

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
23     Output:
24     Poisson probability for a given lambda and k
25
26     We divide the calculation into:
27     1. Calculate  $\lambda^k$  in log space
28     2. Calculate  $e^{-\lambda}$ 
29     3. Calculate  $k!$  and the cumulative sum of  $\log(k!)$  for all k from 2.
30     The reason for starting from 2 is that the first two values of  $k!$  are 0 and 1, not
31     useful for the calculation.
32     The cumulative sum is useful for next calculation
33     4. Calculate the Poisson probability in log space and convert it back to normal
34     space.
35
36     '''
37
38     k_values = np.float32(np.arange(k + 1))
39
40     # 1.  $\log(\lambda^k)$ 
41     lam_k_log = np.float32(k_values * log10(l))
42     # 2.  $e^{-\lambda}$ 
43     e_neg_lambda = np.float32(log10(exp(-l)))
44     # 3. Factorial calculation:  $\log(k!)$  and cumulative sum of  $\log(k!)$  for all k from 2
45     log_fact_k = np.float32(np.log10(np.arange(2, k + 1)))
46     tot_log_fact = np.float32(np.cumsum(log_fact_k))
47     # 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
    probabilities
46     poisson_distribution = np.float32(np.concatenate([[(1**k) * exp(-1)] for k in
    k_values[:2]] + [np.power(10, log_p_k)]))
47
48     return poisson_distribution
49
50 # Test the function for the given values
51 parameter_values = [(1, 0), (5, 10), (3, 21), (2.6, 40), (101, 200)]
52
53 # Save Poisson probability results to a text file
54 output_data = []
55 for lambda_val, k_val in parameter_values:
56     poisson_distribution = poisson_prob(lambda_val, k_val)
57     output_data.append([lambda_val, k_val, poisson_distribution[int(k_val)]])
58
59 np.savetxt('poisson_output.txt', np.array(output_data), header='Lambda\tK\tPoisson
    Probability', delimiter='\t')

```

poisson.py

. The necessary explanations are in the comments of the code. The output produced for the values required is in `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
    system.
14 The problem is divided into:
15
16 1. LU decomposition: the matrix A is decomposed into a lower triangular matrix L and an
    upper triangular matrix U.
17 Firstly we initialize L and U as zero matrices of the same size as A.
18 Setting alpha_ij is done implicitly in passage 1:
19 since L[i, :i] and U[:i, i] are both zero vectors at this point this dot product is zero
    , and so U[i, i] is just A[i, i].
20 Then we update the elements of L and U loop over the columns.
21
22
23 2. Forward substitution: we solve Ly = b for y.
24 We initialize y as a zero vector of the same size as b.
25 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): #Ã Update the upper and lower triangular matrices
42         U[i, i:] = A[i, i:] - np.dot(L[i, :i], U[:i, i:]) # Passage 1
43         L[i:, i] = (A[i:, i] - np.dot(L[i:, :i], U[:i, i])) / U[i, i] # Passage 2
44
45     return L, U
46
47 # Forward substitution
48 def forward_substitution(L, b):
49     n = L.shape[0]
50     y = np.zeros_like(b, dtype=np.double)
51     # Solve Ly = b for y
52     y[0] = b[0] / L[0, 0]
53     for i in range(1, n):
54         y[i] = (b[i] - np.dot(L[i, :i], y[:i])) / L[i, i] # Passage 3
55     return y
56
57 # Backward substitution
58 def backward_substitution(U, y):
59     n = U.shape[0]
60     x = np.zeros_like(y, dtype=np.double)
61     # Solve Ux = y for x
62     x[-1] = y[-1] / U[-1, -1]
63     for i in range(n-2, -1, -1):
64         x[i] = (y[i] - np.dot(U[i, i:], x[i:])) / U[i, i] # Passage 4
65     return x
66
67 # Load the data
68 data = np.genfromtxt("./Vandermonde.txt", comments='#', dtype=np.float64)
69 x = data[:, 0]
70 y = data[:, 1]
71
72 # Create the Vandermonde matrix
73 def create_vandermonde(x, N=None):
74     if N is None: # If N is not provided, use the length of x
75         N = len(x)
76     V = np.zeros((N, N))
77     for i in range(N):
78         for j in range(N):
79             V[i, j] = x[i] ** j
80     return V
81
82 V = create_vandermonde(x)
83
84 # LU decomposition using Crout's algorithm
85 L, U = lu_decomposition(V)
86
87 # Solve the system
88 y_forward = forward_substitution(L, y)
89 c = backward_substitution(U, y_forward)
90
91 # Save the coefficients to a text file
92 np.savetxt('coefficients.txt', c)
93
94 # Plot the 19-th degree polynomial evaluated at 1000 equally-spaced points
95 # Along with the original data points
96 # This plot is just to have another look at the region of interest (it is the same as
97 # 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')

```

```

104 plt.legend()
105 plt.savefig('./plots/polynomial_plot.png')
106
107
108 xx=np.linspace(x[0],x[-1],1001) #x values to interpolate at
109 # Calculate interpolated y values
110 yya = np.array([np.sum(c * xi ** np.arange(len(c))) for xi in xx])
111 ya = np.array([np.sum(c * xi ** np.arange(len(c))) for xi in x])
112
113 # Plot 2a
114 fig=plt.figure()
115 gs=fig.add_gridspec(2,hspace=0,height_ratios=[2.0,1.0])
116 axs=gs.subplots(sharex=True,sharey=False)
117 axs[0].plot(x,y,marker='o',linewidth=0)
118 plt.xlim(-1,101)
119 axs[0].set_ylim(-400,400)
120 axs[0].set_ylabel('$y$')
121 #axs[1].set_ylim(1e-16,1e1)
122 axs[1].set_ylim(1e-20,1e1)
123 axs[1].set_ylabel('$|y-y_i|$')
124 axs[1].set_xlabel('$x$')
125 axs[1].set_yscale('log')
126 line,=axs[0].plot(xx,yya,color='orange')
127 line.set_label('Via LU decomposition')
128 axs[0].legend(frameon=False,loc="lower left")
129 axs[1].plot(x,abs(y-ya),color='orange')
130 plt.savefig('./plots/my-vandermonde-sol-2a.png',dpi=600)
131
132
133 '''
134 Point b):
135 We implement Neville's algorithm to interpolate the data points.
136 1) We start by identifying the M tabulated points closest to the point x0
137 To do so, we sort the points by their distance to x0 (using the function argsort) and
    select the M closest points.
138 2) We set the initial p_i = y_i for i in 0 to M.
139 3) We loop over each order of the interpolation from 1 to M-1
140 For each order, we loop over each interval of the current order from 0 to M-k-1
141 and update the p value for the interval, overwriting the previous orders' values.
142 4) We return the first element of p, which is the interpolated value.
143
144
145 '''
146
147 def argsort(seq):
148     # Pair each element in the sequence with its index
149     indexed_seq = [(val, i) for i, val in enumerate(seq)]
150     # Sort the indexed sequence
151     sorted_indexed_seq = sorted(indexed_seq)
152     # Extract the indices from the sorted indexed sequence
153     sorted_indices = [i for val, i in sorted_indexed_seq]
154     return sorted_indices
155
156
157 def neville(x, y, x0, M):
158     '''
159     Parameters:
160     x: x-coordinates of the data points
161     y: y-coordinates of the data points
162     x0: x-coordinate to interpolate at
163     M: number of points to use for interpolation
164     Returns:
165     p[0]: interpolated value at x0
166     '''
167     # Sort the points by their distance to x0 and select the M closest points
168     sorted_indices = argsort(np.abs(x - x0))
169     x = x[sorted_indices[:M]]
170     y = y[sorted_indices[:M]]
171
172     # Set p = y

```

```

173     p = np.copy(y)
174
175     # Loop over each order of the interpolation
176     for k in range(1, M):
177         # Loop over each interval of the current order
178         for i in range(M-k):
179             # Update the p value for the interval
180             p[i] = ((x0 - x[i+k])*p[i] + (x[i] - x0)*p[i+1]) / (x[i] - x[i+k])
181     return p[0] # Return the first element which is the interpolated value
182
183
184 # Values for interpolation
185 xx = np.linspace(0, 100, 1000)
186
187 # Interpolation using Neville's algorithm
188 yyb = np.array([neville(x, y, xi, 20) for xi in xx])
189 yb = np.array([neville(x, y, xi, 20) for xi in x])
190
191 # Plotting
192 fig = plt.figure()
193 gs = fig.add_gridspec(2, hspace=0, height_ratios=[2.0, 1.0])
194 axs = gs.subplots(sharex=True, sharey=False)
195 axs[0].plot(x, y, marker='o', linewidth=0)
196 plt.xlim(-1, 101)
197 axs[0].set_ylim(-400, 400)
198 axs[0].set_ylabel('$y$')
199 axs[1].set_ylim(1e-30, 1e1)
200 axs[1].set_ylabel('$|y-y_i|$')
201 axs[1].set_xlabel('$x$')
202 axs[1].set_yscale('log')
203 line, = axs[0].plot(xx, yyb, linestyle='dashed', color='green')
204 line.set_label("Via Neville's algorithm")
205 axs[0].legend(frameon=False, loc="lower left")
206 axs[1].plot(x, abs(y - yb), linestyle='dashed', color='green')
207 plt.savefig('./plots/my_vandermonde.sol_2b.png', dpi=600)
208
209 '''
210 Point c):
211 Iterative version of LU decomposition.
212 To do so, we start by saving the value of the initial guess x0
213 (this passage is necessary to avoid overwriting the initial guess during iterations).
214
215 We then loop over the number of iterations and perform the following steps:
216 1) Calculate delta_b from Ax'=b+delta_b where x'=x0+delta_x (the imperfect solution)
217 2) Perform LU decomposition
218 3) Solve Ly = Î'b for y
219 4) Solve U Î'x = y for Î'x. This Î'x is used to improve the solution x0: x''=x'-Î'x (
    where x' is the current imperfect solution)
220
221 '''
222 # Function for iterative solution using LU decomposition
223 def iterative_improvement(A, b, x0, iterations):
224     '''
225     Inputs:
226     A: matrix
227     b: vector
228     x0: initial guess for the solution
229     iterations: number of iterations to perform
230     Returns:
231     x: improved solution
232     '''
233
234     x = np.copy(x0)
235     for _ in range(iterations):
236         # Difference between actual b and b calculated using current solution x
237         delta_b = np.dot(A, x) - b
238         # Perform LU decomposition
239         L, U = lu_decomposition(A)
240         # Solve Ly = Î'b for y
241         y = forward_substitution(L, delta_b)

```

```

242     # Solve  $U \hat{I}'x = y$  for  $\hat{I}'x$ 
243     delta_x = backward_substitution(U, y)
244     # Improve the solution
245     x -= delta_x
246     return x
247
248
249 # Perform 1 LU iteration
250 c1 = iterative_improvement(V, y, c, 1)
251 yya1 = np.array([np.sum(c1 * xi ** np.arange(len(c1))) for xi in xx])
252 ya1 = np.array([np.sum(c1 * xi ** np.arange(len(c1))) for xi in x])
253
254 # Perform 10 LU iterations
255 c10 = iterative_improvement(V, y, c, 10)
256 yya10 = np.array([np.sum(c10 * xi ** np.arange(len(c10))) for xi in xx])
257 ya10 = np.array([np.sum(c10 * xi ** np.arange(len(c10))) for xi in x])
258
259 # Plot 2c
260 fig=plt.figure()
261 gs=fig.add_gridspec(2,hspace=0,height_ratios=[2.0,1.0])
262 axs=gs.subplots(sharex=True,sharey=False)
263 axs[0].plot(x,y,marker='o',linewidth=0)
264 plt.xlim(-1,101)
265 axs[0].set_ylim(-400,400)
266 axs[0].set_ylabel('$y$')
267 axs[1].set_ylim(1e-16,1e3)
268 axs[1].set_ylabel('$|y-y_i|$')
269 axs[1].set_xlabel('$x$')
270 axs[1].set_yscale('log')
271 line,=axs[0].plot(xx,ya1,linestyle='dotted',color='red')
272 line.set_label('LU with 1 iteration')
273 axs[1].plot(x,abs(y-ya1),linestyle='dotted',color='red')
274
275 line,=axs[0].plot(xx,ya10,linestyle='dashdot',color='purple')
276 line.set_label('LU with 10 iterations')
277 axs[1].plot(x,abs(y-ya10),linestyle='dashdot',color='purple')
278
279 axs[0].legend(frameon=False,loc="lower left")
280 plt.savefig('./plots/my_vandermonde_sol_2c.png',dpi=600)
281
282 '''
283 Point d): Exection time comparison
284
285 To measure the execution time of the three methods, we use the timeit module.
286 We run each method 100 times (specified by 'numnber') and save the results to a text
287     file.
288
289     - globals() is used to pass the variables defined in the code to the timeit function
290     - the first instance of timeit is used to measure the execution time
291     - the second instance of timeit is used to display the results obtained in previous
292       step
293
294     '''
295
296 # (a) LU decomposition
297 lu_time = timeit.timeit("lu_decomposition(V)", globals=globals(), number=100)
298
299 # (b) Neville's algorithm
300 neville_time = timeit.timeit("np.array([neville(x, y, xi, 20) for xi in xx])", globals=
301     globals(), number=100)
302
303 # (c) Iterative improvement with LU decomposition (10 iterations)
304 iterative_time = timeit.timeit("iterative_improvement(V, y, c, 10)", globals=globals(),
305     number=100)
306
307 # Write results to a text file
308 with open('timing_results.txt', 'w') as file:
309     file.write(f"LU Decomposition Time: {lu_time:.6f} seconds\n")
310     file.write(f"Neville's Algorithm Time: {neville_time:.6f} seconds\n")
311     file.write(f"Iterative Improvement Time: {iterative_time:.6f} seconds\n")

```

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

- **LU Decomposition Time:** 0.012404 seconds. 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:** 9.874374 seconds. 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:** 0.117441 seconds. 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.

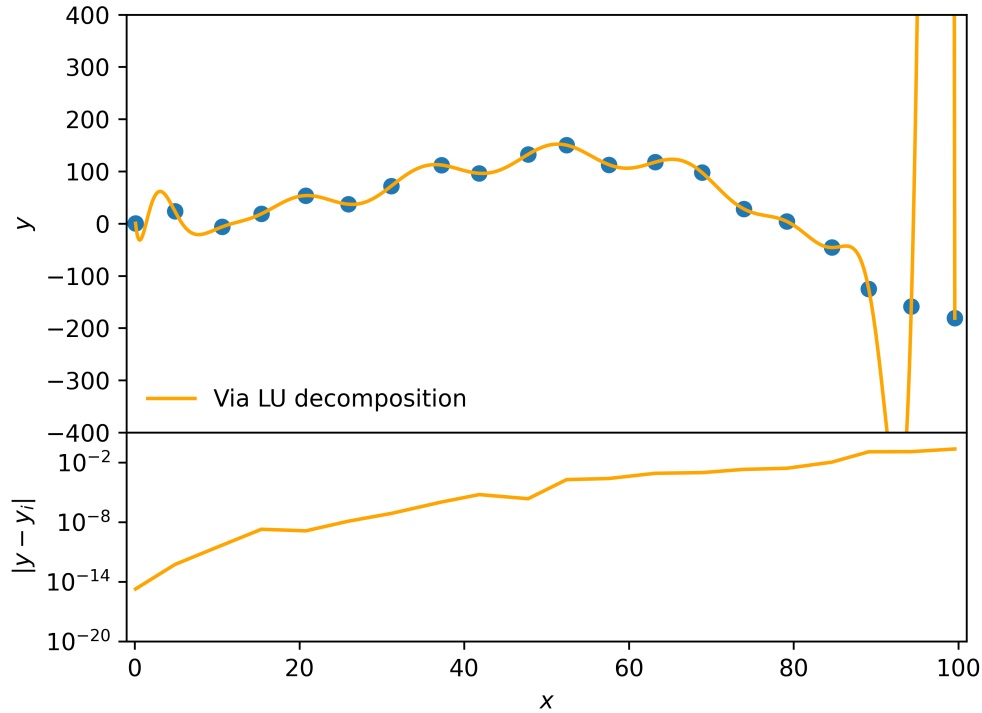


Figure 1: Upper panel: Interpolation on a set of given data points via LU decomposition. The fit is going through all the data points exactly. Nevertheless, it has an oscillating behaviour in correspondence of the last two points. Bottom panel: Absolute difference between the given points y_i and our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

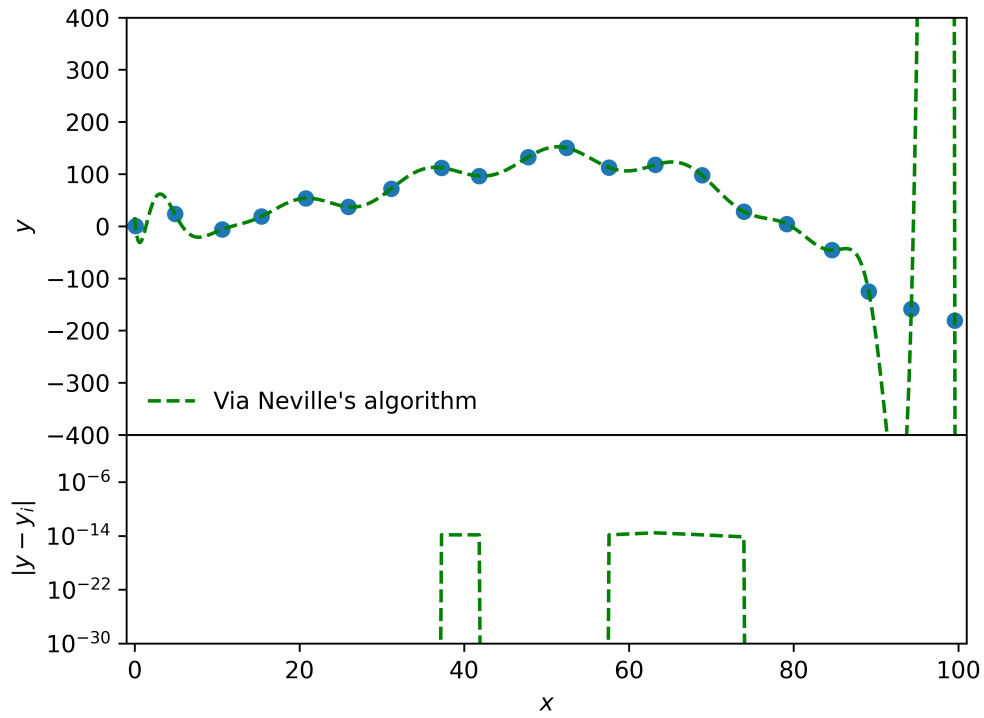


Figure 2: Upper panel: Interpolation on a set of given data points via Neville's algorithm. The polynomial obtained fits the data points well in the range considered, in the same way the interpolation of LU decomposition does. Bottom panel: Absolute difference between the given points y_i and our result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

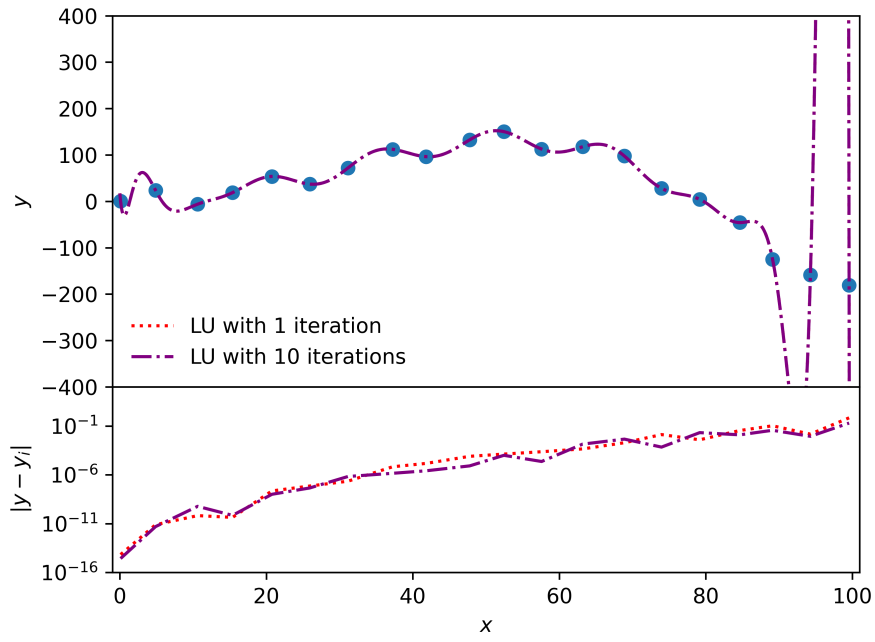


Figure 3: Upper panel: Interpolation on a set of given data points via LU decomposition, with iterative improvement of the found solution (done with 1 and 10 iterations). Both the fits are going through all the data points exactly, hence the overlap. Bottom panel: Absolute difference between the given 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.