Tutorial 3

The core exercises are considered part of the course material, and you are advised to at least finish these. For additional practice and interesting applications, advanced exercises are available as well – feel free to pick and choose from these.

Unless otherwise specified, we expect you to create your own algorithms for the exercises in each tutorial. This means that you should **not use libraries (e.g. `SciPy`, `numpy`)** for the methods that are part of the course material.

You are free to write and execute code on your own laptop, but please be aware that all code you hand in needs to run on the vdesk machines[1] out of the box (e.g. no installing additional packages), so you may want to get used to this already.

Core exercises

1. **Image Compression via SVD**

   In much the same way humans do not need the entirety of an eye's perceptive strength in order to understand most of what is going on, singular value decomposition (SVD) can be used to tell which parts of an image contain the most information. The largest singular values capture the most important information, such as the dominant shapes as encoded by the largest brightness variations. The things we still see even when we squint. The information that is tossed away is instead encoded on the smaller singular values. This is often exploited by modern communication technologies in order to compress images.

   Download this, frankly incredible, picture `https://home.strw.leidenuniv.nl/~kilmetis/pics/petiti_luigi.jpg`. You can load it into `Python` through the following script.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

img = mpimg.imread('petiti_luigi.jpg')
grey_img = np.dot(img[..., :3], [0.640, 0.595, 0.155]) # Rec601 hack to turn greyscale
# Do NOT fall into the rabbit hole of that is the
# luminous efficiency function for human eyes.

# In case you need to see the image
plt.imshow(grey_img, cmap="gray")
```

   (a) Use `np.svd`, to calculate the singular value decomposition of the provided picture, that is the U, W, V^T matrices. Reconstitute the original picture via the U, W, V^T matrices.

   (b) Compress the picture by truncating the U, W, V^T arrays to their first 5, 10, 50 elements and use the truncated arrays to reconstruct the image. How many singular values do we *really* need to keep the important details of an image?

[1] `https://helpdesk.strw.leidenuniv.nl/wiki/doku.php?id=manuals:virtualdesktopserver`

2. **Balancing Chemical Equations**

One useful application of solving systems of linear equations is to balance chemical equations. For simpler equations, this can be done more quickly by hand, however for complicated equations, being able to solve them numerically can be very handy. Consider the following chemical equation:

$$H_3PO_4 + (NH_4)_2MoO_4 + HNO_3 \rightarrow (NH_4)_3PO_4.12MoO_3 + NH_4NO_3 + H_2O$$

To balance the equation, we first put coefficients x_1 to x_6 before each part of the above equation and construct linear equations for each of the elements H, P, O, N, and Mo as follows:
$H : 3x_1 + 8x_2 + 1x_3 = 12x_4 + 4x_5 + 2x_6$
$P : 1x_1 + 0x_2 + 0x_3 = 1x_4 + 0x_5 + 0x_6$
$O : 4x_1 + 4x_2 + 3x_3 = 40x_4 + 3x_5 + 1x_6$
$N : 0x_1 + 2x_2 + 1x_3 = 3x_4 + 2x_5 + 0x_6$
$Mo : 0x_1 + 1x_2 + 0x_3 = 12x_4 + 0x_5 + 0x_6$

As there are 6 variables and 5 equations, we have one free parameter. For convenience, we can take the values corresponding to x_6 to the right side and obtain a square matrix by taking the other values to the left side. The augmented matrix will be:

$$\left[\begin{array}{ccccc|c} 3 & 8 & 1 & -12 & -4 & 2 \\ 1 & 0 & 0 & -1 & -0 & 0 \\ 4 & 4 & 3 & -40 & -3 & 1 \\ 0 & 2 & 1 & -3 & -2 & 0 \\ 0 & 1 & 0 & -12 & -0 & 0 \end{array}\right]$$

Let's try to solve the above system of equations. When implementing the methods discussed in the lectures, you will get values for x_1 to x_5 with respect to the value of x_6. Choose a convenient normalisation to obtain the balancing coefficients with the smallest integer numbers.

(a) Implement an algorithm that solves the above system of equations using your choice of either: Gaussian elimination with back substitution, or Gauss-Jordan elimination, following the procedure described in the lecture.

(b) Now, try to solve the same system using the LU decomposition method, for example by implementing Crout's algorithm, as discussed in the lecture. For this you could make use of parts your code for part (a).

(c) Use the `timeit` module to compare the time required for the above two methods. Which one is more efficient in solving this system of equations? Why?

Advanced exercises

3. **Tridiagonal systems**

Sometimes the matrix for a system of equations has a particular form which makes it possible to solve it more efficiently than e.g. making an LU decomposition. In this exercise we'll see a common example of this: a tridiagonal system. In this case the matrix only has non-zero entries along its diagonal and directly next to it on both sides.

First a reminder for lecture 2: for cubic spline interpolation, given N sample points (x_i, y_i), we saw last week that there is a linear system of equations we have to solve in order to find the three non-trivial polynomial coefficients in each of the $N-1$ intervals. Let's call the polynomial defined on interval $[x_i, x_{i+1}]$ $f_i(x)$. Its value, first and second derivative are then given by:

$$\begin{aligned} f_i(x) &= y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \\ f_i'(x) &= b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2 \\ f_i''(x) &= 2c_i + 6d_i(x - x_i). \end{aligned} \tag{1}$$

Note that for $x = x_i$, we just find $f_i(x_i) = y_i$, $f_i'(x_i) = b_i$ and $f_i''(x_i) = 2c_i$. Recall that to get the system of equations to solve, we demand that the values, derivatives and second derivatives on the left and right intervals match at each sample point, i.e. $f_i(x_{i+1}) = f_{i+1}(x_{i+1})$, $f_i'(x_{i+1}) = f_{i+1}'(x_{i+1})$ and $f_i''(x_{i+1}) = f_{i+1}''(x_{i+1})$ for all relevant i. We need two extra demands, which in the case of the natural cubic spline are that $f_0''(x_0) = f_{N-2}''(x_{N-1}) = 0$.

At this point we could write our equations in matrix form $\mathbf{Ax} = \mathbf{b}$ with a $3(N-1)$-dimensional vector of unknowns $\mathbf{x}$ consisting of all b_i, c_i and d_i – but this would be incredibly inefficient! In general, you want to reduce the dimensionality of your problem before you try to solve it numerically. The important thing to realize, in this case, is that (1) all the equations are linear in each of the polynomial coefficients, and (2) that the coefficients of each interval only (directly) depend on those of the neighbouring intervals, e.g. c_4 directly depends on c_3 and c_5 but no other c_i. This means that we might rewrite this system of equations to a tridiagonal one. There are a couple of ways to do so (we could for example write every equation in terms of the second derivative), below we work through just one.

(a) First use $f_i''(x_{i+1}) = f_{i+1}''(x_{i+1})$ to find an expression for d_i. Use the shorthand $\Delta x_i \equiv (x_{i+1} - x_i)$ and don't worry about the boundaries for now. Then substitute this expression for d_i in the equations $f_i(x_{i+1}) = f_{i+1}(x_{i+1})$ and $f_i'(x_{i+1}) = f_{i+1}'(x_{i+1})$. You now have a system of equations only in terms of b_i, b_{i+1}, c_i and c_{i+1}.

(b) Next, use the new equations for $f_i(x_{i+1}) = f_{i+1}(x_{i+1})$ to express b_i in terms of c_i and c_{i+1}. Note that you can also use this same equation to write b_{i+1} in terms of c_{i+1} and c_{i+2}.

(c) Now use the equations of (b) in $f_i'(x_{i+1}) = f_{i+1}'(x_{i+1})$ to find a set of equations in terms of only the coefficients c_i, c_{i+1} and c_{i+2}. Then, substitute $i \to i-1$. Now we are left with just $N-1$ equations, in terms of only c_i and the c-coefficients of the directly neighbouring intervals!

(d) Recall that for the *natural* cubic spline, $c_0 = c_n = 0$, so we have $N-1$ equations with $N-1$ unknowns. Write the $N-1$ equations you found in (c) in matrix form, with $\mathbf{x} = (c_1\, c_2\, \ldots\, c_{N-1})^{\mathrm{T}}$. You should now have a tridiagonal matrix $\mathbf{A}$!

(e) An LU decomposition of a tridiagonal system only has non-zero entries on the diagonal and one off-diagonal for both the $\mathbf{L}$ and $\mathbf{U}$ matrices. Check that this must indeed be the case, and derive equations for each of the non-zero LU matrix coefficients α_{ij} and β_{ij} in terms of the (non-zero) elements a_{ij} of a tridiagonal matrix $\mathbf{A}$ (remember that $\alpha_{ii} = 1$). You should find that each a_{ij} can be written as the sum of only two terms, only one of which is unknown if you derive the α_{ij} and β_{ij} in order – this means that the LU decomposition can be found very quickly, in only $\mathcal{O}(N)$ operations!

(f) Code up a natural cubic spline interpolation solver using this tridiagonal system and the accompanying LU decomposition. Then test it by taking $N = 4$ and sample points $(0, 2)$, $(1, 0)$, $(2, 1)$ and $(3, 2)$ and solve for the coefficients c_i $(0 < i < N)$. Use substitution to calculate b_i and d_i and plot the four points and your resulting cubic spline.