Coursework 2

Lucjano Muhametaj

December 7, 2021

1 Exercise 1

The weekly exercises can be found on
https://github.com/Imperial-MATH96023/clacourse-2021-lm1621/tree/master/cla$_utils$

2 Exercise 2

2.1 a

I want to compute the matrix $C = (xy^T)^k$. So what I want to do is: $C = \underbrace{xy^T xy^T ...xy^T}_{ktimes}$. We can see that $y^T x$ is an inner product and we say it's equal to μ. So we will have that $C = \mu^{k-1}xy^T$, where xy^T is an outer product, which has computational cost $O(n^2)$, since I do n times n multiplications.

2.2 b

Let's suppose to have two matrices B (mxm) and A (mxn). We want to compute the multiplication BA. Let's observe that for determining the first element of the resulting matrix I will multiply the elements of the first row of B with the elements of the first column of A and sum them. This will give us $m + m - 1 = 2m - 1$ operations. We will repeat this mn times (for the other columns and rows). So we get that to compute the multiplication BA we do $mn(2m - 1)$ operations.
Let's now observe that BA is a mxn matrix and A^T is a nxm matrix, so when we compute $A^T(BA)$ if we proceed similarly as before we get that the operations count is $n^2(2m-1)$. So to compute $A^T(BA)$ we will have an operation count equal to $n^2(2m - 1) + mn(2m - 1)$.
Similarly we can compute the computation count for $(A^T B)A$, which is going to be $n^2(2m - 1) + mn(2m - 1)$.
So the two algorithms have the same operation count. So using the second one instead of the first doesn't change anything in terms of operations count in any case.

2.3 c

Let's observe that $A = PR + iQR + iPS - QS$ and $T = PR - PS + QR - QS$. It's enough to compute the multiplication T (which is just a multiplication between the two matrices $P + Q$ and $R - S$) and the multiplications QR and PS. So we can basically the real part of A as $A = T + PS - QR$ and the the complex part as $iQR + iPS$. So I computed A using only three matrix multiplications.
When we multiply two mxm matrices we have an operations count which is $m^2(2m-1)$, on the other hand when we sum two mxm matrices we have an operations count of m^2. So when we compute the T multiplication we have in total $2m^2+m^2(2m-1)$, while for QR and PS, we have for each one $m^2(2m-1)$. So at the end we are going to have a total operation count of $\underbrace{2m^2 + m^2(2m-1)}_{T} + \underbrace{2m^2(2m-1)}_{QR,QS} + 3m^2$, where $3m^2$ counts the two final sums I did to compute the real part of A and the one sum I did to compute the complex part of A.
At the end we are going to have $6m^3 + 2m^2$ operations.

3 Exercise 3

3.1 a

Given λ_1, λ_2 the eigenvalues of A, we can say that this algorithm is stable if for little perturbations of A we have little perturbations of (λ_1, λ_2).
Let's write the $2x2$ matrix A as

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{1}$$

Using the quadratic formula we get that $\lambda_1 = \frac{a+d+\sqrt{(a+d)^2-4(ad-cb)}}{2}$ and $\lambda_2 = \frac{a+d-\sqrt{(a+d)^2-4(ad-cb)}}{2}$.
Let's call $f : R^4 \to R^2$ (which takes the matrix A, and so (a, b, c, d) and gives (λ_1, λ_2)) and $\hat{f}$ the floating point implementation of f. We have that (according the definition given in the course notes) the algorithm $\hat{f}$ is stable for f if

$$\frac{\|\hat{f}(A) - f(A + \delta A)\|}{\|f(A + \delta A)\|} = \mathcal{O}(\epsilon) \tag{2}$$

for some δA such that

$$\frac{\|\delta A\|}{\|A\|} = \mathcal{O}(\epsilon). \tag{3}$$

It's important to note that the quadratic formula is not always stable, for example if the term under the square root is really near the term outside (like what happens in our case if $4(ad - cb)$ is really small), then we are going to end up subtracting two nearly equal numbers numbers for one of the two lambdas. This problem can be avoided if we use an alternative version of the quadratic formula.

3.2 b

As before let's call $f : R^4 \to R^2$ (which takes the matrix A, and so (a, b, c, d) and gives (λ_1, λ_2)) and $\hat{f}$ the floating point implementation of f. We say that this algorithm is backard stable for f if for each A there exist δA, such that

$$\hat{f}(A) = f(A + \delta A) \tag{4}$$

with $\frac{\|\delta A\|}{\|A\|} = \mathcal{O}(\epsilon)$.

3.3 c

In the code I have written a function which computes the eigenvalues of a given $2x2$ matrix using the quadratic formula. Then taken the two matrices $A_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and $A_2 = \begin{pmatrix} 1 + 10^{-14} & 0 \\ 0 & 1 \end{pmatrix}$, we can find the eigenvalues for both of them using the function and we can also compute an error (based on the norm of the difference between the eigenvalues I get from the algorithm and the exact eigenvalues, which are for each matrix the terms on the diagonal). We observe that for the second matrix if we stay on real values when we run the code we get an error. This is the reason why in the code I passed on complex values.
In particular for the first matrix we get: (1,1) with error 0, while for the second one we have (1.+1.49011612e-08j, 1.-1.49011612e-08j) with an error 2.10734242554482e-08.
The code for this exercise can be found in the script `cw3c.py`

3.4 d

Let's observe that for the second matrix we have a characteristic polynomial equal to $x^2-(2+10^{-14})x+1+10^{-14}$. Now let's observe that computing the coefficients of this polynomial we expect to find errors of the order of the machine error. We remember that the machine error is $\epsilon_{machine} = 10^{-16}$ and so we will find roots with an error of the order of $\sqrt{\epsilon_{machine}}$, which is practically 10^{-8}, which is the error we found in the previous answer.
Let's now try to be more precise. We know that the sum is backward stable, when we compute

the quadratic formula, we will have that for example λ_1 we have: (for λ_2 we can make the same observations):

$$\frac{2+10^{-14}+\epsilon_1+\sqrt{(2+10^{-14}+\epsilon_2)^2-4(1+10^{-14}+\epsilon_3)}}{2} \tag{5}$$

Now if we consider only the root part we will have that $\sqrt{(2+10^{-14}+\epsilon_2)^2-4(1+10^{-14}+\epsilon_3)} = \sqrt{10^{-28}+\epsilon_2^2+4(\epsilon_2-\epsilon_3)+2\epsilon_2^{-14}}$. We know that $\epsilon_{machine}$ is greater than $|\epsilon_2|$ and $|\epsilon_3|$ and that (as explained in the section 3.8 of the notes) the gaps in $[2,4]$ are greater than the gaps between $[1,2]$ (in particular we are talking about 2^{-51} and 2^{-51} respectively). Now we can note that the only terms that make a difference under the root are basically of the order of 10^{-16}, so that computing the root we will have 10^{-8}, as we wanted.

4 Exercise 4

4.1 a

We write a function to generate A for given n, following the structure described in the exercise. Notice that in the code I took directly $\epsilon = 0.1$, but the function can also be written in order to take the value as an input. Let's also notice that for each B_k we sum a $5x5$ matrix and that two consecutive matrices B_k and B_{k+1} overlap only in one element of the matrix (where we will have a sum).
So basically A has a sort of banded structure with block of $5x5$ overlapping matrices. (So indeed if we have two blocks we will have one overlap which makes sense dimensionally, since we get 5+5-1=9). Let's also note that the overlapping happens always on the diagonal and in particular on the $A[4k+1, 4k+1]$ for $k = 1, 2, ...$
Once written a function for generating a matrix A like asked in the exercise we can easily apply the `LUinplace()` function taken from the `clautils` and then extract L and U.
We can check that for $n = 2$ (so for $9x9$ matrices) we have that L (which is of course lower triangular) can be seen as a matrix written by $5x5$ blocks such that each block is a lower triangular matrix and they overlap on the corners. We can see that this happens for each n. We can extrapolate a similar structure for U but with upper triangular matrix blocks. So we can say that both U and L have a structure similar to A but upper/lower triangular.
The code for this exercise can be found on `cw2` in the file `cw4a.py`

4.2 b

We can implement Gaussian elimination to the given matrix A avoiding operations related to zero (so basically additions and multiplications by zero). We saw that A has basically a banded structure, so if we are in a block we can avoid operating on what is outside our block.
So what we do is taking U equal to A, L equal to the identity and then start a loop for k from 1 to $m-1$, in which we determinate in which block we are at the moment. We do it setting the block number as the integer part of the division $k/4$ and then summing 1. Then we can compute the max index of the block we are in, which is going to be $m = 4 * blocknumber + 1$. Now we start an inner loop similar to the one in the classical Gaussian algorithm for banded matrix, but we run this loop for j from $k+1$ to m and we basically compute

$l_{jk} \leftarrow u_{jk}/u_{kk}$ and then $u_{jk:m} \leftarrow u_{j,k:m} - l_{jk}u_{k,k:m}$ Now for the operations count we can use the fact from the notes that the decomposition LU has an operations count equal to $\frac{2m^3}{3}$. In the algorithm we actually computed this for each of the $5x5$ blocks, so if we have n blocks (for a $4n+1x4n+1$ final matrix A) we are going to end up with $\mathcal{O}(n)$. This can be shown also calculating the number of operations by hands from the algorithm

4.3 c

The code for this exercise can be found on `cw4b.py` (`bandedalgorithm`) (NOTE: NOT `cw4.b.py`, the right one is the one without the dot between 4 and b) in the `cw2` folder. In particular it has been computed using the outer product as in the weekly exercise. The automatic test can be found in the `testcw4b.py`

4.4 d

(1)If we take a matrix created with 2 overlapping blocks matrices we can transform the first block in an upper triangular block and the second block in a lower triangular one with an opportune multiplication for a matrix B. We want to solve BAx=Bb now. We observe that now is enough considering a matrix 3x3 in correspondence with the overlap. In particular I take using python notation (and indices starting from 1) we note that we obtained a 3x3 tridiagonal matrix A'=BA[4:6,4:6] and now we can solve easily A'x[4:6]=Bb[4:6] obtaining x_4, x_5, x_6. Now it's evident that this is enough to solve the initial matrix-vector problem (it's enough to use backward substitution on one of the blocks and forward on the other one). Now if the number of the blocks is even we can proceed in pairs of two consecutive overlapping blocks, if it's odd we do the same except from the last block. So in this case we write BA such that we have an upper block, then a lower, then an upper then a lower ... (so we proceed in pairs) until the last one which will be an upper triangular matrix 5x5 since we have odd number of blocks. So now we apply what we said before on the pairs and finally we just apply backward substitution to the last block. We can easily check that using this algorithm the number of the flops for transforming a block into a lower or upper tridiagonal is a constant depending on the dimension 5x5 of the blocks. So the operation count depends on the number of the blocks (O(n)). Also the the solution of the 3x3 tridigonal matrices and for the substitutions the operation count for each one of them is constant, so at the end considering that I repeat those steps for each block I get $O(n)$
(2) We notice that we can compute other algorithms really similar to this one, just changing the way we work on the blocks.

4.5 e

In `cw4d.py` in the folder `cw2` there are the starting tools to write the code for this question, but I didn't manage to finish it

5 Exercise 5

5.1 a

We want to write the expression as a matrix-vector equation for the vector v. First of all we observe by the expression of v that $v = (u_{1,1}, u_{1,2}, ..., u_{2,1}, u_{2,2}...., u_{n-1,n-1})$, and that we can write S in a similar way. In particular we can write a b such that $b = (S_{1,1}, S_{1,2}, ..., S_{2,1}, ...S_{n-1,n-1})$. Then we can construct a A in order to write the equation as $Av = b$. Let's observe that in the equation given in the exercise, we have a sum of three parts on the left. This makes easier to write A as a sum of three matrices. In particular considering that $\Delta x = \frac{1}{n}$, we are going to end up with an expression of this tipe:

$$A = \frac{1}{2}nB - \mu n^2 C + cI \tag{6}$$

where A, B, C, I are $(n-1)^2$x$(n-1)^2$ matrices and in particular I is the identity matrix.
Now we can see from the expression what structure B and C are going to have. In particular B is going to look like:

$$B = \begin{pmatrix} 0 & b^2_{1,1} & 0 & \cdots & 0 & b^1_{1,1} & 0 & \cdots & 0 \\ -b^2_{1,2} & 0 & b^2_{1,2} & 0 & \cdots & 0 & b^1_{1,2} & 0 & \cdots \\ 0 & \ddots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \ddots & \ddots & \vdots \\ -b^1_{2,1} & 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & -b^1_{2,2} & 0 & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \ddots & \ddots & \vdots \end{pmatrix}$$

And A is going to be like:

$$C = \begin{pmatrix} -4 & 1 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 1 & -4 & 1 & 0 & \cdots & 0 & 1 & 0 & \cdots \\ 0 & \ddots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \ddots & \ddots & \vdots \\ 1 & 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 1 & 0 & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \ddots & \ddots & \vdots \end{pmatrix}$$

Both these matrix are banded, so we are going to end up with a final matrix A which is going to be banded. I particular both the upper and the lower bandwidths are equal to $n-1$.
Now for what regards the operation count, we recall what's written in section 4.4 of the notes, where is specified that the operation count for banded matrix with dimension nxn and bandwidths equal to p and q is equal to $\mathcal{O}(npq)$. So in this case, if we consider large n we can say that the operation count is $\mathcal{O}(n^4)$

5.2 b

The code can be found on `cw5b.py` in the `cw2` folder. In the file I have also implemented a function that times the algorithm for different n and plots the timing divided for the expected operation count n^4. The resulting plot is the the one below. So for big n the timing divided by n^4 converges to a constant, which is what we expected.

5.3 c

In the previous questions we've seen that $u_{i,j}$ can be flattened in a vector v and we can do the same for $S_{i,j}$. Moreover we've seen that we can think about the problem presented in this exercise as a matrix-vector equation. In the same way we can see the expression (6) of this point of the exercise 5 as matrix-vector equation $Av^{k+1/2} = b$, where b can be calculated starting from v^k (note that we are supposed to know v^k, to compute $v^{k+1/2}$). We can do the same thing for expression (7) to get v^{k+1} from $v^{k+1/2}$. We remember that $u_{ij} = v_{(n-1)(i-1)+j}$, so if we follow the same reasoning that we used in the part (a) of this exercise to get the matrix, we are going to have for expression (6) a matrix-vector equation with a banded matrix (with lower and upper bandwidth equal to $n-1$). On the other hand for (7) we are going to get a tridiagonal matrix.
Now we observe that we can apply the LU algorithm for banded matrices that we have written previously to solve both (6) and (7). However for (7) the operations are going to be less. An idea for avoiding this problem is to change the interpretation of (6). We can do it using $u_{ij} = v'_{(n-1)(j-1)+i}$ instead of $u_{ij} = v_{(n-1)(i-1)+j}$. In this way the LHS can be seen as $A'v'$, where this time A' is now a tridiagonal matrix.
(Note that $S_{i,j}$ is going to flattened in the same way as v' in the (6) and in the same way as v in the (7)).
So to sum up starting from the stage k, we flatten $u^{k+1/2}$ as v', calculate b' ,just doing operations on the RHS (can be done writing a matrix and performing a multiplication or if we want to avoid some sums and multiplications for 0 just doing the operations in the expression (for example in a loop)) and then solve the tridiagonal system $A'v' = b'$, using LU factorisation (and then backward/forward substitution). Then we calculate b of the (7) and solve now $Av = b$. In this way we can obtain u^{k+1}.
For the operation count we see that we can write b and b' just doing the operations of the expression (No more than $\mathcal{O}(n^2)$). Then we see that we perform LU for banded matrices on matrices with bandwidth 1, so we have $\mathcal{O}(n^2)$.Then for the backward and forward substitution we can see in the lecture that we have $\mathcal{O}(mp)$ and $\mathcal{O}(mq)$. Considering that the bandwidths are 1 we have again $\mathcal{O}(n^2)$. So at the end for all the algorithm we have $\mathcal{O}(n^2)$.
This exercise can be done also avoiding to change the flattening of u between the different steps. (check ex 5d explanation below)

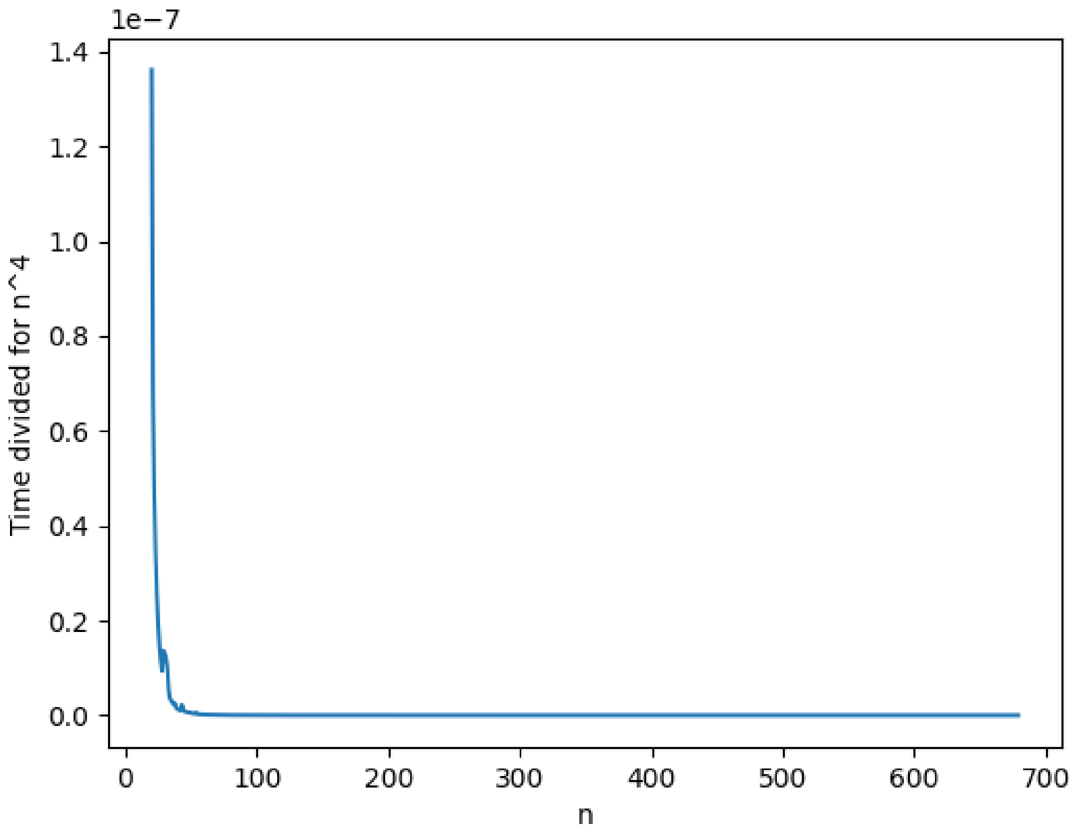

Figure 1: Plot

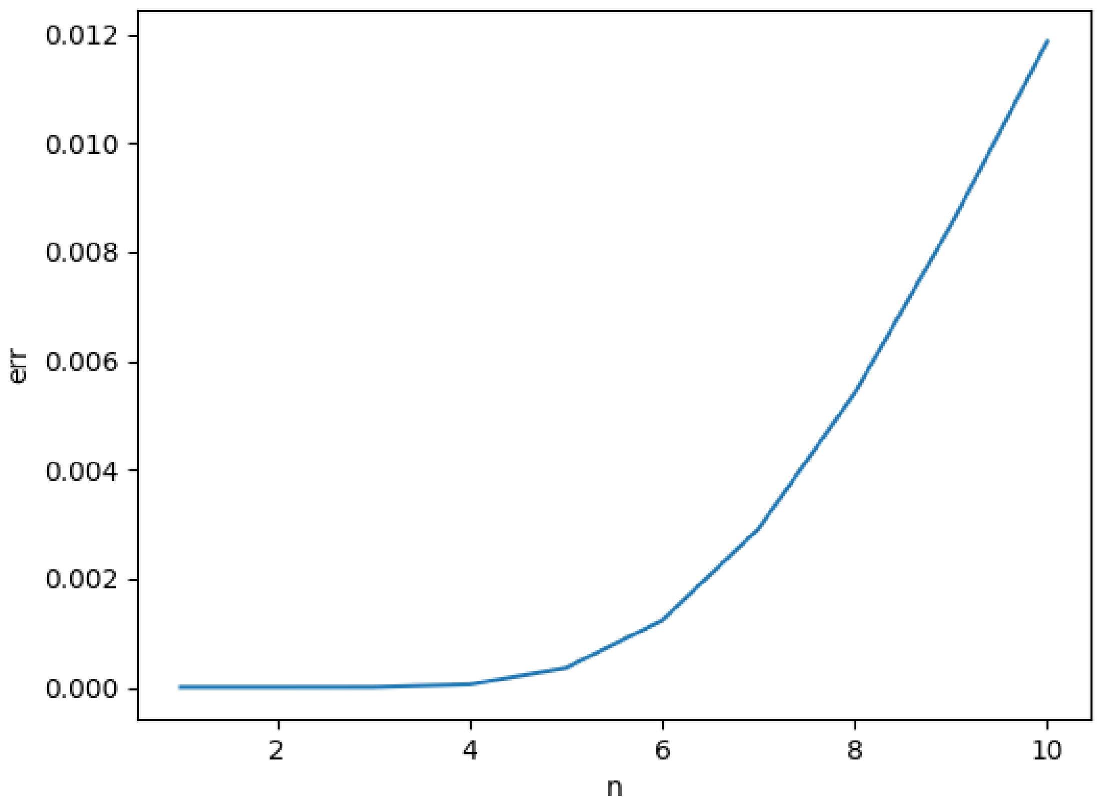

Figure 2: Caption

5.4 d

The code is implemented in the folder `cw2` in `cw5d.py`. Firstly I wrote the matrices S and b from their definition (I wrote them directly flattened). Then I have written a function for banded matrices for backward substitution, one for forward substitution and a function which uses them to solve sistems with banded matrices. Then I wrote a function to implement a modified version of the algorithm written above. In this version I don't use the different flattening trick, but I flatten only one time using the standard rule given by the exercise. ((n-1)(i-1)+j). In this way called A0 the matrix on the LHS for the half step and A1 the LHS matrix for the full step, and B0 and B1 this two matrices without the elements on the mai diagonal it's enough to iterate the following thing :
I write $b_k = flattened(S) - B1 * v_k$, I solve $A_0 v_{k+1/2} = b_k$, then I write $b_{k+1/2} = flattened(S) - B0 * v_{k+1/2}$. I noticed in particular that if I implement everything after dividing for $\Delta(x)^2$ and I use the builtin function for solving systems the code works way better (I did this in order to observe the behaviour of the algorithm better). Also for small parameters (smaller than 1 (except of course n)) the code doesn't create problems.

5.5 e

We can easily note that the choice of parameters changes the accuracy of the solution and the speed for reaching the solution. For example plotting the error and the number of iterations for increasing dimension starting from n=1 (so for the the step i on the plot we refer to n=n+i) we get what follows:
We then note that given a max number of iterations (for example 55 as in our case) both the two plots tend to stabilize. This is because the while loop in the code stops before reaching the desired accuracy since it goes over the 55 iterations. We can do similar observation for the other parameters. In general for smaller parameters the algorithm is going to work better and so it will need less steps.

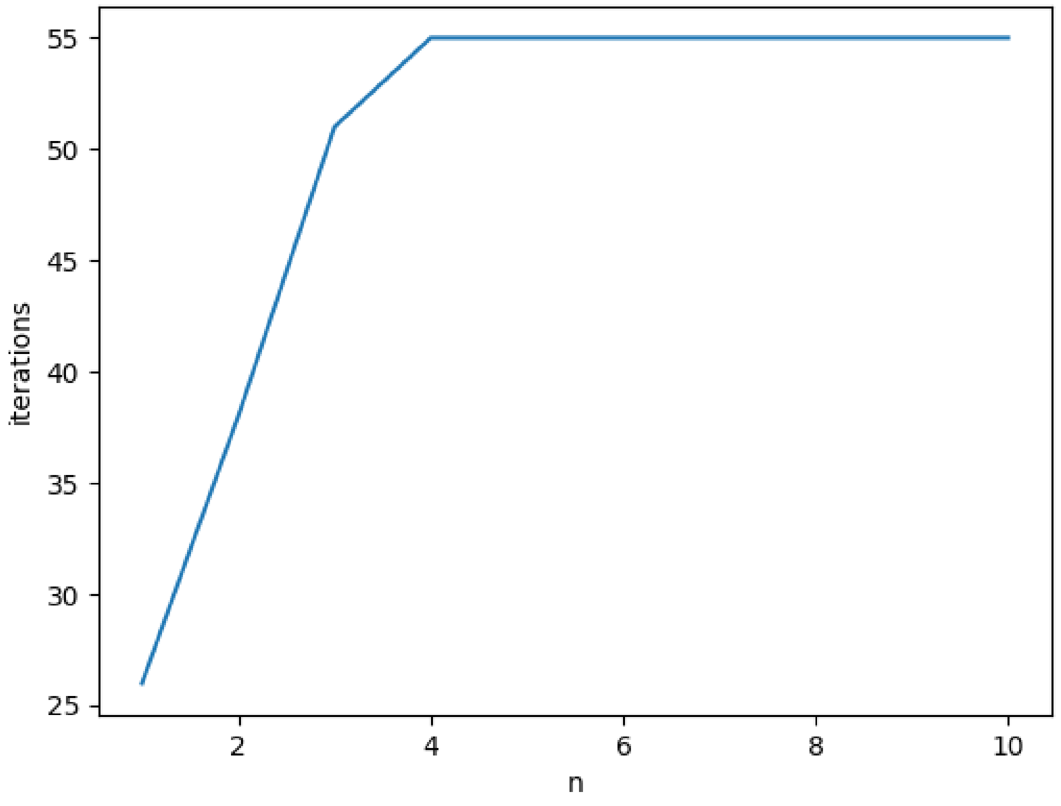

Figure 3: number of iterations

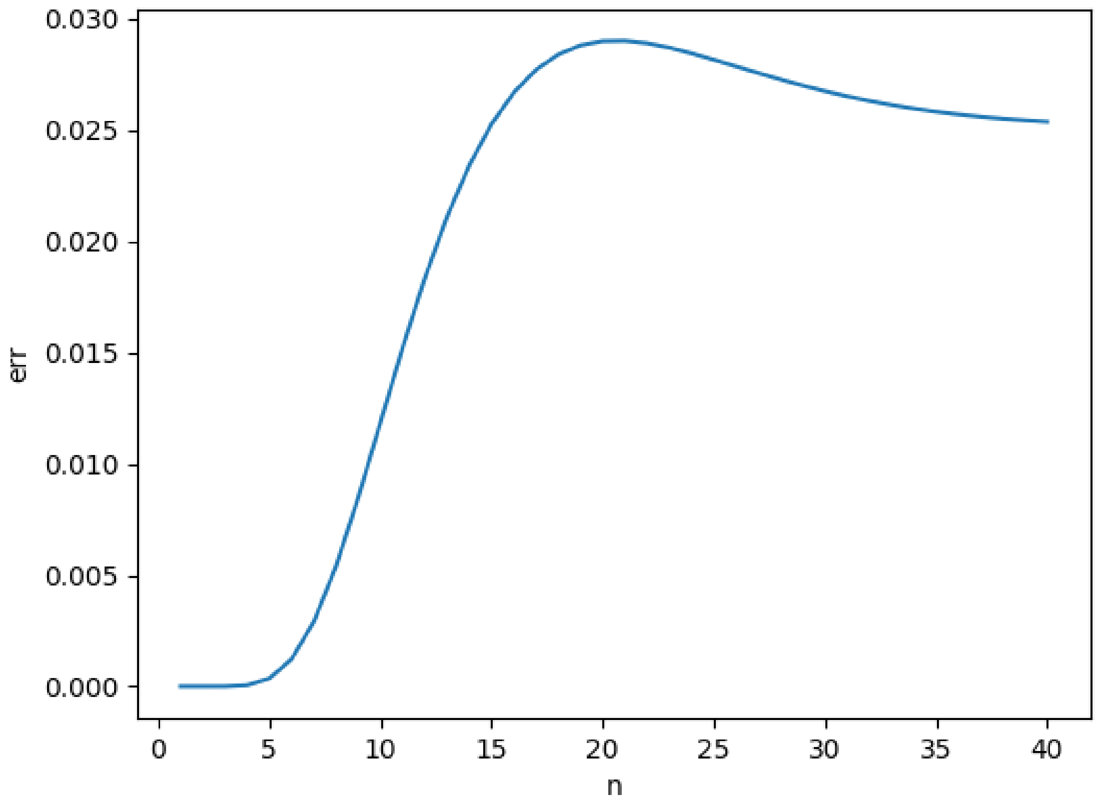

Figure 4: Error for i from 1 to 40

The code for the plotting functions can be found on `cw5b.py` in the `cw2` folder. (NOTE: some of the function plotting stuff in the code of this exercise are really slow and require a lot of time)