

# NUR - Hand-in Exercise 1

Martina Cacciola (s4170814)

March 7, 2024

## Abstract

In this document, I present the results for the Hand-in Exercise 1 for the course Numerical Recipes for Astrophysics.

## 1 Poisson distribution

The exercise is done in the script

```
1 #!/usr/bin/env python
2 import numpy as np
3 from math import exp, log10
4
5 '''
6 This script contains the code for Exercise 1.
7 '''
8
9 def poisson_prob(l, k):
10     '''
11     Calculates the Poisson probability for a given lambda and k
12     At each passage, we work with 32 bit floats and in log space to avoid overflow.
13     By taking the logarithm, we transform multiplications into additions and divisions
14     into subtractions.
15     This helps keep the intermediate results within a manageable range.
16     We then take the exponent of the final result, ensuring both numerical stability and
17     correct results.
18
19     Inputs:
20     l: lambda value
21     k: k value
22     Output:
23     Poisson probability for a given lambda and k
24
25     We divide the calculation into:
26     1. Calculate  $\lambda^k$  in log space
27     2. Calculate  $e^{-\lambda}$ 
28     3. Calculate  $k!$  and the cumulative sum of  $\log(k!)$  for all  $k$  from 2.
29     The reason for starting from 2 is that the first two values of  $k!$  are 0 and 1, not
30     useful for the calculation.
31     The cumulative sum is useful for next calculation
32     4. Calculate the Poisson probability in log space and convert it back to normal
33     space.
34
35     '''
36
37     k_values = np.float32(np.arange(k + 1))
38
39     # 1.  $\log(\lambda^k)$ 
40     lam_k_log = np.float32(k_values * log10(l))
41     # 2.  $e^{-\lambda}$ 
42     e_neg_lambda = np.float32(log10(exp(-l)))
43     # 3. Factorial calculation:  $\log(k!)$  and cumulative sum of  $\log(k!)$  for all  $k$  from 2
44     log_fact_k = np.float32(np.log10(np.arange(2, k + 1)))
45     tot_log_fact = np.float32(np.cumsum(log_fact_k))
46     # 4. Poisson probability in log space and conversion
```

```

43     log_p_k = np.float32(lam_k.log[2:] + e_neg_lambda - tot_log_fact)
44     # Terms k=0,1 are calculated separately to maintain accuracy
45     # The remaining k are obtained by exponentiating the logarithmic Poisson
46     # probabilities
47     poisson_distribution = np.float32(np.concatenate([[(1**k) * exp(-1)] for k in
48     k_values[:2]] + [np.power(10, log_p_k)]))
49
50     return poisson_distribution
51
52 # Test the function for the given values
53 parameter_values = [(1, 0), (5, 10), (3, 21), (2.6, 40), (101, 200)]
54
55 # Save Poisson probability results to a text file
56 output_data = []
57 for lambda_val, k_val in parameter_values:
58     poisson_distribution = poisson_prob(lambda_val, k_val)
59     output_data.append([lambda_val, k_val, poisson_distribution[int(k_val)]])
60
61 np.savetxt('poisson_output.txt', np.array(output_data), header='Lambda\tK\tPoisson
62     Probability', delimiter='\t')

```

poisson.py

. The necessary explanations are in the comments of the code. The output produced for the values required is in

```

1 # Lambda    K    Poisson Probability
2 1.0000000000000000e+00    0.0000000000000000e+00    3.678794503211975098e-01
3 5.0000000000000000e+00    1.0000000000000000e+01    1.813277788460254669e-02
4 3.0000000000000000e+00    2.1000000000000000e+01    1.019341815816643404e-11
5 2.6000000000000000e+00    4.0000000000000000e+01    3.615135162986033128e-33
6 1.0100000000000000e+02    2.0000000000000000e+02    1.269425385826567513e-18

```

poisson\_output.txt

## 2 Vandermonde matrix

In this section, we add the comments to the plots produced by the script given by:

```

1 #!/usr/bin/env python
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import timeit
5
6 '''
7 This script contains the code for Exercise 2.
8 '''
9
10 '''
11 Point a):
12
13 We implement the LU decomposition using Crout's algorithm to solve the Vandermonde
14     system.
15 The problem is divided into:
16
17 1. LU decomposition: the matrix A is decomposed into a lower triangular matrix L and an
18     upper triangular matrix U.
19 Firstly we initialize L and U as zero matrices of the same size as A.
20 Setting alpha_ii is done implicitly in passage 1:
21 since L[i, :i] and U[:i, i] are both zero vectors at this point this dot product is zero
22     , and so U[i, i] is just A[i, i].
23 Then we update the elements of L and U loop over the columns.
24
25 2. Forward substitution: we solve Ly = b for y.
26 We initialize y as a zero vector of the same size as b.
27 Then we solve the system iteratively for each element of y starting from first element.

```

```

26
27 3. Backward substitution: we solve  $Ux = y$  for  $x$ .
28 We initialize  $x$  as a zero vector of the same size as  $y$ .
29 Then we solve the system iteratively for each element of  $x$  starting from the last
    element
30 '''
31
32 # LU Decomposition
33 # beta_ij corresponds to  $U[i, j]$ , and alpha_ij corresponds to  $L[i, j]$ 
34
35 def lu_decomposition(A):
36     n = len(A)
37     L = np.zeros((n, n))
38     U = np.zeros((n, n))
39
40     # Loop over the columns
41     for i in range(n):
42         # Update the upper and lower triangular matrices
43         U[i, i:] = A[i, i:] - np.dot(L[i, :i], U[:i, i:]) # Passage 1
44         L[:, i] = (A[:, i] - np.dot(L[:, :i], U[:i, i])) / U[i, i] # Passage 2
45
46     return L, U
47
48 # Forward substitution
49 def forward_substitution(L, b):
50     n = L.shape[0]
51     y = np.zeros_like(b, dtype=np.double)
52     # Solve  $Ly = b$  for  $y$ 
53     y[0] = b[0] / L[0, 0]
54     for i in range(1, n):
55         y[i] = (b[i] - np.dot(L[i, :i], y[:i])) / L[i, i] # Passage 3
56     return y
57
58 # Backward substitution
59 def backward_substitution(U, y):
60     n = U.shape[0]
61     x = np.zeros_like(y, dtype=np.double)
62     # Solve  $Ux = y$  for  $x$ 
63     x[-1] = y[-1] / U[-1, -1]
64     for i in range(n-2, -1, -1):
65         x[i] = (y[i] - np.dot(U[i, i:], x[i:])) / U[i, i] # Passage 4
66     return x
67
68 # Load the data
69 data = np.genfromtxt("./Vandermonde.txt", comments='#', dtype=np.float64)
70 x = data[:, 0]
71 y = data[:, 1]
72
73 # Create the Vandermonde matrix
74 def create_vandermonde(x, N=None):
75     if N is None: # If N is not provided, use the length of x
76         N = len(x)
77     V = np.zeros((N, N))
78     for i in range(N):
79         for j in range(N):
80             V[i, j] = x[i] ** j
81     return V
82
83 V = create_vandermonde(x)
84
85 # LU decomposition using Crout's algorithm
86 L, U = lu_decomposition(V)
87
88 # Solve the system
89 y_forward = forward_substitution(L, y)
90 c = backward_substitution(U, y_forward)
91
92 # Save the coefficients to a text file
93 np.savetxt('coefficients.txt', c)
94

```

```

95 # Plot the 19-th degree polynomial evaluated at 1000 equally-spaced points
96 # Along with the original data points
97 # This plot is just to have another look at the region of interest (it is the same as
   the next one)
98 x_values = np.linspace(0, 100, 1000)
99 y_values = np.zeros_like(x_values)
100 for i in range(len(c)):
101     y_values += c[i] * x_values ** i
102 plt.plot(x_values, y_values, label="19-th degree polynomial")
103 plt.ylim(-1000, 6000)
104 plt.scatter(x, y, color='red', label='Data points')
105 plt.legend()
106 plt.savefig('./plots/polynomial_plot.png')
107
108
109 xx=np.linspace(x[0],x[-1],1001) #x values to interpolate at
110 # Calculate interpolated y values
111 yya = np.array([np.sum(c * xi ** np.arange(len(c))) for xi in xx])
112 ya = np.array([np.sum(c * xi ** np.arange(len(c))) for xi in x])
113
114 # Plot 2a
115 fig=plt.figure()
116 gs=fig.add_gridspec(2,hspace=0,height_ratios=[2.0,1.0])
117 axs=gs.subplots(sharex=True,sharey=False)
118 axs[0].plot(x,y,marker='o',linewidth=0)
119 plt.xlim(-1,101)
120 axs[0].set_ylim(-400,400)
121 axs[0].set_ylabel('$y$')
122 #axs[1].set_ylim(1e-16,1e1)
123 axs[1].set_ylim(1e-20,1e1)
124 axs[1].set_ylabel('$|y-y_i|$')
125 axs[1].set_xlabel('$x$')
126 axs[1].set_yscale('log')
127 line,=axs[0].plot(xx,yya,color='orange')
128 line.set_label('Via LU decomposition')
129 axs[0].legend(frameon=False,loc="lower left")
130 axs[1].plot(x,abs(y-ya),color='orange')
131 plt.savefig('./plots/my_vandermonde_sol.2a.png',dpi=600)
132
133
134 '''
135 Point b):
136 We implement Neville's algorithm to interpolate the data points.
137 1) We start by identifying the M tabulated points closest to the point x0
138 To do so, we sort the points by their distance to x0 (using the function argsort) and
   select the M closest points.
139 2) We set the initial p_i = y_i for i in 0 to M.
140 3) We loop over each order of the interpolation from 1 to M-1
141 For each order, we loop over each interval of the current order from 0 to M-k-1
142 and update the p value for the interval, overwriting the previous orders' values.
143 4) We return the first element of p, which is the interpolated value.
144
145
146 '''
147
148 def argsort(seq):
149     # Pair each element in the sequence with its index
150     indexed_seq = [(val, i) for i, val in enumerate(seq)]
151     # Sort the indexed sequence
152     sorted_indexed_seq = sorted(indexed_seq)
153     # Extract the indices from the sorted indexed sequence
154     sorted_indices = [i for val, i in sorted_indexed_seq]
155     return sorted_indices
156
157
158 def neville(x, y, x0, M):
159     '''
160     Parameters:
161     x: x-coordinates of the data points
162     y: y-coordinates of the data points

```

```

163     x0: x-coordinate to interpolate at
164     M: number of points to use for interpolation
165     Returns:
166     p[0]: interpolated value at x0
167     '''
168     # Sort the points by their distance to x0 and select the M closest points
169     sorted_indices = argsort(np.abs(x - x0))
170     x = x[sorted_indices[:M]]
171     y = y[sorted_indices[:M]]
172
173     # Set p = y
174     p = np.copy(y)
175
176     # Loop over each order of the interpolation
177     for k in range(1, M):
178         # Loop over each interval of the current order
179         for i in range(M-k):
180             # Update the p value for the interval
181             p[i] = ((x0 - x[i+k])*p[i] + (x[i] - x0)*p[i+1]) / (x[i] - x[i+k])
182     return p[0] # Return the first element which is the interpolated value
183
184
185 # Values for interpolation
186 xx = np.linspace(0, 100, 1000)
187
188 # Interpolation using Neville's algorithm
189 yyb = np.array([neville(x, y, xi, 20) for xi in xx])
190 yb = np.array([neville(x, y, xi, 20) for xi in x])
191
192 # Plotting
193 fig = plt.figure()
194 gs = fig.add_gridspec(2, hspace=0, height_ratios=[2.0, 1.0])
195 axs = gs.subplots(sharex=True, sharey=False)
196 axs[0].plot(x, y, marker='o', linewidth=0)
197 plt.xlim(-1, 101)
198 axs[0].set_ylim(-400, 400)
199 axs[0].set_ylabel('$y$')
200 axs[1].set_ylim(1e-30, 1e1)
201 axs[1].set_ylabel('$|y-y_i|$')
202 axs[1].set_xlabel('$x$')
203 axs[1].set_yscale('log')
204 line, = axs[0].plot(xx, yyb, linestyle='dashed', color='green')
205 line.set_label("Via Neville's algorithm")
206 axs[0].legend(frameon=False, loc="lower left")
207 axs[1].plot(x, abs(y - yb), linestyle='dashed', color='green')
208 plt.savefig('./plots/my-vandermonde-sol-2b.png', dpi=600)
209
210 '''
211 Point c):
212 Iterative version of LU decomposition.
213 To do so, we start by saving the value of the initial guess x0
214 (this passage is necessary to avoid overwriting the initial guess during iterations).
215
216 We then loop over the number of iterations and perform the following steps:
217 1) Calculate delta_b from Ax'=b+delta_b where x'=x0+delta_x (the imperfect solution)
218 2) Perform LU decomposition
219 3) Solve Ly = \del b for y
220 4) Solve U \del x = y for \del x. This \del x is used to improve the solution x0: x'=x
221    '-\del x (where x' is the current imperfect solution)
222
223 '''
224 # Function for iterative solution using LU decomposition
225 def iterative_improvement(A, b, x0, iterations):
226     '''
227     Inputs:
228     A: matrix
229     b: vector
230     x0: initial guess for the solution
231     iterations: number of iterations to perform
232     Returns:

```

```

232     x: improved solution
233     '''
234
235     x = np.copy(x0)
236     for _ in range(iterations):
237         # Difference between actual b and b calculated using current solution x
238         delta_b = np.dot(A, x) - b
239         # Perform LU decomposition
240         L, U = lu.decomposition(A)
241         # Solve Ly = \del b for y
242         y = forward_substitution(L, delta_b)
243         # Solve U \del x = y for \del x
244         delta_x = backward_substitution(U, y)
245         # Improve the solution
246         x -= delta_x
247     return x
248
249
250 # Perform 1 LU iteration
251 c1 = iterative_improvement(V, y, c, 1)
252 yya1 = np.array([np.sum(c1 * xi ** np.arange(len(c1))) for xi in xx])
253 ya1 = np.array([np.sum(c1 * xi ** np.arange(len(c1))) for xi in x])
254
255 # Perform 10 LU iterations
256 c10 = iterative_improvement(V, y, c, 10)
257 yya10 = np.array([np.sum(c10 * xi ** np.arange(len(c10))) for xi in xx])
258 ya10 = np.array([np.sum(c10 * xi ** np.arange(len(c10))) for xi in x])
259
260 # Plot 2c
261 fig=plt.figure()
262 gs=fig.add_gridspec(2,hspace=0,height_ratios=[2.0,1.0])
263 axs=gs.subplots(sharex=True,sharey=False)
264 axs[0].plot(x,y,marker='o',linewidth=0)
265 plt.xlim(-1,101)
266 axs[0].set_ylim(-400,400)
267 axs[0].set_ylabel('$y$')
268 axs[1].set_ylim(1e-16,1e3)
269 axs[1].set_ylabel('$|y-y_i|$')
270 axs[1].set_xlabel('$x$')
271 axs[1].set_yscale('log')
272 line,=axs[0].plot(xx,yya1,linestyle='dotted',color='red')
273 line.set_label('LU with 1 iteration')
274 axs[1].plot(x,abs(y-ya1),linestyle='dotted',color='red')
275
276 line,=axs[0].plot(xx,yya10,linestyle='dashdot',color='purple')
277 line.set_label('LU with 10 iterations')
278 axs[1].plot(x,abs(y-ya10),linestyle='dashdot',color='purple')
279
280 axs[0].legend(frameon=False,loc="lower left")
281 plt.savefig('./plots/my_vandermonde_sol.2c.png',dpi=600)
282
283 '''
284 Point d): Exection time comparison
285
286 To measure the execution time of the three methods, we use the timeit module.
287 We run each method 100 times (specified by 'numnber') and save the results to a text
288     file.
289
290 - globals() is used to pass the variables defined in the code to the timeit function
291 - the first instance of timeit is used to measure the execution time
292 - the second instance of timeit is used to display the results obtained in previous
293     step
294
295 '''
296
297 # (a) LU decomposition
298 lu_time = timeit.timeit("lu_decomposition(V)", globals=globals(), number=100)
299
300 # (b) Neville's algorithm
301 neville_time = timeit.timeit("np.array([neville(x, y, xi, 20) for xi in xx])", globals=

```

```

300     globals(), number=100)
301 # (c) Iterative improvement with LU decomposition (10 iterations)
302 iterative_time = timeit.timeit("iterative_improvement(V, y, c, 10)", globals=globals(),
303                                number=100)
304 # Write results to a text file
305 with open('timing_results.txt', 'w') as file:
306     file.write(f"LU Decomposition Time: {lu_time:.6f} seconds\n")
307     file.write(f"Neville's Algorithm Time: {neville_time:.6f} seconds\n")
308     file.write(f"Iterative Improvement Time: {iterative_time:.6f} seconds\n")

```

vandermonde.py

For question (a), we are performing a polynomial interpolation on a set of data points using a 19th-degree polynomial, evaluated at  $\approx 1000$  equally-spaced points. The coefficients of the polynomial are determined by solving a system of linear equations using LU decomposition: they are computed in

```

1 1.745449629462546071e+01
2 -1.912658070469094298e+02
3 2.400858901688944798e+02
4 -1.102904824261681966e+02
5 2.688278188786916090e+01
6 -4.073856572355006200e+00
7 4.175711801925817701e-01
8 -3.045818340884471134e-02
9 1.634034746488180218e-03
10 -6.591724714078017072e-05
11 2.028537534785111589e-06
12 -4.801909239978146986e-08
13 8.767406678983276242e-10
14 -1.230929039236915523e-11
15 1.315942367469549149e-13
16 -1.051190828510772090e-15
17 6.072075154696671321e-18
18 -2.394606872271598716e-20
19 5.766526114820814136e-23
20 -6.394781464844003863e-26

```

coefficients.txt

. We also plot the absolute difference between the given points  $y_i$  and our result  $y(x)$ , i.e.  $|y(x) - y_i|$ .

From Fig.1, it seems that the polynomial is fitting the data points quite well for most of the range. This is expected as a polynomial of degree  $(n)$  can always fit  $(n + 1)$  data points exactly. However, towards the end (around  $x = 100$ ), the polynomial shoots up drastically. This is a common issue with high-degree polynomial interpolation known as Runge's phenomenon. It is a form of overfitting where the polynomial oscillates significantly at the boundaries of the data set.

The choice of a 19th-degree polynomial for this data might not be the best. While it fits the given data points well, the extreme behavior at the boundaries suggests that it might not generalize well to other data. A lower-degree polynomial or a different type of function might provide a better fit without the extreme oscillations.

The bottom part of the plot shows the absolute difference between the given  $y_i$  values and the calculated  $y(x)$  values. This represents the error in the polynomial fit at the data points. Since the polynomial fits the data points exactly, these errors are close to zero, going from order  $10^{-14}$  to  $10^{-2}$ . We can note how the error increases at the boundaries, consistently with the issue presented before.

For question (b), we implement Neville's algorithm (see Fig. 2). It works by recursively evaluating a set of polynomials and combining them to form the final interpolating polynomial. Regarding the interpolation, we are getting the same results as LU decomposition, so they have same efficiency in fitting the given data points. The errors are drastically small for Neville's (reaching a maximum of  $10^{-14}$ ) while LU decomposition have errors slightly higher. We could explain this addressing to the nature of the algorithm, which tends to be stable and accurate, especially when the data points are equally

spaced. The accuracy of LU decomposition for interpolation can be influenced by the condition of the matrix involved. Ill-conditioned matrices may lead to numerical instability and higher errors. In our case, the condition number of the Vandermonde matrix depends on the arrangement and spacing of the data points. If the data points are well-spaced and not too close to each other, the condition number may be reasonable. However, as the data points become closely spaced or nearly collinear, the problem of ill-conditioning can arise.

For question (c), we improve iteratively the solution found by LU decomposition. We do such procedure for 1 and 10 iterations (Fig. 3). The interpolation is the same for both cases, since we have overlapping fits. Regarding the trend for the error, the two procedures get similar results, with the LU with 10 iterations having slightly lower errors in some regions of the data.

For question (d), we obtain the execution times for LU decomposition, Neville's algorithm, and LU with iterative improvement: the results are given by

```
1 LU Decomposition Time: 0.006999 seconds
2 Neville's Algorithm Time: 9.412113 seconds
3 Iterative Improvement Time: 0.104147 seconds
```

timing\_results.txt

- **LU Decomposition Time:** The algorithm is known for its efficiency in solving linear systems of equations, and it typically has a low time complexity. This outcome indicates that LU decomposition is well-suited for our data set.
- **Neville's Algorithm Time:** The algorithm involves several polynomial evaluations and recursive calculations: the complexity of these calculations can lead to longer execution times, especially as the number of data points increases.
- **Iterative Improvement Time:** It is refining a solution through iterative steps, in a relatively fast process. The efficiency of this process can depend on the convergence behavior of the iterative method and the initial guess.

From these results, the LU decomposition method appears to be the most efficient in terms of computational time. If we take the speed as a critical factor and the accuracy achieved by LU decomposition is sufficient for your application, then LU decomposition may be the preferred choice. In our case, the accuracy achieved by LU decomposition is sufficient for our application. However, we have to keep in mind that LU decomposition, while efficient, might be sensitive to ill-conditioned matrices, potentially affecting accuracy in certain cases. In general, accuracy depends on the interpolation requirements and the nature of your data. However, Neville's algorithm is generally considered accurate, especially for well-behaved, evenly spaced data. In our implementation, it is behaving considerably slower than the other, so it might not be the suitable option for this particular setting, even though it is reaching the best performance in terms of lowest errors in the interpolation.



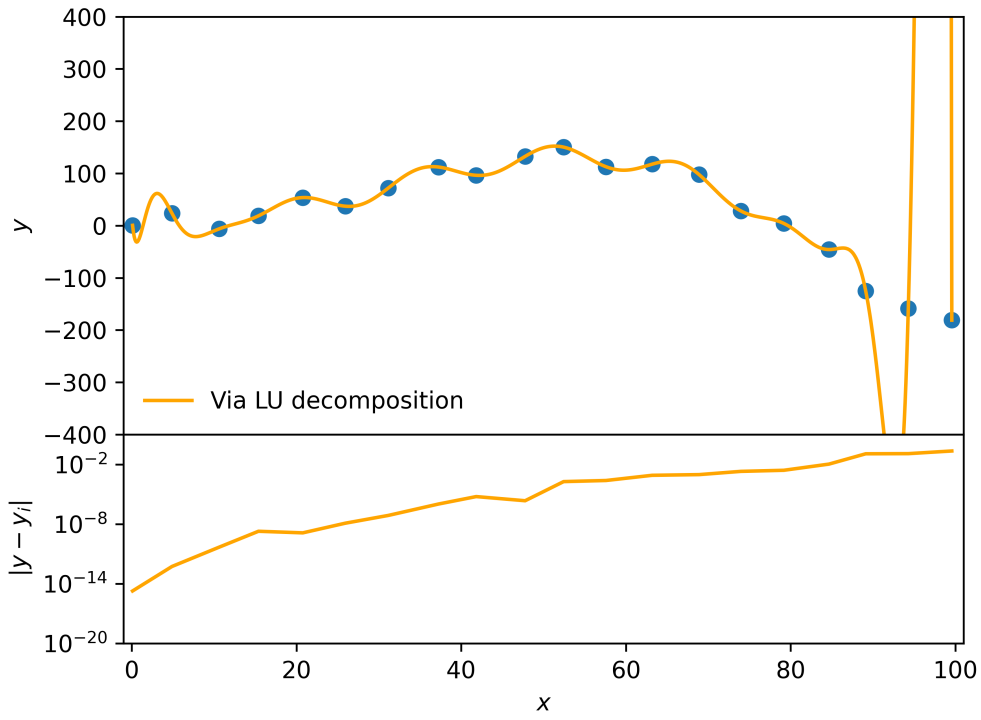


Figure 1: Upper panel: Interpolation on a set of given data points via LU decomposition. The fit is going through all the data points exactly. Nevertheless, it has an oscillating behaviour in correspondence of the last two points. Bottom panel: Absolute difference between the given points  $y_i$  and our result  $y(x)$ , i.e.  $|y(x) - y_i|$ . The error holds at values close to zero, with a small increase towards the boundaries.

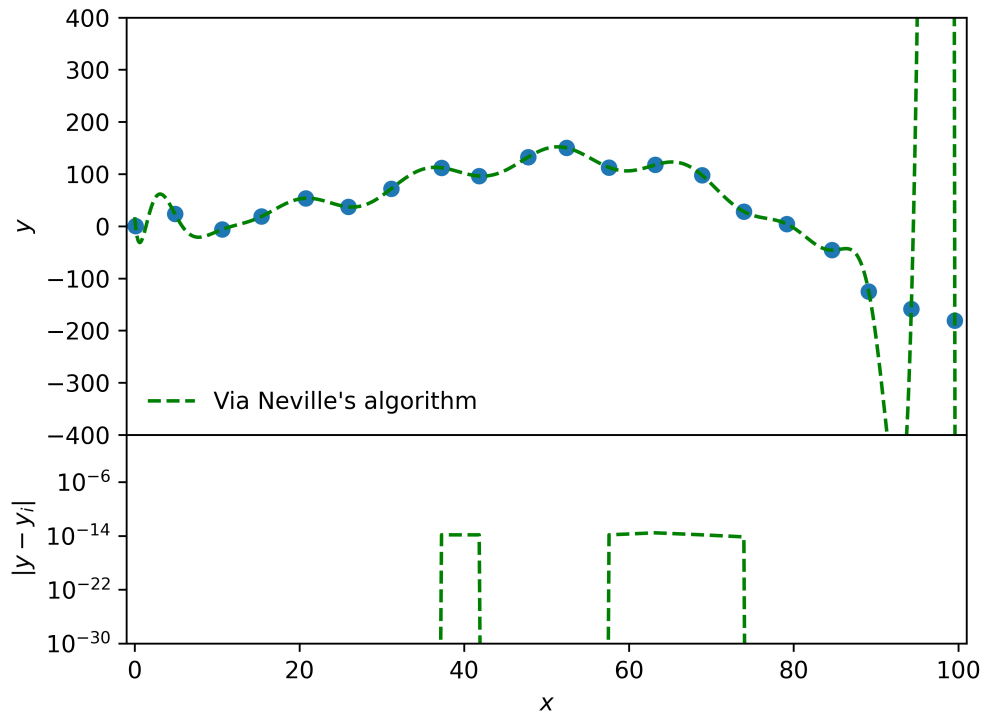


Figure 2: Upper panel: Interpolation on a set of given data points via Neville's algorithm. The polynomial obtained fits the data points well in the range considered, in the same way the interpolation of LU decomposition does. Bottom panel: Absolute difference between the given points  $y_i$  and our result  $y(x)$ , i.e.  $|y(x) - y_i|$ . The errors of Neville's algorithm reach very small values with respect to LU decomposition's ones.

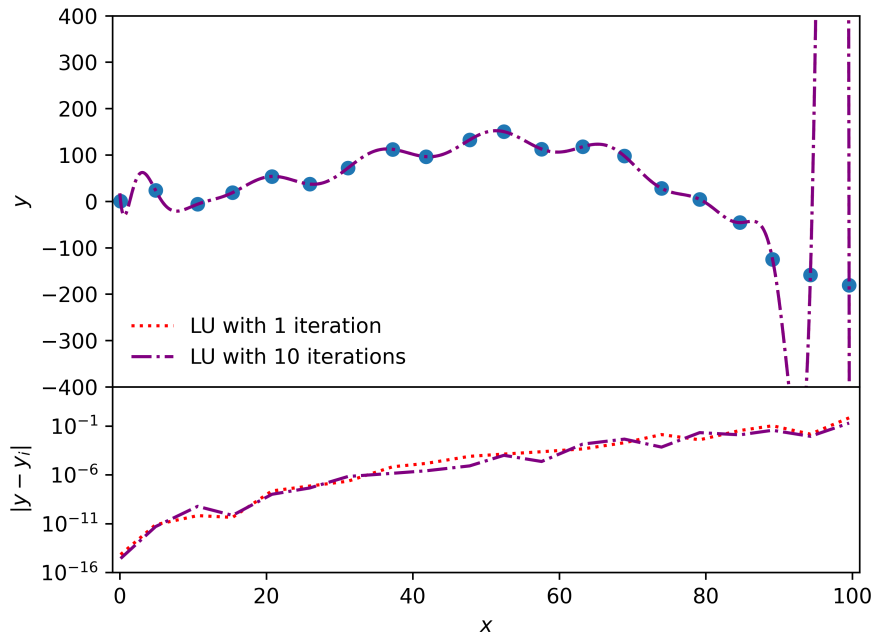


Figure 3: Upper panel: Interpolation on a set of given data points via LU decomposition, with iterative improvement of the found solution (done with 1 and 10 iterations). Both the fits are going through all the data points exactly, hence the overlap. Bottom panel: Absolute difference between the given points  $y_i$  and our result  $y(x)$ , i.e.  $|y(x) - y_i|$ . The trend is similar for the two implementations, with a small decrease in values reached by LU with 10 iterations.