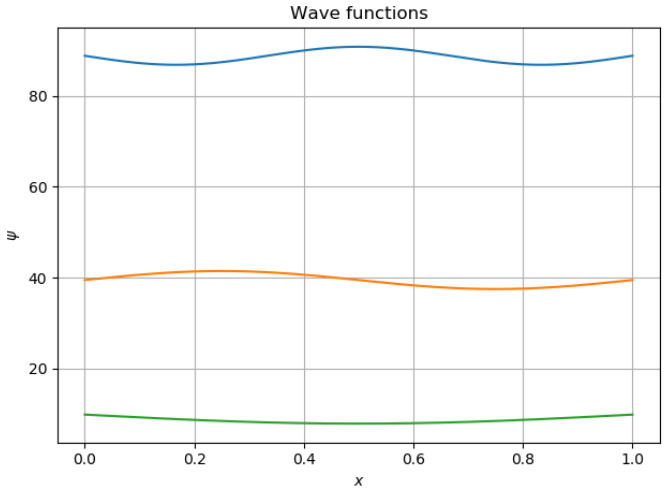

Figure 6: Eigenmodes for $V(x) = 0$

As we can see in Figure 6 the first three modes are the expected sin-waves, similar to the ones in Figure 5. Therefore the solver method works.

2.2.2 Using the solver on less simple problems

We can now use the solver on less simple and more interesting problems.

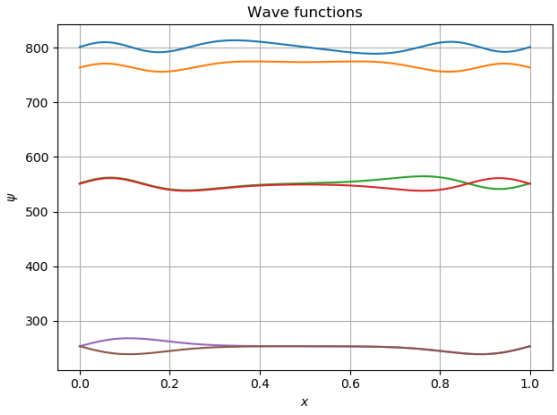

Figure 7: Eigenmodes for $V(x) = 800sin^2(\pi x)$

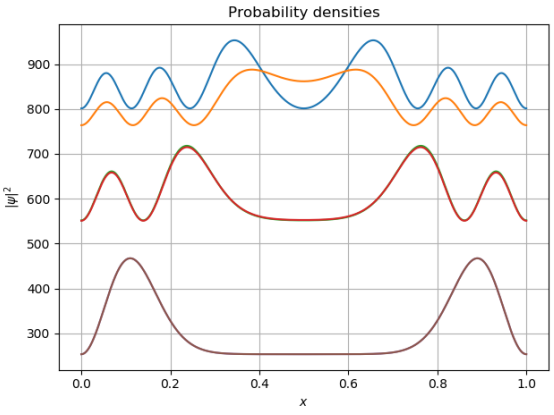

Figure 8: Probability densities for $V(x) = 800sin^2(\pi x)$

In Figure 7 we can see the effects of a "doublet state", meaning, two states of almost the same energy level (but different parity). Specifically, we have doublet states for the two lowest energy levels $E \approx 253$ and $E \approx 551$. Additionally, the probability of finding the particle at $x = 0.5$ is only significant for the second highest energy level $E \approx 764$ according to Figure 8.

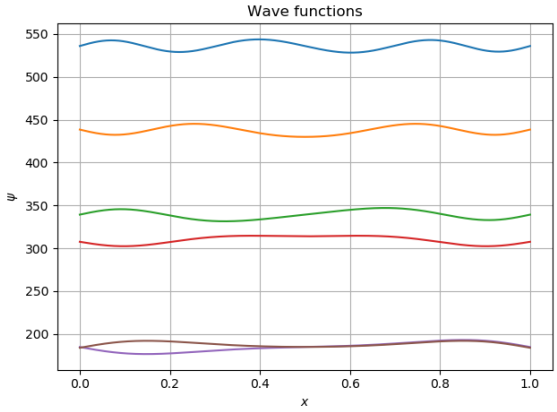

Figure 9: Eigenmodes for $V(x) = 700(0.5 - |x - 0.5|)$

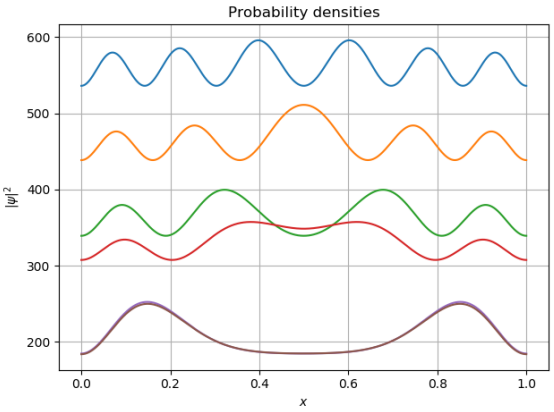

Figure 10: Probability densities for $V(x) = 700(0.5 - |x - 0.5|)$

In this case, there is only one doublet state for the lowest energy level $E \approx 184$, however the probability of finding the particle at $x = 0.5$ is now significant for two energy levels, namely $E \approx 308$ and $E \approx 439$ (see Figure 9 and 10).

The goal now was to create a potential with three "wells" so that we could get a triplet state. After some trial and error we found $V(x) = 500(1 + \sin(7\pi x))$.

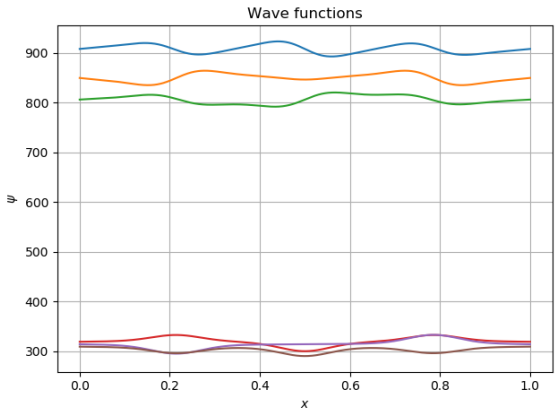

Figure 11: Eigenmodes for $V(x) = 500(1 + \sin(7\pi x))$

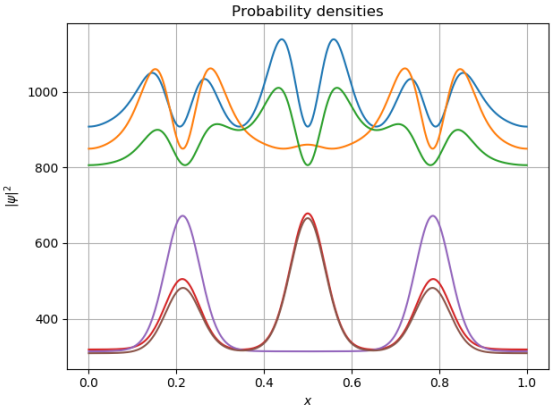

Figure 12: Probability densities for $V(x) = 500(1 + \sin(7\pi x))$

Here we have a rough triplet state for the lowest energy level $E \approx 314$ (see Figure 11). Interestingly though, finding the particle at $x = 0.5$ is only significant for two of these (see Figure 12).

Conclusions

We performed the project tasks individually, and wrote our own code. Then we compared solutions and discussed differences and possible improvements. In the presentation we chose to present Erik's solutions in the first task and Daniel's solutions in the second one. We have gained some general knowledge of boundary value problems, numerical approximation of higher order derivatives and their areas of use. Specifically setting the discretizations up as matrix problems was very interesting. Furthermore we have learned some tricks involving the use of sparse matrices for significantly faster computation. Finally we have gotten more comfortable in using Python and the NumPy and SciPy packages.

For some acknowledgements we would like to give credit to classmates for some rewarding discussions regarding the tasks and general ideas, in particular Max Gustafson and Kåre von Geijer.

References

All the theory comes from the lecture notes and project instructions.

Code

Task 1

The code in this task was written by Erik.

```
import numpy as np
import scipy.sparse as spa
import scipy as sp
import scipy.sparse.linalg as spb
from  matplotlib.pyplot import *
```

1.1

```
def func(x):
    return -sin(x)

def vecfunc(x):
    f = []
    if isinstance(x, list) or isinstance(x, np.ndarray):
        for i in x:
            f.append(func(i))
        s = np.asarray(f)
    else:
        s = func(x)
    return s

def genfunc(L, N):
    deltax = L/(N + 1)
    x = linspace(deltax,L-deltax, N)
    f = vecfunc(x)
    return f

def exact(x):
    return sin(x)

def vecexact(x):
    f = []
    if isinstance(x, list) or isinstance(x, np.ndarray):
        for i in x:
            f.append(exact(i))
        s = np.asarray(f)
    else:
```

```
        s = exact(x)
    return s

def twopBVP(fvec, alpha, beta, L, N):

    deltax = L/(N + 1)
    sub = ones(N - 1)
    sup = ones(N - 1)
    main = -2*ones(N)
    fvec[0] = fvec[0]-alpha/(deltax**2)
    fvec[-1] = fvec[-1]-beta/(deltax**2)
    al = np.array([alpha])
    be = np.array([beta])

    A = spa.diags(sup,-1) + sp.sparse.diags(main,0) + sp.sparse.diags(sub,1)
    A = A/(deltax**2)
    y = spb.spsolve(A, fvec)
    y = np.insert(y, 0, al)
    y = np.append(y, be)

    return y

def ploterror(alpha, beta, L):
    N=array([2**(i+2) for i in range(9)])
    E = []
    for i in range(len(N)):
        t = linspace(0, L, N[i]+2)
        exakt = vecexact(t)
        fvec = genfunc(L, N[i])
        s = twopBVP(fvec, alpha, beta, L, N[i])
        e = abs(s - exakt)
        E.append(norm(e)/sqrt(N[i]+1))
    F = np.asarray(E)
    deltax = L/(N+1)
    loglog(deltax,F,'o')
    title('Error as a function of stepsize')
    xlabel('Stepsize')
    ylabel('Error')
    grid(color='gray', linestyle='--')
```

1.2

```
def eifunc(L):
    def somefunc(x):
        return 1.9e8*(3-2*cos((pi*x)/L)**12)
```

```
    return somefunc

def vecifunc(x):
    f = []
    if isinstance(x, list) or isinstance(x, np.ndarray):
        for i in x:
            f.append(eifunc(i))
        s = np.asarray(f)
    else:
        s = eifunc(x)
    return s

def solvebeam(L=10, N=999):
    q = -50e3*ones(N)
    M = twopBVP(q, 0, 0, L, N)
    x = linspace(0, L, N+2)
    func = eifunc(10)
    EI = func(x)
    f = M/EI
    u = twopBVP(f[1:-1], f[0], f[-1], L, N)
    plot(x,u)
    xlabel('x(m)')
    ylabel('u(m)')
    title('Deflection u as a function of x')
    print(u[500].round(9))
```

Task 2

The code for this task was written by Daniel.

functions.py

```
import numpy as np
from scipy.sparse import diags, linalg
import matplotlib.pyplot as plotter

def slv(N, k):
    dx = 1 / (N + 1)
    diagonals = [np.ones(N - 1), -2 * np.ones(N), np.ones(N - 1)]
    T = 1 / (dx ** 2) * diags(diagonals, [-1, 0, 1], format="csr")
    eig, u = linalg.eigsh(T, k, which='SM')
    u = np.insert(u, 0, np.zeros([1, k]), axis=0)  # insert zeroes at beginning
    u = np.append(u, np.zeros([1, k]), axis=0)  # append zeroes

    return eig, u
```

```
def schr(N, v, k=6):
    dx = 1 / (N + 1)
    diagonals = [np.ones(N - 1), -2 * np.ones(N), np.ones(N - 1)]
    T = 1 / (dx ** 2) * diags(diagonals, [-1, 0, 1], format="csr")
    V = diags(v, format="csr") # both matrices in sparse format

    eig, u = linalg.eigsh(V - T, k, which='SM')
    u = np.insert(u, 0, np.zeros([1, k]), axis=0)  # insert zeroes at beginning
    u = np.append(u, np.zeros([1, k]), axis=0)  # append zeroes

    scale = abs(max(eig)) * .2 # scale for plotting purposes.

    x = np.linspace(0, 1, N + 2)

    wave = plotter.figure(1)
    for i in range(k): # plotting in reverse order to get smallest eigenvalues first
        plotter.plot(x, eig[k - 1 - i] + scale * u[:, k - 1 - i], )
    plotter.xlabel(r'$x$')
    plotter.ylabel(r'$\psi$')
    plotter.title('Wave functions', figure=wave)
    plotter.grid()

    dens = plotter.figure(2)
    for i in range(k):
        plotter.plot(x, eig[k - 1 - i] + (scale * abs(u[:, k - 1 - i])) ** 2)

    plotter.xlabel(r'$x$')
    plotter.ylabel(r'$|\psi|^2$')
    plotter.title('Probability densities', figure=dens)
    plotter.grid()
    plotter.show()

    return eig
```

task2.1a.py

```
import numpy as np
import functions as f
from scipy.linalg import norm
import matplotlib.pyplot as plotter
import math
import time
```

```
K = 11
N = np.arange(2, K)
N = np.power(2, N)


def ev(k):  # eigenvalues are -(kpi)^2 for positive integers k
    return -(k * math.pi) ** 2


ev = ev(np.arange(1, 4))  # first 3 exact eigenvalues
err = np.zeros([3, K - 2])  # error array for the 3 eigenvalues, all N

ex = time.time()

for i in range(K - 2):
    eig, u = f.slv(N[i], 3)  # first 3 eigenvalues

    for k in range(3):  # error for k-th eigenvalue
        err[k, i] = norm(eig[k] - ev[2 - k])

print(time.time() - ex)

plotter.loglog(N, err[2, :], label=r'$\lambda_1$')
plotter.loglog(N, err[1, :], label=r'$\lambda_2$')
plotter.loglog(N, err[0, :], label=r'$\lambda_3$')
plotter.xlabel('N')
plotter.ylabel(r'$\lambda_{\Delta x} - \lambda$')
plotter.legend()
plotter.title('Numerical errors')
plotter.grid()
plotter.show()
```

task2.1b.py

```
import numpy as np
import functions as f
import matplotlib.pyplot as plotter
import time

N = 999

ex = time.time()

eig, u = f.slv(N, 3)
```

```
print(time.time() - ex)

x = np.linspace(0, 1, N + 2)

plotter.plot(x, u[:, 2], label=r'$\sin(\pi x)$')
plotter.plot(x, u[:, 1], label=r'$\sin(2\pi x)$')
plotter.plot(x, u[:, 0], label=r'$\sin(3\pi x)$')
plotter.xlabel(r'$x$')
plotter.ylabel(r'$u(x)$')
plotter.title('First three numerical modes')
plotter.legend()
plotter.grid()
plotter.show()
```

task2.2a.py

```
import numpy as np
import functions as f
import time

N = 999

v = np.zeros(N)

ex = time.time()

eig = f.schr(N, v, k=3) # Only want the first 3

print(time.time() - ex)
```

task2.2b.py

```
import numpy as np
import functions as f
import math
import time

N = 999


def eq1(x):
    return 700 * (.5 - abs(x - .5))
```

```
def eq2(x):
    return 800 * np.sin(math.pi * x) ** 2


def eq3(x):
    return 500 + 500 * np.sin(7 * math.pi * x)


x = np.linspace(0, 1, N + 2)  # Interior grid points and the boundary points

# Evaluate the potential at the interior grid points

v1 = eq1(x[1:-1])
v2 = eq2(x[1:-1])
v3 = eq3(x[1:-1])

ex = time.time()

# I comment out different parts to get different plots

# eig1 = f.schr(N, v1)
# eig2 = f.schr(N, v2)
eig3 = f.schr(N, v3)

print(time.time() - ex)
```