MATH3511/6111: Scientific Computing
Mathematical Sciences Institute, Australian National University
Semester 1, 2022

Lab 5: Fourier Transform, ODEs
Lindon Roberts
Due date: 9am, Tuesday 31 May (week 13)

This lab is based on earlier versions by Stephen Roberts, Graeme Chandler, Jimmy Thomson, Linda Stals and Kenneth Duru.

1 Fast Fourier Transform

Remember from lab 1 that Python can easily work with complex numbers (using the `cmath` package):

```
import cmath

z1 = 1 - 2j  # j represents the complex number i
z2 = 3 + 1j  # 'j' is the variable j, so use '1j' to mean the complex number
print("z1 =", z1, ", z2 =", z2)
print("z1 + z2 =", z1 + z2, ", z1 * z2 =", z1*z2)

theta = 2 * cmath.pi / 3
z = cmath.exp(theta * 1j)  # remember exp(theta * i) = cis(theta)
print("z =", z)
```

NumPy can also create and manipulate complex vectors/matrices:

```
import numpy as np

x = np.array([z1, z2, z])          # create directly from complex numbers
y = np.zeros((3,), dtype=complex)  # zero vector, but all entries are 'complex' data type
print(x)
print(np.exp(y))
```

Lab Book 1. Write a function which creates the Fourier transform matrix F_n (for any input n, not necessarily a power of 2).[1] For $n = 10$, use suitable norms to verify that F_n is symmetric and that $F_n\overline{F}_n = nI$. [5 points]

Hint: `np.conjugate(x)` calculates the complex conjugate of every entry entry in an array. Also, `np.linalg.norm` can give norms of complex vectors/matrices.

As we know from lectures, we can calculate the DFT of a vector $\boldsymbol{x}$ by computing $F_n\boldsymbol{x}$. However, using the FFT is a better method if n is a power of 2.[2] A Python version of the FFT algorithm is given in the lectures, and is also available as `np.fft.fft(x)`.

Lab Book 2. For $n = 1, 2, 4, 8, 16, \ldots, 8192$ (or less if it takes too long), plot the runtime of (a) building the matrix F_n using your code above; (b) calculating the DFT of $\boldsymbol{x} = (1, 2, 3, \ldots, n)^T$ using matrix-vector multiplication (not including creating the matrix F_n); and (c) calculating the same DFT using NumPy's FFT function. See lab 4 for code which can measure runtime. Discuss whether your results match the expected results from theory. [4 points]

[1] You can test your function by comparing F_1, F_2 and F_4 to the matrices given in the lecture notes.

[2] If n is not a power of 2, similar FFT-like algorithms can be used and they are still very fast, although often not quite as good as powers of 2. The specific algorithm depends on the prime factorisation of n.

1.1 Application: Denoising Audio Data

On Wattle I have saved a file `lab5_piano_data.csv`, which has the data from the piano note analysis from lectures (slides 47–49).[3] Specifically, it contains the amplitude of the sound wave of a 1-second recording of a piano playing 'middle C' ($\approx$ 262Hz), stored as a vector of length 44100 (the recording microphone samples the sound wave 44100 times per second). If you save this csv in the same directory as your Python file, you can read this file into a NumPy vector with

```
data = np.loadtxt('lab5_piano_data.csv', delimiter=',')
time = np.linspace(0.0, 1.0, len(data))  # data represents 1 second of audio
print("CSV has a vector of size =", data.shape)
```

Note that `np.savetxt` can be used to save a NumPy array to a text file; this is useful if you want to load the data into another program.

Replicate the plot from lectures: plot the magnitude of the first 2000 entries of the DFT of the data — you should see spikes at multiples of 262Hz. The relative size of the spikes at different multiples of the 'base' frequency are called overtones. Each musical instrument sounds different, even when playing the same note, because they all have different combinations of overtones.

We are going to use the DFT (which is quick to compute for $n = 44100$ because of the FFT algorithm) to remove noise from our data. First, create a copy of the audio data polluted by random noise:

```
np.random.seed(0)  # produce the same random numbers every time the code is run (optional)
noisy_data = data + 0.005 * np.random.randn(len(data))  # perturb data with random noise
```

Now, if you plot the magnitude of the DFT of both `data` and `noisy_data` (just plot the first 1000 entries so you can see the plot clearly), you will see that the noisy data has the same spikes but also lots of frequencies where the magnitude is small, but much larger than for `data`.

Lab Book 3. Looking at your plots, pick a magnitude which is slightly larger than the typical 'noise' contribution, but much smaller than the size of any spikes. Modify the DFT of `noisy_data` by setting to zero all entries with magnitude less than your chosen level, and calculate the inverse DFT to produce a 'denoised' signal (you will have to look up how to calculate an inverse DFT in NumPy). Produce a plot the original signal, noisy signal and your denoised signal and check that you have removed a reasonable amount of the noise. Quantitatively compare the error (versus `data`) of the noisy and your denoised signal using suitable norms, to check that your denoised signal is closer to the true data than `noisy_data`. [5 points]

If you want to listen to the original, noisy and your denoised data, you can save the NumPy vectors to an audio file (`.wav` format) using SciPy:

```
import scipy.io.wavfile as wavfile

samplerate = 44100  # samples per second in the audio
# Note: after an inverse DFT you usually get complex values with very small imaginary parts
# (of size machine epsilon), which we need to remove before saving.
wavfile.write('my_audio.wav', samplerate, np.real(data))
```

If your computer can't play this file, you can convert to common formats such as mp3 using online converters.

This technique (remove noise by zeroing out frequencies with small magnitudes) is common in signal processing, and can be used for audio, images and video. It can also be used for things like data compression (just save the locations/values of the spikes in frequency). The DFT is best for periodic data (like this particular audio file), but other related transforms are better for non-periodic data such as images (e.g. wavelet transform).

[3] Source: University of Iowa Electronic Music Studios, `http://theremin.music.uiowa.edu/MISpiano.html`.

2 Solving ODEs

In this section you will implement various methods for solving ODEs.

As our first test problem, we will solve

$$\frac{du}{dt} = cu(t)(1-u(t)), \qquad u(0) = u_0,$$

where c is some constant and u_0 is an initial condition.

This is a simple model for the spread of disease through a population. Suppose the population has size P, and let $u(t)$ be the fraction of the population which is infected at time t, so $1-u(t)$ is the fraction of the population that is healthy. New infections occur (i.e. $\frac{du}{dt} > 0$ so $u(t)$ increases) when healthy and infected populations meet, which is proportional to $u(t)(1-u(t))$. In our model, we use the constant c to be the constant of proportionality, which will depend on the infectiousness of the disease and how frequently populations meet.

Let's suppose we have at total population $P = 10000$ and $c = 0.2$, and we start with one infectious person, $u_0 = 1/P$. The true solution of the ODE is

$$u(t) = \frac{1}{1+(P-1)e^{-ct}}.$$

Show mathematically that the formula for $u(t)$ solves the ODE.

Note that if $u_0 = 0$, then the solution is $u(t) = 0$ for all t, but if we use any $u_0 > 0$, the solution $u(t)$ does not converge to zero as $t \to \infty$. This means that $u(t) = 0$ is an unstable solution to the ODE.

We have studied several one-step methods ($u_{k+1} = u_k + h\phi(t_k, u_k)$ where $h = t_{k+1} - t_k$) for solving ODEs. The following code gives a basic outline of how to use a one-step method to solve a general ODE,

$$\frac{du}{dt} = f(t,u), \qquad u(0) = u_0,$$

on an interval $[0, T]$ using $n+1$ equally spaced points.

```
# RHS of the ODE
def f(t, u):
    return 0.0  # TODO - implement this

# One-step method
def phi(t, u, h):
    return 0.0  # TODO - implement this

u0 = 0.0  # initial condition
T = 10.0   # end time

n = 10  # use n+1 equally spaced time steps
ts = np.linspace(0, T, n+1)  # vector of timesteps, tk = ts[k]
h = T / n  # gap between timesteps

u = np.zeros((n+1,))  # create an empty vector for our solution
u[0] = u0             # set initial condition
# Run the one-step method
for k in range(n):
    u[k+1] = u[k] + h * phi(ts[k], u[k], h)
```

Lab Book 4. Modify the above code to implement our ODE (f and u_0) for time horizon $T = 100$ days. Then, implement the one-step methods: explicit Euler, Heun's method and the classical 4th order Runge-Kutta scheme (RK4). For $n = 200$, plot your three computed solutions against the true solution $u(t)$. Interpret the solution to this ODE in terms of the application setting (infectious disease). [6 points]

Lab Book 5. For $n = 50, 100, 200, 400, 800, 1600$ and each of the three one-step methods, produce a table similar to slide 41 of the ODE lectures, computing the maximum absolute error $|u_k - u(t_k)|$ at any time step t_k for each method. From your table, determine what order of convergence you are seeing in each method and compare this to the theory in lectures. [6 points]

2.1 Implicit Methods

We can also use implicit methods. Modify your code to implement the implicit Euler method (slide 49). You will need to use a rootfinding method (such as `scipy.optimize.root_scalar` from lab 2) in each iteration to calculate u_{k+1}.

2.2 Built-in Methods

Of course, SciPy implements many different ODE methods in the `scipy.integrate` module. You can access a variety of solvers using the `solve_ivp` function.

```
import scipy.integrate as integrate

# RHS of the ODE
def f(t, u):
    return 0.0  # TODO - implement this

u0 = np.array([0.0])  # initial condition (must be a NumPy array)
T = 10.0              # end time

sol = integrate.solve_ivp(f, [0, T], u0, method='RK45')
# sol.t has the time steps where the solution is evaluated
# sol.y[0,:] has the value of the solution at times sol.t (even if only one
# output, to be consistent even for systems of ODEs).
# There are many choices for 'method' - check the documentation

# If you want sol.y at specific times (not the ones chosen by the solver),
# then add this an extra input: solve_ivp(..., t_eval=ts)
```

The same function can also solve systems of ODEs (where $f(t, u)$ returns a vector and u_0 is a vector of the same length).

We will now assume that we start administering vaccinations at day $t = 5$, where 100 people are (fully) vaccinated per day. This means that the number of immune people is

$$\frac{100 \max(t - 5, 0)}{10000}.$$

Then, our ODE changes: vaccinated people can't get infected so new infections only happen when an infected person meets a healthy, vaccinated person. We get

$$\frac{du}{dt} = cu(t) \max\left(1 - u(t) - 0.01 \max(t - 5, 0), 0\right).$$

Lab Book 6. Use `solve_ivp` with RK45 to solve the original and our new ODE and plot the two solutions. How many people are healthy after 100 days under the two scenarios? How many extra people would be healthy at day 100 if we started vaccinating on day $t = 0$ rather than day $t = 5$? [5 points]

2.3 Solving BVPs

Boundary value problems (BVPs) are much more complicated mathematically than initial value problems. The simplest method for solving BVPs is the shooting method: try many different initial conditions, and use a rootfinding method to find the initial condition(s) for which the actual boundary conditions are satisfied. In the lectures, we looked at the Bratu problem

$$u''(t) + 3e^{u(t)} = 0, \qquad u(0) = u(1) = 0,$$

with the domain $t \in [0, 1]$, which has two solutions.

For the shooting method, we instead look at the family of IVPs

$$u''(t) + 3e^{u(t)} = 0, \qquad u(0) = 0, \quad u'(0) = v_0,$$

where v_0 is a parameter that we can change. Use `solve_ivp` to plot the solution $u(t)$ for a few different choices of v_0.

Write a function which, given a value of v_0, uses `solve_ivp` to solve this IVP and returns the solution $u(1)$. Make a plot of $u(1)$ versus v_0 (like slide 79 of the lectures). Then, run a rootfinding algorithm (like `scipy.optimize.root_scalar`) from two different starting choices of v_0, to find the two values of v_0 which give $u(1) = 0$.

Check that these values of v_0 give a function $u(t)$ which solves the BVP by plotting both solutions (like slide 80 of the lectures).

Warning! We do not have the ability to differentiate the value of $u(1)$ with respect to v_0, so we cannot use a rootfinding method which uses derivatives (like Newton's method). However, there are techniques for calculating the derivative of solutions to differential equations with respect to various inputs ('adjoint methods'). This is particularly useful when using implicit schemes with the method of lines for nonlinear PDEs (see Section 2.4), because multidimensional rootfinding without derivatives is very difficult.

2.4 Solving PDEs: Method of Lines

Although there are mahy techniques for solving PDEs, one approach is the 'method of lines', where we discretise a PDE in one (or more) dimensions, so that it becomes a system of ODEs. We will try this technique here to solve the heat equation (but also models option prices in finance and chemical diffusion, for example):

$$\frac{\partial u}{\partial t} = c\frac{\partial^2 u}{\partial x^2},$$

on the spatial domain $x \in [0, 1]$ and time domain $t \geq 0$. We impose the initial conditions

$$u(0, x) = f(x),$$

and boundary conditions

$$u(t, 0) = a, \qquad \text{and} \qquad u(t, 1) = b.$$

Here, a, b, c are constants and $f(x)$ is a known starting distribution (e.g. of temperature).

If we apply finite differencing to the spatial derivative, we would get

$$\frac{\partial^2 u}{\partial x^2}(t, x) \approx \frac{u(t, x - \Delta x) - 2u(t, x) + u(t, x + \Delta x)}{\Delta x^2}.$$

Now, if we discretise in space, $x_j = j\,\Delta x$ for $j = 0, \ldots, n$ and $\Delta x = 1/n$, we can define the vector function

$$\boldsymbol{u}(t) = \begin{bmatrix} u_0(t) \\ u_1(t) \\ \vdots \\ u_n(t) \end{bmatrix} = \begin{bmatrix} u(t, x_0) \\ u(t, x_1) \\ \vdots \\ u(t, x_n) \end{bmatrix},$$

where we introduce the notation $u_j(t) = u(t, x_j)$. From our boundary conditions we have $u_0(t) = a$ and $u_n(t) = b$ and so $u_0'(t) = u_n'(t) = 0$. Otherwise, we can use our finite differencing and get the system of ODEs

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{f}(t, \boldsymbol{u}(t)) = c \begin{bmatrix} 0 \\ (u_0(t) - 2u_1(t) + u_2(t))/\Delta x^2 \\ \vdots \\ (u_{n-2}(t) - 2u_{n-1}(t) + u_n(t))/\Delta x^2 \\ 0 \end{bmatrix},$$

with initial conditions

$$\boldsymbol{u}(0) = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix}.$$

Of course, we can solve this system of ODEs with whatever method we wish, but for simplicity we will use the explicit Euler method.

Lab Book 7. Solve this system of ODEs using the explicit Euler method, $\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \Delta t\, \boldsymbol{f}(t, \boldsymbol{u}_k)$ for some time discretisation Δt. Using data $a = b = 0$, $c = 1$ and $f(x) = \sin(\pi x)$, plus spatial discretisation $n = 10$ and time step $\Delta t = 0.5/n^2$, plot the solution $u(t, x)$ versus x for $t = 0, 0.1, 0.2, \ldots, 0.5$. [5 points]

If our initial data is $f(x) = 0.5 - x$, $a = 0.5$ and $b = -0.5$, the true solution is $u(t, x) = f(x)$ (i.e. the temperature doesn't change over time).

Lab Book 8. With this choice of $f(x)$, plot the solution after 500 iterations for $\Delta t = \alpha/n^2$ where $\alpha \in \{0.48, 0.5, 0.52\}$ and $n = 100$. What do you observe? [4 points]

One important reason to mathematically study the method of lines is to know the relationship needed between Δt and Δx for the method to be stable. For the heat equation, stability analysis can be used to show that we should always set $\Delta t \leq \Delta x^2/(2c)$. If we want a very fine spatial discretisation (i.e. Δx small) in order to see small-scale behaviour of the solution, this means we have to use extremely small timesteps, which will be very slow.