Handin Assignment 1

Ioannis Koutalios s3365530

January 6, 2025

Abstract

Code and results for Handin assignment 1 for the course Numerical Recipes in Astrophysics.

1 Poisson Distribution

```python
import numpy as np

def Poisson32(l: float, k: int) -> float:
    '''
    we calculate the poisson probability given l and k
    we make sure we operate with 32bit numbers in every step
    when we multiply 32bit int with float the output is a 64bit float
    so we convert int32 to float32 before making the operations
    result is float32
    '''
    l = np.float32(l)
    k = np.int32(k)
    ans = np.float32(1)
    for i in range(k,0,-1):
        x = np.float32(i)
        ans *= l/x
    return ans*np.exp(-l)


if __name__ == "__main__":
    # we define the pairs of lamda,k and print the output of our poisson function
    l , k = 1 , 0
    print(f'P({l,k}) = {Poisson32(l,k):.6E}')

    l ,k = 5 ,  10
    print(f'P({l,k}) = {Poisson32(l,k):.6E}')

    l , k = 3 , 21
    print(f'P({l,k}) = {Poisson32(l,k):.6E}')

    l , k = 2.6 , 40
    print(f'P({l,k}) = {Poisson32(l,k):.6E}')

    l , k = 101 , 200
    print(f'P({l,k}) = {Poisson32(l,k):.6E}')
```

poisson.py

For this task, we want to create a function that takes as an input two parameters (λ and k) and returns the Poisson probability. The k parameter is an integer while λ is a float. We use numpy types to ensure that every variable that we use is at any point in 32bit accuracy. The output is also a 32bit float.

The function works by using a for loop to calculate the $\frac{\lambda^k}{k!}$. Inside the for loop, we calculate the fraction $\frac{l}{x}$ where x goes from k to 1 and add the result of each loop. The operation is done using float numbers because otherwise the precision of the output changes. After that, we multiply with $e^{-\lambda}$ to get the probability. The reason we chose this approach is that by calculating the ratio we avoid any overflow issues that would come if we multiplied all the numbers and then divided the two values.

```
P((1, 0)) = 3.678794E-01
P((5, 10)) = 1.813279E-02
P((3, 21)) = 1.019340E-11
P((2.6, 40)) = 3.615118E-33
P((101, 200)) = 1.299921E-18
```

output/poisson.txt

We test our function by printing the results for five different pairs of (λ, k). The results are printed with 7 significant digits, using scientific notation.

When comparing the results with other ways of calculating the same probability we can see that the precision of our function is great. The only pair that has some loss of precision is $(\lambda, k) = (101, 200)$. The loss is on the 3rd significant digit.

2 Vandermonde matrix

In this section, we want to create and test a script that uses the Vandermonde matrix to calculate the Lagrange polynomial. The Vandermonde matrix can be defined with the relation $V_{ij} = x_i^j$ where i, j go from 0 to $N-1$, with N being the length of x. If we solve the system $Vc = y$ we can get the c coefficients of the Lagrange polynomial.

2.1 LU Decomposition

```python
import os, sys
import numpy as np
import matplotlib.pyplot as plt

def Vandermonde(x: np.ndarray) -> np.ndarray:
    '''
    creates the vandermonde matrix
    loops over all columns and rows, calculates
    the value and saves it to the correct position
    '''
    van = np.zeros((len(x),len(x)))
    for j,x_i in enumerate(x):
        for i in range(len(x)):
            van[j,i]=x_i**i
    return van

def LUDecomp(A: np.ndarray) -> tuple:
    '''
    function to perform the LU decomposition in matrix A
    we initialize L U and then apply the Crouts algorithm
    we use matrix multiplications to avoid unnecessary loops
    '''
    N = len(A)
    L = np.zeros((N,N))
    U = np.zeros((N,N))
    for i in range(N):
        L[i,i]=1
        summ=np.dot(L[i,:i],U[:i,i:])
        U[i,i:]=A[i,i:]-summ
        summ=np.dot(L[i:,:i],U[:i,i])
        L[i:,i]=(A[i:,i]-summ)/U[i,i]
    return L,U

def ForwSub(A: np.ndarray, Y: np.ndarray) -> np.ndarray:
    '''
    forward substitution to be used when solving a system
    '''
    X = np.zeros_like(Y)
    for i in range(len(X)):
        summ = 0
        for j in range(i):
```

```
            summ += A[i,j]*X[j]
        X[i]=(Y[i]-summ)/A[i,i]
    return X

def BackSub(A: np.ndarray, Y: np.ndarray) -> np.ndarray:
    '''
    backward substitution to be used when solving a system
    '''
    X = np.zeros_like(Y)
    for i in range(len(X)-1,-1,-1):          # we iterate backwards
        summ = 0
        for j in range(len(X)-1,i-1,-1):     # we iterate backwards
            summ += A[i,j]*X[j]
        X[i]=(Y[i]-summ)/A[i,i]
    return X

def PredictVander(x: np.ndarray, c: np.ndarray) -> np.ndarray:
    '''
    generater the lagrange polynomial
    takes as an inpute an array x to interpolate at
    and the array of coefficients c
    '''
    y_pred = np.zeros_like(x)
    for i,x_i in enumerate(x):
        for j,c_j in enumerate(c):
            y_pred[i] += c_j*(x_i**j)
    return y_pred


if __name__ == "__main__":
    # get the data
    data = np.genfromtxt(os.path.join(sys.path[0],"data/Vandermonde.txt"),comments='#',
    dtype=np.float64)
    x = data[:,0]
    y = data[:,1]
    xx = np.linspace(x[0],x[-1],1001)   # x values to interpolate at

    vander=Vandermonde(x)            # calculate the vandermonde matrix
    l , u = LUDecomp(vander)         # decompose it to l and u
    u_c = ForwSub(l,y)               # use forward substitution to solve l*(u_c)=y
    c = BackSub(u,u_c)               # use backward substitution to solve u*c=u_c
    print(f'c = {c}')                # print the solutions c
    y_pred = PredictVander(xx,c)     # generate the predictions

    # plot the polynomial with the data points
    fig_1 = plt.figure()
    plt.scatter(x,y,label='data',color='black')
    plt.plot(xx,y_pred,label='Lagrange polynomial',color='r')
    plt.xlim(-1,101)
    plt.ylim(-400,400)
    plt.legend()
    plt.xlabel('$x$')
    plt.ylabel('$y$')
    plt.savefig('./plots/vander_pol.png', bbox_inches='tight', dpi=300)
    plt.close()

    y_2 = PredictVander(x,c)        # generate predictions in the same positions with the
     data points

    # plot the absolute difference between predictions and actual points
    # second plot is in logscale to show the small differences in the first points
    fig_2 = plt.figure()
    plt.scatter(x,abs(y-y_2),label=r'$|y(x)-y_i|$',color='sienna')
    plt.xlabel('$x$')
    plt.ylabel('$y$')
    plt.legend()
    plt.grid()
    plt.yscale('log')
    plt.savefig('./plots/vander_dif.png', bbox_inches='tight', dpi=300)
    plt.close()
```

lu_decomp.py

For this task, we want to create a function that does an LU decomposition of the Vandermonde matrix and then use it to solve for c. The function "vandermonde" uses the definition of the Vandermonde matrix to generate it given an array of x. Then we have the function "LU_dec" which uses Crout's algorithm to decompose a matrix A into two matrices L, U. The way we perform the calculations is by using a single for loop and matrix multiplications to calculate $\beta_{ij} = \alpha_{ij} - \sum_{k=0}^{i-1} \alpha_{ik}\beta_{kj}$ and $\alpha_{ij} = \frac{1}{\beta_{jj}}(\alpha_{ij} - \sum_{k=0}^{i-1} \alpha_{ik}\beta_{kj})$, where α, β are the elements of L, U respectively.

We then define two routines to perform forward and backward substitution that is needed to solve the system $LUc = y$. Finally, we need a function to generate the interpolated values. The function "predict_vander" takes as input an array x of the values we want to interpolate at and an array c that corresponds to the coefficients of the Lagrange polynomial. After that it uses the formula $y_i = \sum_{j=0}^{N-1} c_j x_i^j$ to calculate the predictions.

In our script, we load the data we were given and create the array we will use for our interpolation. We then generate the Vandermonde matrix and decompose it. After that, we use forward substitution to solve $Lb = y$ where $b = Uc$ and backward substitution to solve $Uc = b$ for c. We print the results and then generate the interpolated predictions. We create a plot for both the original data points and the generated polynomial. The final plot we create shows the absolute difference between the predicted points and the actual data. To create it we need to interpolate at the same values of x as our original points.

```
c = [ 1.73240075e+01 -1.89881240e+02  2.39181294e+02 -1.10029212e+02
  2.68384792e+01 -4.06887408e+00  4.17173672e-01 -3.04347333e-02
  1.63298394e-03 -6.58808502e-05  2.02755266e-06 -4.79981672e-08
  8.76391617e-10 -1.23047440e-11  1.31548547e-13 -1.05084380e-15
  6.07015041e-18 -2.39387188e-20  5.76479982e-23 -6.39290272e-26]
```

output/lu_decomp.txt

The coefficients of the Lagrange polynomial, as were calculated in our script.

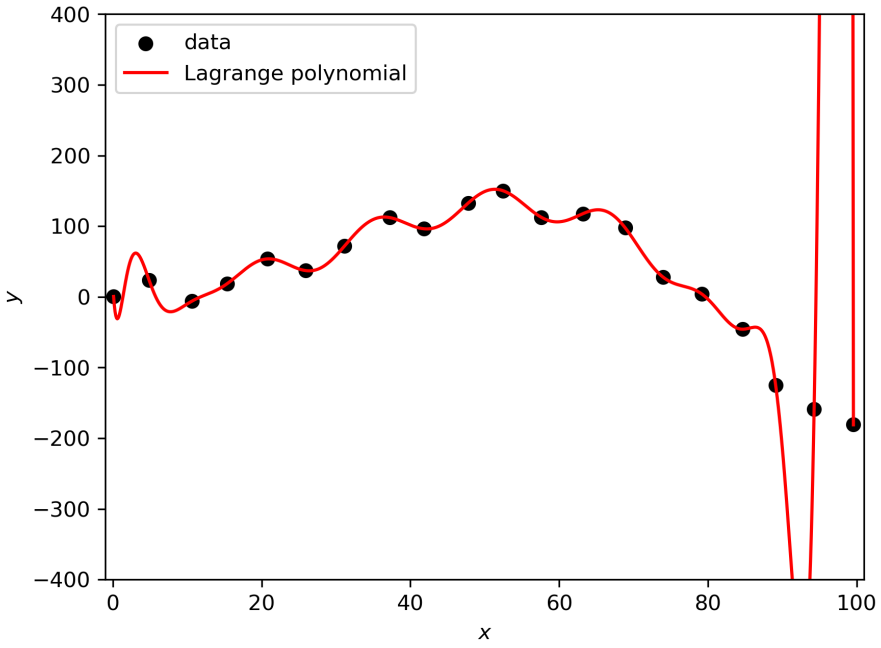

Figure 1: The Lagrange polynomial with the original data points. To generate the polynomial we used the Vandermonde matrix. The system was solved using the LU decomposition method. We notice that the polynomial crosses all the points in our dataset.

As we see in Figure 1 the generated polynomial passes through all of our data points as we would expect. It is worth mentioning that for this example we use the full 19th order polynomial that can be generated using all 20 points of our data. However, this interpolation is not so useful as it greatly overfits our data.

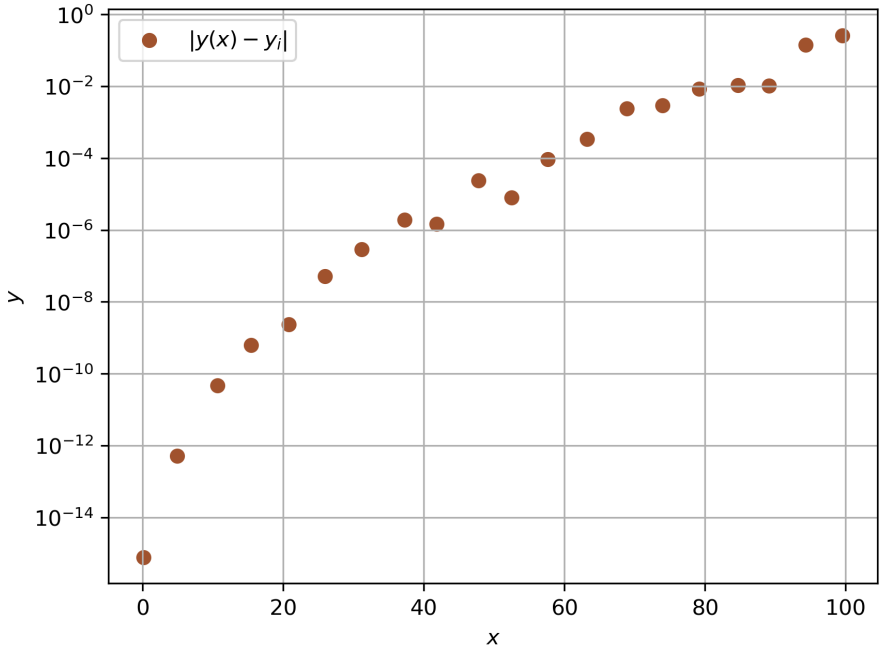

Figure 2: The absolute difference between the actual and the interpolated values. To perform the interpolation we used the Vandermonde matrix and LU decomposition to solve the system. We see that the precision at the first points is great but gets increasingly worst for bigger values of x.

In Figure 2 we plot the absolute difference between the given points and the predictions at the same position (x). By doing this we can check if the polynomial is lying exactly on top of our data. As we can see the polynomial is very accurate for the first few data points and gets increasingly less precise for the last points. Still, the deviation is smaller than 1 while the range of our dataset is $[-180.8, 150.2]$.

2.2 Neville's Algorithm

```
import os , sys
import numpy as np
import matplotlib.pyplot as plt
from lu_decomp import Vandermonde, ForwSub, BackSub, LUDecomp, PredictVander

def Neville(x_pred: np.ndarray, x_true: np.ndarray, y_true: np.ndarray, degree: int) ->
    np.ndarray:
    '''
    applies the Neville's algorithm to find the Lagrange polynomial
    for each point we use the bisection algorithm to find the m nearest neighbors
    by finding j_low and we initialize p with these points
    we then loop and each time we update the values of p
    at the end p[0] is the interpolated value at this point
    '''
    m=degree+1
    nev=np.zeros_like(x_pred)
    for i,x in enumerate(x_pred):
        low=0
        up=len(x_true)-1
        while low+1<up:
            if x<x_true[int((up+low)/2)]:
                up=int((up+low)/2)
```

```
            else:
                low=int((up+low)/2)
        j_low=low-int((m-1)/2)
        if j_low<0:
            j_low=0
        if j_low+m>len(y_true):
            j_low=len(y_true)-m
        p=[y_true[i] for i in range(j_low,j_low+m)]
        for k in range(1,m):
            for j in range(m-k):
                p[j] = ((x-x_true[j_low+j+k])*p[j] + (x_true[j_low+j]-x)*p[j+1])/(x_true[j_low+j]-x_true[j_low+j+k])
        nev[i]=p[0]
    return nev


if __name__ == "__main__":
    # get the data
    data = np.genfromtxt(os.path.join(sys.path[0],"data/Vandermonde.txt"),comments='#',dtype=np.float64)
    x = data[:,0]
    y = data[:,1]
    xx = np.linspace(x[0],x[-1],1001)   # x values to interpolate at

    # solve for the method using the vandermonde matrix like we did in the previous code
    vander = Vandermonde(x)
    l , u = LUDecomp(vander)
    u_c = ForwSub(l,y)
    c = BackSub(u,u_c)
    y_van = PredictVander(xx,c)

    y_nev = Neville(xx,x,y,len(x)-1)        # solve using the Neville's algorithm

    # plot the polynomials with the data points
    fig_1 = plt.figure()
    plt.scatter(x,y,label='data',color='black')
    plt.plot(xx,y_van,label='Vandermonde',color='orange')
    plt.plot(xx,y_nev,linestyle='--',label='Neville',color='green')
    plt.xlim(-1,101)
    plt.ylim(-400,400)
    plt.xlabel('$x$')
    plt.ylabel('$y$')
    plt.legend()
    plt.savefig('./plots/compare.png', bbox_inches='tight', dpi=300)
    plt.close()

    y_2 = Neville(x,x,y,len(x)-1)       # generate predictions in the same positions with the data points

    # plot the absolute difference between predictions and actual points
    # second plot is in logscale to show the small differences in the first points
    fig_2 = plt.figure()
    plt.scatter(x,abs(y-y_2),label=r'$|y(x)-y_i|$',color='crimson')
    plt.xlabel('$x$')
    plt.ylabel('$y$')
    plt.legend()
    plt.grid()
    plt.yscale('log')
    plt.savefig('./plots/nevil_dif.png', bbox_inches='tight', dpi=300)
    plt.close()
```

neville.py

We want to confirm that the polynomial we calculated in Section 2.1 is the same as the Lagrange polynomial that we can find when applying Neville's algorithm. To do that we define a new function "nevil" that applies this algorithm on a set of points (x_true, y_true). The input should also include the x values we want to interpolate at and the degree of the outcome polynomial.

The way the function works is that it finds j_low by using the bisection algorithm. The j_low variable keeps the position of the lowest of the m nearest neighbors where m is the degree of the polynomial +1.

The function then initializes an array with the m nearest neighbors and updates using two nested loops. The final value we will interpolate for each point is given by the updated value of the first position of the array. This process is repeated for every single point we want to interpolate for.

In the main body of our script, we load our data as before and also apply the method we mentioned in the previous task (Section 2.1) to calculate the polynomial with the Vandermonde matrix. We then apply our "nevil" function for $n = \text{len}(x) - 1$ to find the full Lagrange polynomial and interpolate on the same x values. We plot both polynomials with the original data points. We also plot the absolute difference between the interpolated and the real values as we did in the previous code.

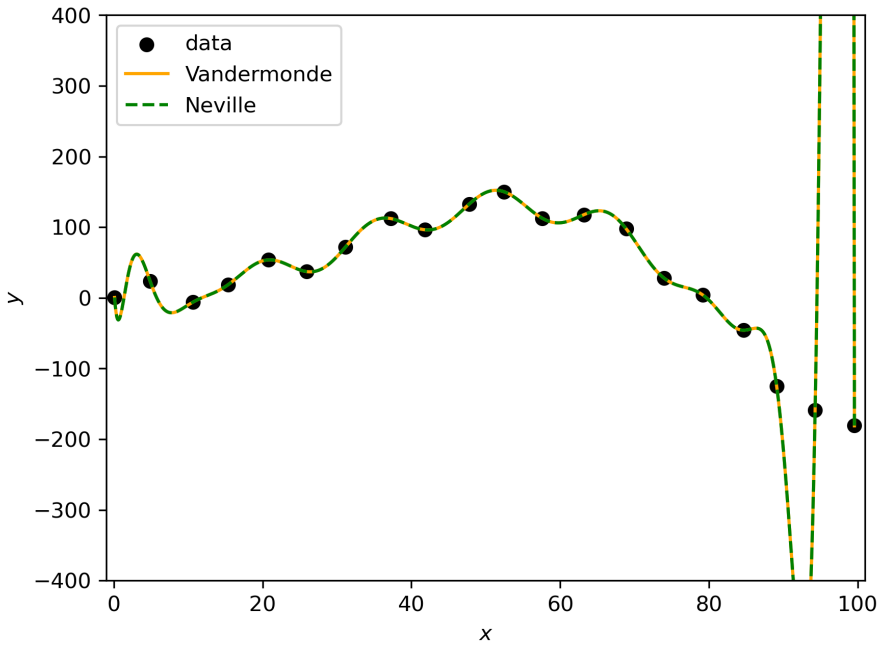

Figure 3: The data points with the full Lagrange polynomial that passes through them as it was calculated using two different approaches. With orange we show the method we implemented in Section 2.1 that uses the Vandermonde matrix. With the dashed green line, we show Neville's method which we implemented for this task. We used a dashed line to show how both of the approaches give us the same polynomial as they fall on top of each other.

In Figure 3 we can see that the two different approaches to calculating the Lagrange polynomial give us the same solution as we would expect. The two plots lie on top of each other and pass through all the data points. In Figure 4 we have a closer look at the absolute difference between the interpolated (with Neville's algorithm) and the true values for all the points. We can achieve higher precision with this approach as these differences are smaller compared with Figure 2 where we used the Vandermonde matrix.

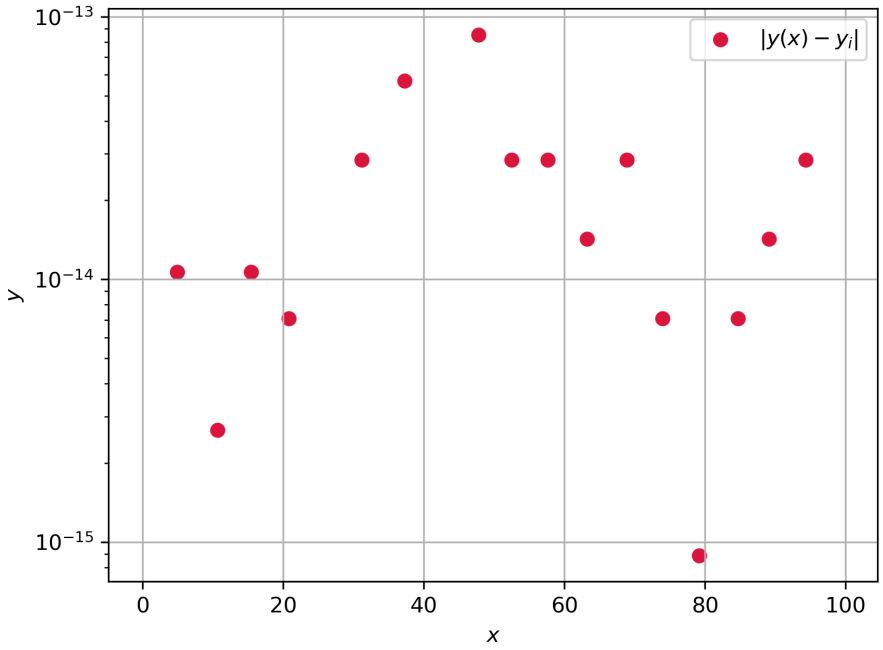

Figure 4: The absolute difference between the actual and the interpolated values. To perform the interpolation we used Neville's algorithm. We see that we managed to achieve great precision for all the data points.

2.3 Iterative Improvements

```python
import os , sys
import numpy as np
import matplotlib.pyplot as plt
from lu_decomp import Vandermonde, ForwSub, BackSub, LUDecomp, PredictVander
from neville import Neville

def LUIter(x_pred: np.ndarray, x: np.ndarray, y: np.ndarray, n: int) -> np.ndarray:
    '''
    interpolate using the vandermonde matrix
    first we define the vandermonde matrix
    we then use the LU decomposition method to get the solution
    we iterate on the solution to get better results
    '''
    vander=Vandermonde(x)
    l , u = LUDecomp(vander)
    b = y
    u_c = ForwSub(l,b)
    c = BackSub(u,u_c)
    for i in range(n-1):
        b = np.dot(vander,c) - y
        u_c = ForwSub(l,b)
        c -= BackSub(u,u_c)
    return PredictVander(x_pred,c)


if __name__ == "__main__":
    # get the data
    data = np.genfromtxt(os.path.join(sys.path[0],"data/Vandermonde.txt"),comments='#',
    dtype=np.float64)
    x = data[:,0]
    y = data[:,1]
    xx = np.linspace(x[0],x[-1],1001)    # x values to interpolate at

    y_pred_1 = LUIter(xx,x,y,1)          # generate predictions using 1 iteration
```

```
34
35      y_pred_10 = LUIter(xx,x,y,10)        # generate predictions using 10 iterations
36
37
38      # plot the polynomial with the data points
39      fig_1 = plt.figure()
40      plt.scatter(x,y,label='data',color='black')
41      plt.plot(xx,y_pred_1,label='1 iteration',color='purple')
42      plt.plot(xx,y_pred_10,linestyle='--',label='10 iterations',color='yellow')
43      plt.xlim(-1,101)
44      plt.ylim(-400,400)
45      plt.xlabel('$x$')
46      plt.ylabel('$y$')
47      plt.legend()
48      plt.savefig('./plots/iter_comp.png', bbox_inches='tight', dpi=300)
49      plt.close()
50
51      # generate predictions in the same positions with the data points
52      y_2 = LUIter(x,x,y,1)
53      y_2_10 = LUIter(x,x,y,10)
54
55      # plot the absolute difference between predictions and actual points
56      fig_2 = plt.figure()
57      plt.scatter(x,abs(y-y_2),label=r'$|y_{1}(x)-y_i|$',color='olive')
58      plt.scatter(x,abs(y-y_2_10),label=r'$|y_{10}(x)-y_i|$',color='teal')
59      plt.xlabel('$x$')
60      plt.ylabel('$y$')
61      plt.legend()
62      plt.grid()
63      plt.yscale('log')
64      plt.savefig('./plots/iter_dif.png', bbox_inches='tight', dpi=300)
65      plt.close()
```

iteration_lu.py

We now want to use the LU decomposition to iterate on the solution in an attempt to improve the results we got from the Vandermonde matrix method. To do that we create a new function "lu_iter" that uses all the functions we used to get the interpolated results by using the Vandermonde matrix. For $n = 1$ the function is finding the solution in the same way that was described in Section 2.1. For $n > 1$ we have more iterations over our solutions by solving the system $LU\delta c = Vc - y$ where V is the Vandermonde matrix and c the array with our initial solution. When we find δc we can update our solutions by calculating $c_{new} = c - \delta c$

In our script, we simply use our function to generate the interpolated predictions using $n = 1$ to get the results we had in Section 2.1 and $n = 10$ to get the new results for 10 iterations. We also plot the absolute difference between the interpolated and the real values, this time for both 1 and 10 iterations at the same figure.

In Figure 5 we can see how the two polynomials calculated for 1 and 10 iterations are on top of each other as we would expect. To distinguish between them we can look at Figure 6 where we have the absolute differences at the data points. We can say that increasing the number of iterations did not achieve better results as we would expect, but rather both approaches had the same deviations from the true values.

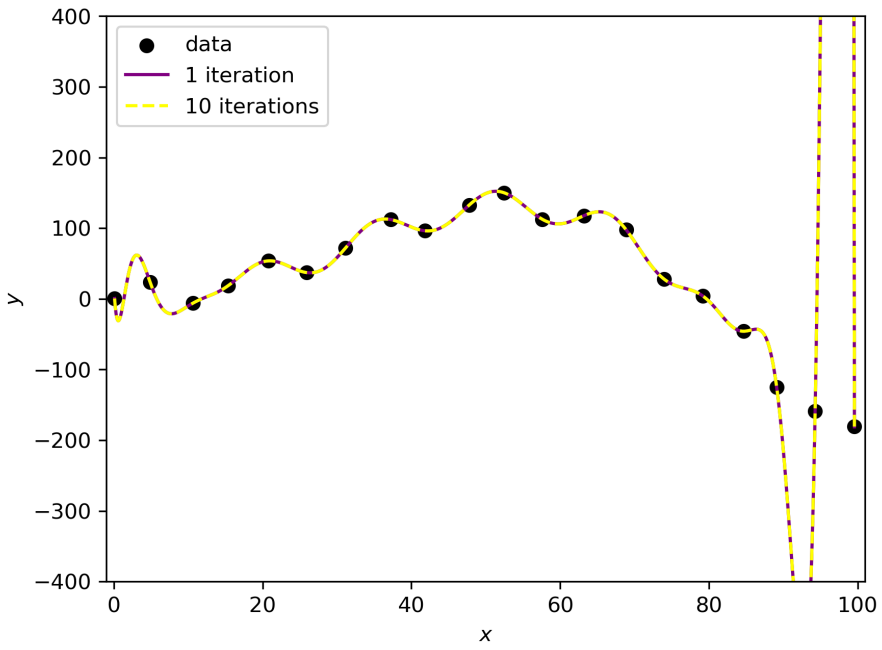

Figure 5: The Lagrange polynomial with the original data points. To generate the polynomial we used the Vandermonde matrix. The system was solved using the LU decomposition method for 1 (purple) and 10 (yellow) iterations. We notice that the polynomials fall on top of each other and cross all the points in our dataset.

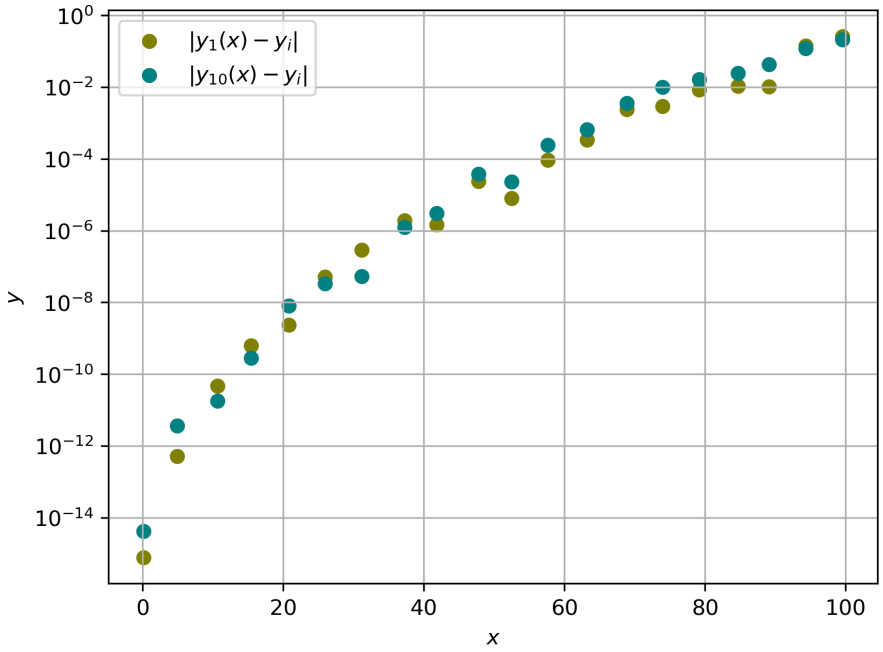

Figure 6: The absolute difference between the actual and the interpolated values. To perform the interpolation we used the Vandermonde matrix and LU decomposition to solve the system with 1 iteration (green) and 10 iterations (blue). For both approaches the precision at the first points is great but gets increasingly worst for bigger values of x. The increased number of iterations did not yield better precision.

2.4 Timing all executions

```python
import os , sys
import timeit
import numpy as np
import matplotlib.pyplot as plt


if __name__ == "__main__":
    # get the data
    data = np.genfromtxt(os.path.join(sys.path[0],"data/Vandermonde.txt"),comments='#',
    dtype=np.float64)
    x = data[:,0]
    y = data[:,1]
    xx = np.linspace(x[0],x[-1],1001)   # x values to interpolate at

    # time all the different ways of generating the Lagrange polynomial and print the
    results
    print(f'Time needed for 100 repetitions of each algorithm in seconds')

    # use 1000 repetitions then divide by 10 to find the time needed for 100 repetitions
    time_iter_1 = timeit.timeit("pred = LUIter(xx,x,y,1)", "from iteration_lu import
    LUIter; from __main__ import x , y, xx",number=10**3)
    time_iter_1 /= 10
    print(f'Using LU decomposition of Vandermonde matrix with 1 iteration: {time_iter_1
    :.4f}')

    # Neville's algorithm is much slower so we use 100 repetitions
    time_nevil = timeit.timeit("pred = Neville(xx,x,y,len(x)-1)", "from neville import
    Neville; from __main__ import x , y, xx",number=10**2)
    print(f'Using Neville algorithm: {time_nevil:.4f}')

    # use 1000 repetitions then divide by 10 to find the time needed for 100 repetitions
    time_iter_10 = timeit.timeit("pred = LUIter(xx,x,y,10)", "from iteration_lu import
    LUIter; from __main__ import x , y, xx",number=10**3)
    time_iter_10 /= 10
    print(f'Using LU decomposition of Vandermonde matrix with 10 iterations: {
    time_iter_10:.4f}')

    fig1 = plt.figure()
    bars=plt.barh(['LU decomposition \n1 iteration','Neville\'s algorithm','LU
    decomposition \n10 iterations'],[time_iter_1,time_nevil,time_iter_10],height=.5,
    color='magenta')
    plt.xlabel('Time [seconds]')
    plt.legend(labels = ['Time needed for\n 100 repetitions'])
    plt.bar_label(bars,fmt='%.2f')
    plt.xlim(0,18)
    plt.savefig('./plots/time.png', bbox_inches='tight', dpi=300)
    plt.close()
```

time_1.py

In this final task, we want to time the three methods we developed in this section. To achieve that we use the "timeit" module. When we call the "timeit" function of that module we can specify the number of repetitions and we also need to import all the necessary functions and data. We will time the "nevil" function from Section 2.2 and the "lu_iter" function from Section 2.3 for $n = 1$, 10 to get the time for 1 and 10 iterations. We are interested in finding the time needed for 100 repetitions of each algorithm. Because "lu_iter" is much faster we can run it for 1 000 repetitions and divide the result by 10 to make our results more accurate.

```
Time needed for 100 repetitions of each algorithm in seconds
Using LU decomposition of Vandermonde matrix with 1 iteration: 1.1621
Using Neville algorithm: 16.3392
Using LU decomposition of Vandermonde matrix with 10 iterations: 1.2770
```

output/time_1.txt

The output of our code shows the time needed for 100 repetitions of each of the methods we used to calculate the Lagrange polynomial in this assignment.

The difference in the execution time comes from the way these algorithms operate and were implemented. For Neville's algorithm, we use two nested loops to update an array for each of the points we wanted to interpolate. All these loops are very costly in terms of execution time. The LU decomposition of the Vandermonde matrix is implemented with only a single loop and uses matrix multiplications to perform all the calculations. This is the reason it is much faster than Neville's algorithm. The slowest part of the function "lu_iter" is executing the "predict_vander" function that generates the predictions after we calculated the array of the coefficients. This is the reason why increasing the number of iterations is not affecting the execution time that much, because the predictions are generated once regardless of the number of iterations.

To determine which one should give us the most accurate results we need to compare Figures 4 and 6. As we already discussed, increasing the number of iterations did not improve our results. Neville's algorithm gives us the best results for the whole set of points and can therefore be considered the most accurate implementation for the Lagrange polynomial.

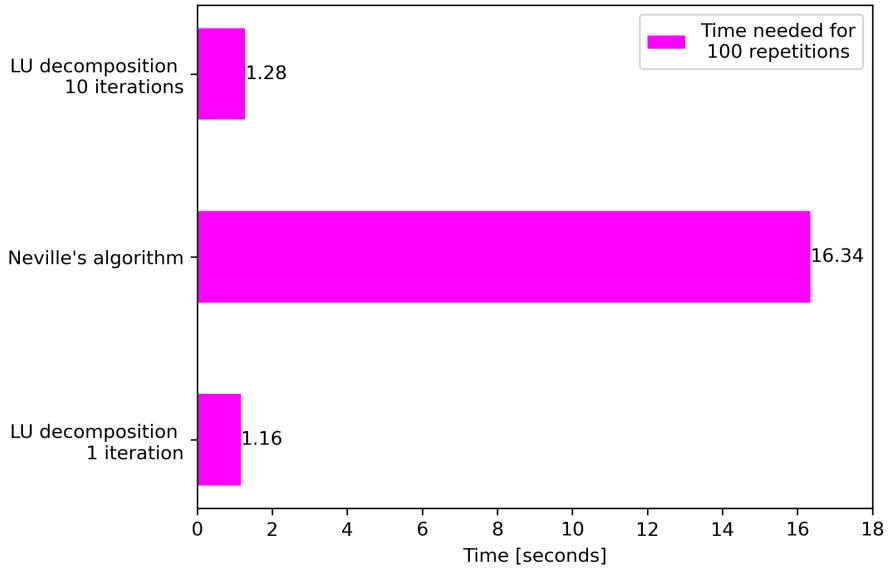

Figure 7: The time needed for 100 repetitions of each of the methods used to calculate the Lagrange polynomial. Neville's algorithm was much slower than the Vandermonde matrix with the LU decomposition. Increasing the number of iterations leads to only a small increase in the execution time.