NUR Hand-in Exercise 1

Angèl Pranger

February 26, 2025

Abstract

This document shows my solutions to hand-in exercise 1 of numerical methods in astrophysics.

1 Poisson distribution

For this exercise, I was asked to write a function that returns the Poisson probability distribution $P_\lambda(k) = \frac{\lambda^k e^{-\lambda}}{k!}$ for integers k given a positive mean λ.

It seems there are three potential problems to tackle. Firstly, there is the possibility of underflow for large values of k, caused by the division by a very large number $k!$. Secondly, there is the factor λ^k, which can cause overflow for large values of λ and/or k. Thirdly, there is the factor $e^{-\lambda}$, which can become too small for large values of λ.

I implemented three functions, which all compute the Poisson probability distribution in a different way. The first function only uses a conversion to log space, such that

$$\begin{aligned} P_\lambda(k) &= \exp(\ln(\frac{\lambda^k e^{-\lambda}}{k!})) \\ &= \exp(\ln(\lambda^k) + \ln(e^{-\lambda}) - \ln(k!)) \\ &= \exp(k\ln(\lambda) - \lambda - \sum_{i=0}^{k-1} \ln(k-i)). \end{aligned}$$

The second function first uses not log space but a different order of operations, given by

$$P_\lambda(k) = e^{-\lambda} \cdot \frac{\lambda}{k} \cdot \frac{\lambda}{k-1} \cdot \dots \cdot \frac{\lambda}{1}.$$

The third function uses a combination of this different order of operations and log space, such that

$$P_\lambda(k) = \exp(\ln(\frac{\lambda}{k} \cdot \frac{\lambda}{k-1} \cdot \dots \cdot \frac{\lambda}{1}) - \lambda).$$

The script in which these functions are implemented is:

```python
import numpy as np

# Using log conversion
def poisson1(lamda:np.float32, k:np.float32) -> np.float32:
    """Computing the poisson distribution value for parameters (lamda, k) using log conversion."""
    j = np.float32(0)
    for i in range(np.int32(k)):
        i = np.float32(i)
        j += np.float32(np.log(k-i))
    P = np.float32(np.exp(k*np.log(lamda)-lamda-j))
    return P

# Using different order of operations
def poisson2(lamda:np.float32, k:np.float32) -> np.float32:
    """Computing the poisson distribution value for parameters (lamda, k) using an unconventional order of operations."""
```

```
    P = np.float32(np.exp(-lamda))
    for i in range(np.int32(k)):
        i = np.float32(i)
        P *= np.float32(lamda/(k-i))
    return P

# Using different order of operations and log conversion, avoiding np.exp(-lamda)
def poisson3(lamda:np.float32, k:np.float32) -> np.float32:
    """Computing the poisson distribution value for parameters (lamda, k) using a
    different order of operations and a log conversion."""
    P = np.float32(1)
    for i in range(np.int32(k)):
        i = np.float32(i)
        P *= np.float32(lamda/(k-i))
    P = np.float32(np.exp(np.log(P) - lamda))
    return P

# Computing and printing results
lamda_k = np.array([[1,0],[5,10],[3,21],[2.6,40],[100,5],[101,200]], dtype=np.float32)
print("[lamda k] P1 P2 P3")
for values in lamda_k:
    P1 = poisson1(values[0], values[1])
    P2 = poisson2(values[0], values[1])
    P3 = poisson3(values[0], values[1])
    print(f"[{values[0]:.1f} {values[1]:.0f}] {P1:.6} {P2:.6} {P3:.6}")
```

1_poisson_distribution.py

The results are given by the following. The first column states the values for λ and k, the columns named $P1, P2, P3$ give the results for the first, second and third function respectively.

```
[lamda k] P1 P2 P3
[1.0 0] 0.367879 0.367879 0.367879
[5.0 10] 0.0181328 0.0181328 0.0181328
[3.0 21] 1.01934e-11 1.01934e-11 1.01934e-11
[2.6 40] 3.61508e-33 3.61512e-33 3.61513e-33
[100.0 5] 3.10006e-36 3.15292e-36 3.10006e-36
[101.0 200] 1.26953e-18 1.07434e-05 1.26953e-18
```

poisson_distribution.txt

We see that the results of the three functions are the same up to 6 significant digits for the first three values of λ, k. This is to be expected, as the corresponding values of λ and k are relatively small and thus no over or underflow is expected for any of the functions. For the last two rows, where λ is relatively large, we find that the first and third function give the same result, while the result of the second function deviates. As potential under and overflow errors should not be exactly the same for the first and third function, this leads us to think that these functions actually give the correct answer (at least up to these 6 significant digits), while the second function then gives an incorrect result. The problem for the second function is here in the factor $e^{-\lambda}$, which for $\lambda = 100$ is of order 10^{-44} and can thus not be stored in a 32 bit float. The values $\lambda, k = 2.6, 40$ gives a slightly different result for each of the three functions. According to WolframAlpha, the result of the second function is actually the most accurate here, after which the result of the third function is a close second. It is not clear to us why the result of the first function deviates here, but this might have something to do with the fact that λ is relatively small while k is relatively large (we found in class that this value of k already gives problems with the original implementation of the Poisson distribution), or with the fact that $\lambda = 2.6$ is not an integer.

In conclusion, the third function gives overall the best result.

2 Vandermonde matrix

The code of any shared modules for question 2 is given by:

```
import numpy as np
import timeit
import sys
```

```
import os
import matplotlib.pyplot as plt

# Importing data
data=np.genfromtxt(os.path.join(sys.path[0],"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

# Functions used
def LU_decomposition(matrix:np.ndarray):
    """Takes a matrix and computes its LU-decomposition.
    Returns the same matrix which now holds the LU-decomposition: the upper triangle and
     diagonal hold the beta_ij (U),
    while the lower triangle hold the alpha_ij (L). The alpha_ii are chosen to be equal
    to 1."""
    # Choose alpha_ii = 1
    # Loop over the columns j
    for j in range(matrix.shape[0]):
        # Keep beta_0j = a_0j
        # Update beta_ij where 0<i<=j
        for i in range(1, j+1):
            term = np.float64(0)
            for k in range(0, i):
                term += matrix[i,k]*matrix[k,j]
            matrix[i,j] -= term
        # Update alpha_ij where i>j
        beta_jj = 1/matrix[j,j]
        for i in range(j+1, matrix.shape[0]):
            term = np.float64(0)
            for k in range(0, j):
                term += matrix[i,k]*matrix[k,j]
            matrix[i,j] = (matrix[i,j]-term)*beta_jj
    return np.float64(matrix)

def solving_system_with_LU(LU:np.ndarray, b:np.array) -> np.array:
    """Solves the system Ax=b by taking the LU decomposition of A and b as input and
    applying forward and backward substitution.
    Returns the vector b which now holds the solution x."""
    # Knowing that alpha_ii = 1
    # Forward substitution
    # Keep y_0 = b_0 as alpha_00 = 1
    for i in range(1, LU.shape[0]):
        term = np.float64(0)
        for j in range(i):
            term += LU[i,j]*b[j]
        b[i] -= term
    # Backward substitution
    b[-1] /= LU[-1,-1]
    for i in range(LU.shape[0]-1):
        i = LU.shape[0]-1 - i
        term = np.float64(0)
        for j in range(i+1, LU.shape[0]):
            term += LU[i,j]*b[j]
        b[i] = (b[i]-term)/LU[i,i]
    return np.float64(b)

def bisection(M:np.int64, x_samp:np.array, x_interp:np.array) -> np.ndarray:
    """Takes as input a number of points M, a sample array x and an interpolation values
     array x_interp.
    Uses bisection to find the closest one or two points in x_samp for each point in
    x_interp. Subsequently finds the closest M
    points in x_samp and returns the index in x_samp of the lowest (leftmost) point."""
    # Assumes x_samp are uniformly spaced
    if (M > x_samp.shape[0]):
        print("M is too large for the array x_samp")
        return
    # Loop over all points x in x_interp
    for (i,x) in enumerate(x_interp):
```

```
68          # Set left and right edge indices within x_samp
69          l_idx = np.int64(0)
70          r_idx = np.int64(x_samp.shape[0]-1)
71          # Half the searching area, check in which half x lies, update edge indices for
        this half,
72          # until the left and right edge index differ by 0 or 1
73          while (r_idx - l_idx > 1):
74              m_idx = np.int64(np.floor((l_idx + r_idx)*0.5))
75              if (x <= x_samp[m_idx]):
76                  r_idx = m_idx
77              elif (x > x_samp[m_idx]):
78                  l_idx = m_idx
79          # Save the middle value of the current interval to later check whether x is
        closer to the left or right edge
80          m = (x_samp[l_idx] + x_samp[r_idx])*0.5
81          # Computing the left boundary index (ignoring range)
82          l_idx -= np.int64(np.ceil(0.5*M-1))
83          # Correcting left boundary if x is closer to the right boundary
84          # (in which case the right boundary is seen as the first point instead of the
        second)
85          if ((x > m) & (M % 2 == 1)):
86              l_idx += np.float64(1)
87          # Checking range (ensuring the left and right edge do not fall outside x_samp)
88          if l_idx < 0:
89              l_idx = 0
90          if (l_idx + M >= x_samp.shape[0]):
91              l_idx -= l_idx + M - x_samp.shape[0]
92          x_interp[i] = l_idx
93      return np.int64(x_interp)
94
95  def neville(M:np.int64, x_samp:np.array, y_samp:np.array, x_interp:np.array) -> np.
        ndarray:
96      """Takes an integer number of points M, sample array x_samp with corresponding y
        values y_samp and interpolation values x_interp.
97      Computes for each x in x_interp the value of the unique Lagrange polynomial through
        the M points in x_samp closest to x, using Neville's algorithm.
98      Returns the M-1 order Lagrange polynomial values corresponding to the x in x_interp
        and the corresponding error estimates."""
99      if (M < 1):
100         print("M is too small")
101         return
102     # Finding left index of M tabulated points using bisection
103     l_indices = bisection(M, x_samp.copy(), x_interp.copy())
104     # Initializing matrix P for Neville's algorithm (each row corresponds to a x in
        x_interp, each column corresponds to one of the M closest sample points)
105     Ps = np.zeros((x_interp.shape[0], M), dtype=np.float64)
106     for i in range(M):
107         Ps[:,i] = y_samp[l_indices+i]
108     # Looping over orders k and current intervals
109     for k in range(1, M):
110         for i in range(M-k):
111             j = i+k # note this j is only for the x_samp, as Ps is overwritten so just j
        =i+1 as index
112             # Ps[:,i] are overwritten to hold the linear interpolation between the
        previous values
113             Ps[:,i] = ((x_samp[l_indices+j]-x_interp)*Ps[:,i]+(x_interp-x_samp[l_indices
        +i])*Ps[:,i+1])/(x_samp[l_indices+j]-x_samp[l_indices+i])
114     if (M < 2): # No error estimate
115         return Ps[:,0], 0
116     error_estimate = np.absolute(Ps[:,0]-Ps[:,1])
117     return Ps[:,0], error_estimate
118
119 def polynomial(c:np.array, x:np.array) -> np.array:
120     """Constructing the values y of the polynomial with coefficients given by c at
        points x.
121     Returns y values."""
122     y = np.float64(0)
123     for j in range(c.shape[0]):
124         y += c[j] * np.power(x,j)
125     return np.float64(y)
```

2_vandermonde_matrix.py

2.1 a

For this exercise, I wrote a code that firstly constructs the Vandermonde matrix V for the 20 data points x using that $V_{ij} = x_i^j$. Subsequently, the LU decomposition of V was found and the system $Vc = y$, for which the data points y where given, was solved for c. The coefficients c of the corresponding 19-th degree polynomial through the data points (x, y) where then used to compute and plot the values of this polynomial for 1001 equally-spaced points within the initial range of x. The corresponding result is shown in Figure 1 and the computed coefficients c are given as well.

The code used is given by:

```
127 # 2a
128
129 def q2a(x:np.array, y:np.array, xx:np.array) -> np.ndarray:
130     """Constructing Vandermonde matrix V from data x, LU-decomposing V and solving Vc=y
        for c.
131     Returns c and the values of the polynomial, yya and ya, corresponding to xx and x.
        """
132     # Constructing V
133     V = np.zeros((x.shape[0], x.shape[0]), dtype=np.float64)
134     for j in range(V.shape[0]):
135         V[:,j] = np.power(x,j)
136     # LU-decomposing V
137     LU_V = LU_decomposition(V.copy())
138     # Solving for c
139     c0 = solving_system_with_LU(LU_V, y.copy())
140     # Finding the polynomial values y
141     yya = polynomial(c0, xx)
142     ya = polynomial(c0, x)
143     return V, LU_V, c0, yya, ya
144
145 V, LU_V, c0, yya, ya = q2a(x, y, xx)
146
147 # Printing the values of c
148 print("The coefficients of the polynomial found with LU-decomposition are")
149 print(c0)
150
151 # Plot of points with absolute difference shown on a log scale (question 2a)
152 fig=plt.figure()
153 gs=fig.add_gridspec(2,hspace=0,height_ratios=[2.0,1.0])
154 axs=gs.subplots(sharex=True,sharey=False)
155 axs[0].plot(x,y,marker='o',linewidth=0)
156 plt.xlim(-1,101)
157 axs[0].set_ylim(-400,600)
158 axs[0].set_ylabel('$y$')
159 axs[1].set_ylim(1e-18,1e4)
160 axs[1].set_ylabel('$|y-y_i|$')
161 axs[1].set_xlabel('$x$')
162 axs[1].set_yscale('log')
163 line,=axs[0].plot(xx,yya,color='orange')
164 line.set_label('Via LU decomposition')
165 axs[0].legend(frameon=False,loc="lower left")
166 axs[1].plot(x,abs(y-ya),color='orange')
167 plt.savefig('my_vandermonde_sol_2a.png',dpi=600)
```

2_vandermonde_matrix.py

```
1 The coefficients of the polynomial found with LU-decomposition are
2 [ 5.21896930e-01  3.53817596e+03 -2.20652879e+03  6.00128574e+02
3  -9.43549171e+01  9.66286617e+00 -6.87646085e-01  3.53596337e-02
4  -1.34619705e-03  3.84953383e-05 -8.32127939e-07  1.35821060e-08
5  -1.65786631e-10  1.48156916e-12 -9.31152862e-15  3.80148410e-17
6  -8.28186823e-20  2.55562475e-23  1.78743492e-25 -1.28961270e-56]
```

vandermonde_matrix.txt

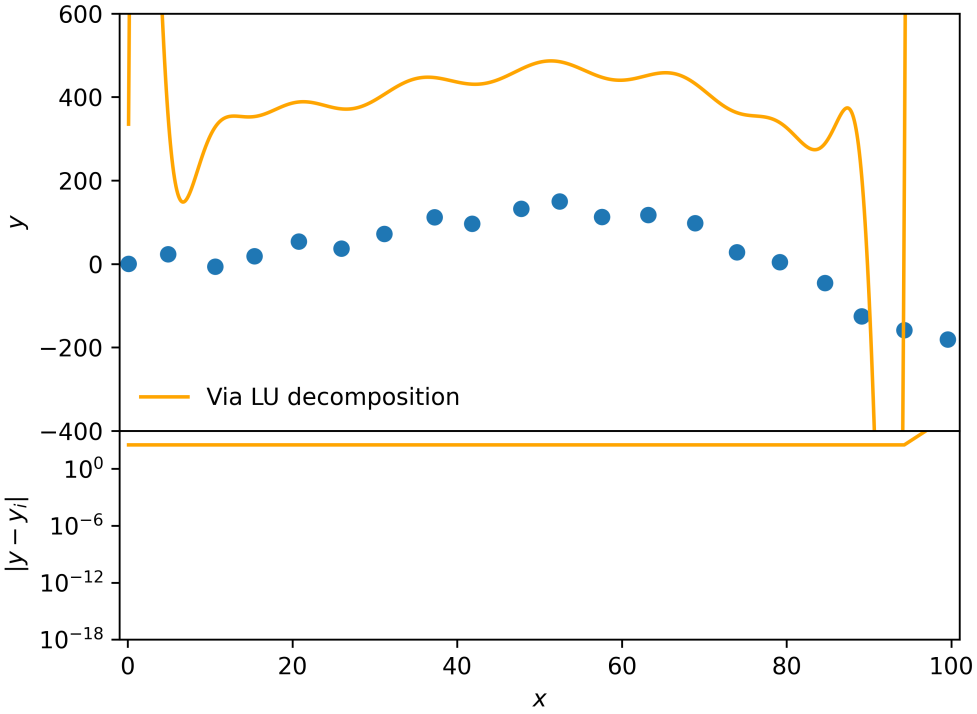

Figure 1: The original data points (x, y) together with the constructed 19-th degree polynomial using the coefficients found by solving the system $Vc = y$ using the LU decomposition of the corresponding Vandermonde matrix are shown. It is clear that the polynomial does not go through the original data points, as it lies way above them. However, the shape of the polynomial does somewhat seem to follow the original data points. It might thus be the case that in particular the offset coefficient c_0 has a large error, causing the polynomial to be shifted this way. The lower panel shows the absolute difference between the polynomial and the data points. As we can see, this offset is about 400 which is very large.

2.2 b

To see whether this polynomial is equal to the Lagrange polynomial, which it should be as they both go through all points and are unique, we compute the Lagrange polynomial using Neville's algorithm on all 20 data points. Although not strictly necessary for this exercise (as M=20 and thus all points are used for the interpolation of every point), we have implemented a bisection algorithm to find the M closest points in the data set to each interpolation value x, as this makes for a more generic Neville's algorithm which can be used for other situations and tested for smaller M. The resulting polynomial is shown in Figure 2.

The code used is given by:

```
169 # 2b
170
171 def q2b(x:np.array, y:np.array, xx:np.array, M:np.int64):
172     """Computing the lagrange polynomial through the given data points (x,y) using
        Neville's algorithm (iff M=x.shape[0]).
173     Returns values yyb and yb of the polynomial corresponding to the points in xx and x
        respectively."""
174     yyb = neville(M, x, y, xx)[0]
175     yb = neville(M, x, y, x)[0]
176     return yyb, yb
177
178 yyb, yb = q2b(x, y, xx, 20)
179
```

```
# For questions 2b and 2c, add this block
line,=axs[0].plot(xx,yyb,linestyle='dashed',color='green')
line.set_label('Via Neville\'s algorithm')
axs[0].legend(frameon=False,loc="lower left")
axs[1].plot(x,abs(y-yb),linestyle='dashed',color='green')
plt.savefig('my_vandermonde_sol_2b.png',dpi=600)
```

2_vandermonde_matrix.py

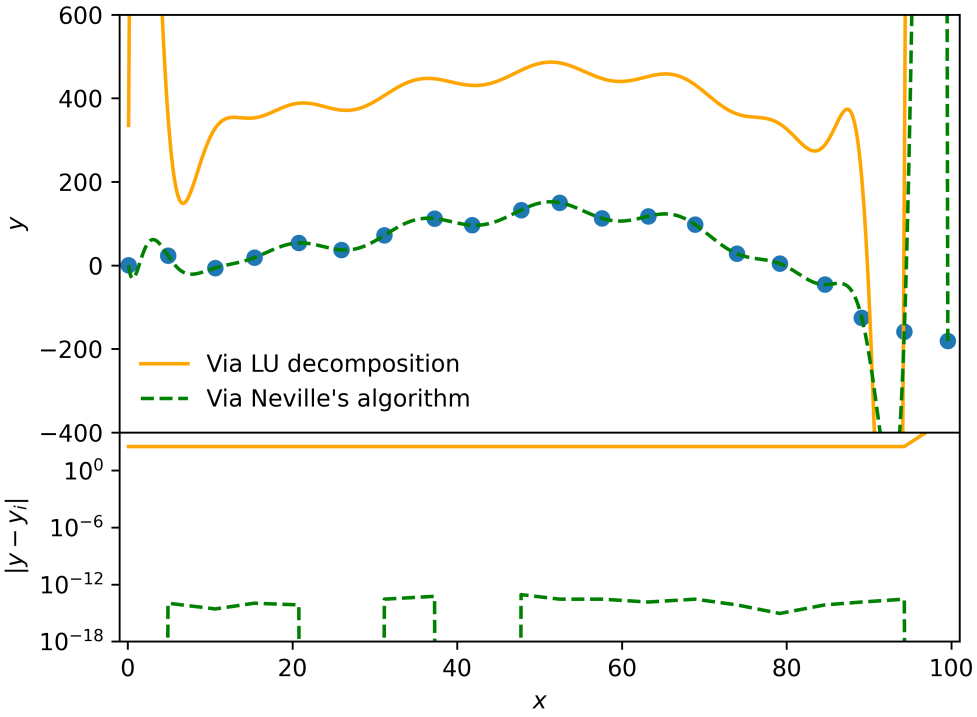

Figure 2: The previous resulting polynomial from LU decomposition is shown together with the Lagrange polynomial found using Neville's algorithm with M=20. We see that the Lagrange polynomial does closely follow the original data points, with a small offset shown in the lower panel of about 10^{-14}. The offset even crashes down for some x values, indicating that the Lagrange polynomial very accurately follows the data points. This is to be expected, as Neville's algorithm uses direct linear interpolation on the data points, and when used on the data points themselves, the result is correct up to round off and machine error. The method using the Vandermonde matrix, however, is a more indirect way of computing the polynomial with more intermediate calculations, which does not compute the polynomial values directly but the polynomial coefficients first. As a small deviation in coefficient value can make a large difference in the polynomial values, this explains why the original LU-decomposition method gives polynomial values with a much larger error than the Lagrange polynomial from Neville's algorithm.

2.3 c

We now try to improve the result for LU decomposition by using iterative improvements. From class we know that $V\delta c = \delta y = Vc - y$. Thus we can now solve this system for δc and get $c = c - \delta c$. We use both 1 and 10 iterational improvements on the coefficients and again plot the resulting polynomials. The results are shown in Figure 3.

The code used is given by:

```
# 2c
```

```python

def q2c(LU_V:np.ndarray, V:np.ndarray, c:np.array, y:np.array, x:np.array, xx:np.array, num:np.int64) -> np.ndarray:
    """Computing num iterative improvements on the solution c of the system Vc=y solved by LU-decomposition.
    Returns values yyc and yc of the polynomial corresponding to the points in xx and x respectively."""
    # Iterative improvements
    for i in range(num):
        c -= solving_system_with_LU(LU_V, (np.matmul(V,c)-y).copy())
    # Computing polynomial values
    yyc = polynomial(c, xx)
    yc = polynomial(c, x)
    return yyc, yc

yyc1, yc1 = q2c(LU_V, V, c0.copy(), y, x, xx, 1)
yyc10, yc10 = q2c(LU_V, V, c0.copy(), y, x, xx, 10)

# For question 2c, add this block too
line,=axs[0].plot(xx,yyc1,linestyle='dotted',color='red')
line.set_label('LU with 1 iteration')
axs[1].plot(x,abs(y-yc1),linestyle='dotted',color='red')
line,=axs[0].plot(xx,yyc10,linestyle='dashdot',color='purple')
line.set_label('LU with 10 iterations')
axs[1].plot(x,abs(y-yc10),linestyle='dashdot',color='purple')
axs[0].legend(frameon=False,loc="lower left")
plt.savefig('my_vandermonde_sol_2c.png',dpi=600)
```

2_vandermonde_matrix.py

2.4 d

Lastly, we use timeit to time the execution times of a, b and c (with 10 iterations). We have set timeit's number parameter to 1000, such that we get a more accurate run time estimate without taking more than a minute to execute.

The code used is given by:

```python
# 2d

# Timing the previous exercises
num = np.int64(1000)
print(f"Running 2a {num} times takes", timeit.timeit(lambda: q2a(x,y,xx), number=num), "seconds.")
print(f"Running 2b {num} times takes", timeit.timeit(lambda: q2b(x,y,xx,20), number=num), "seconds.")
print(f"Running 2c {num} times takes", timeit.timeit(lambda: q2c(LU_V,V,c0.copy(),y,x,xx,10), number=num), "seconds.")
```

2_vandermonde_matrix.py

The resulting run times are:

```
Running 2a 1000 times takes 1.6017186999961268 seconds.
Running 2b 1000 times takes 22.491641599990544 seconds.
Running 2c 1000 times takes 1.7273653000011109 seconds.
```

vandermonde_matrix.txt

We see that question b, using Neville's algorithm, uses most computation time by far. This can be explained by the fact that Neville's algorithm computes the value of the polynomial at each of 1001 different x values, directly interpolating between all the x sample points. This requires multiple calculations for each of the 1001 interpolation points. The method using LU decomposition, however, only needs to compute the LU decomposition once and can then compute the 20 coefficients of the polynomial, which can then be used to easily compute the polynomial value for any x. The LU decomposition method is thus more efficient. However, as we have explained before, Neville's algorithm is more accurate in the sense that it has the smallest offset caused by the fact that the method directly interpolates between the

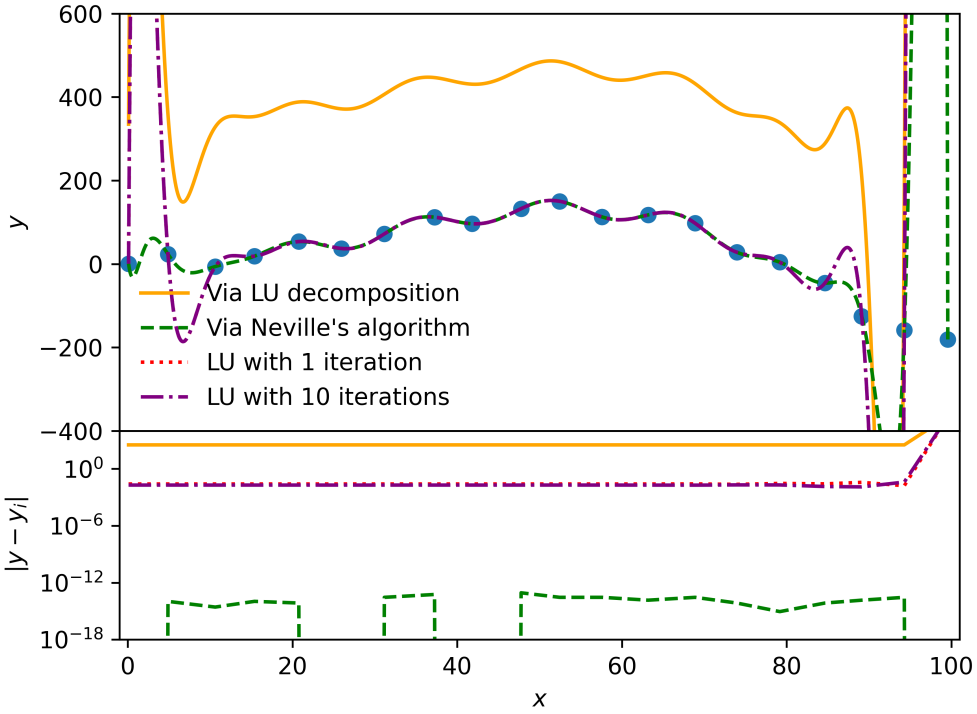

Figure 3: Again, the results from a and b are shown, but the polynomials found using 1 or 10 iterative improvements on the coefficients c are added. We see that the first iterative improvement already greatly improves the polynomial found with LU decomposition: it now follows the data points just as the Lagrange polynomial. However, we do see that the offset is still larger for the improved LU polynomial compared to Neville's Lagrange polynomial. As explained previously, this can be explained by the fact that Neville's algorithm directly computes the polynomial values, while the LU decomposition method computes the coefficients of the polynomial after which those are used to compute the values. This indirect way allows for more error to accumulate. Furthermore, we see that using 10 iterative improvements compared to only 1 does not really improve the result any more. Lastly, we note that all polynomials behave quite normally in the middle of the range of x data points, but more chaotic around the edges of the data points. For example, the LU decomposition polynomials do not even go through the last data point at all. It might be the case that the actual polynomial we are looking for actually behaves more chaotically around the edges, but this can also be a result of the way in which the polynomial is calculated. Especially in the case of Neville's algorithm, the edge points are different from the middle points, because there is no information on what is going on with the polynomial outside the edges.

existing data points. The LU decomposition yield a larger error because it computes the coefficients of the polynomial and not the values of the polynomial directly, causing errors to accumulate and allowing for example the found polynomial to not go through the last data point at all.