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

```python
#!/usr/bin/env python
import numpy as np
from math import exp, log10

'''
This script containes the code for Exercise 1.
'''

def poisson_prob(l, k):
    '''
    Calculates the Poisson probability for a given lambda and k
    At each passage, we work with 32 bit floats and in log space to avoid overflow.
    By taking the logarithm, we transform multiplications into additions and divisions into subtractions.
    This helps keep the intermediate results within a manageable range.
    We then take the exponent of the final result, ensuring both numerical stability and correct results.

    Inputs:
    l: lambda value
    k: k value
    Output:
    Poisson probability for a given lambda and k

    We divide the calculation into:
    1. Calculate lambaˆk in log space
    2. Calculate eˆ−lambda
    3. Calculate k! and the cumulative sum of log(k!) for all k from 2.
    The reason for starting from 2 is that the first two values of k! are 0 and 1, not useful for the calculation.
    The cumulative sum is useful for next calculation
    4. Calculate the Poisson probability in log space and convert it back to normal space.

    '''

    k_values = np.float32(np.arange(k + 1))

    # 1. log(lambaˆk)
    lam_k_log = np.float32(k_values * log10(l))
    # 2. eˆ−lambda
    e_neg_lambda = np.float32(log10(exp(-l)))
    # 3. Factorial calculation: log(k!) and cumulative sum of log(k!) for all k from 2
    log_fact_k = np.float32(np.log10(np.arange(2, k + 1)))
    tot_log_fact = np.float32(np.cumsum(log_fact_k))
    # 4. Poisson probability in log space and conversion
```

```
    log_p_k = np.float32(lam_k_log[2:] + e_neg_lambda - tot_log_fact)
    # Terms k=0,1 are calculated separately to maintain accuracy
    # The remaining k are obtained by exponentiating the logarithmic Poisson
    probabilities
    poisson_distribution = np.float32(np.concatenate([[(l**k) * exp(-l)] for k in
    k_values[:2]] + [np.power(10, log_p_k)]))

    return poisson_distribution

# Test the function for the given values
parameter_values = [(1, 0), (5, 10), (3, 21), (2.6, 40), (101, 200)]

# Save Poisson probability results to a text file
output_data = []
for lambda_val, k_val in parameter_values:
    poisson_distribution = poisson_prob(lambda_val, k_val)
    output_data.append([lambda_val, k_val, poisson_distribution[int(k_val)]])

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

```
# Lambda    K    Poisson Probability
1.00000000000000000e+00    0.00000000000000000e+00    3.67879450321197509 8e-01
5.00000000000000000e+00    1.00000000000000000e+01    1.81327778846025466 9e-02
3.00000000000000000e+00    2.10000000000000000e+01    1.01934181581664340 4e-11
2.60000000000000089e+00    4.00000000000000000e+01    3.61513516298603312 8e-33
1.01000000000000000e+02    2.00000000000000000e+02    1.26942538582656751 3e-18
```

poisson_output.txt

.

2 Vandermonde matrix

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

```
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as plt
import timeit

'''
This script contains the code for Exercise 2.
'''

'''
Point a):

We implement the LU decomposition using Crout's algorithm to solve the Vandermonde
    system.
The problem is divided into:

1. LU decomposition: the matrix A is decomposed into a lower triangular matrix L and an
    upper triangular matrix U.
Firstly we initialize L and U as zero matrices of the same size as A.
Setting alpha_ii is done implicitly in passage 1:
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].
Then we update the elements of L and U loop over the columns.


2. Forward substitution: we solve Ly = b for y.
We initialize y as a zero vector of the same size as b.
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): #Â 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
```

```
# Along with the original data points
# This plot is just to have another look at the region of interest (it is the same as the next one)
x_values = np.linspace(0, 100, 1000)
y_values = np.zeros_like(x_values)
for i in range(len(c)):
    y_values += c[i] * x_values ** i
plt.plot(x_values, y_values, label="19-th degree polynomial")
plt.ylim(-1000, 6000)
plt.scatter(x, y, color='red', label='Data points')
plt.legend()
plt.savefig('./plots/polynomial_plot.png')


xx=np.linspace(x[0],x[-1],1001) #x values to interpolate at
# Calculate interpolated y values
yya = np.array([np.sum(c * xi ** np.arange(len(c))) for xi in xx])
ya = np.array([np.sum(c * xi ** np.arange(len(c))) for xi in x])

# Plot 2a
fig=plt.figure()
gs=fig.add_gridspec(2,hspace=0,height_ratios=[2.0,1.0])
axs=gs.subplots(sharex=True,sharey=False)
axs[0].plot(x,y,marker='o',linewidth=0)
plt.xlim(-1,101)
axs[0].set_ylim(-400,400)
axs[0].set_ylabel('$y$')
#axs[1].set_ylim(1e-16,1e1)
axs[1].set_ylim(1e-20,1e1)
axs[1].set_ylabel('$|y-y_i|$')
axs[1].set_xlabel('$x$')
axs[1].set_yscale('log')
line,=axs[0].plot(xx,yya,color='orange')
line.set_label('Via LU decomposition')
axs[0].legend(frameon=False,loc="lower left")
axs[1].plot(x,abs(y-ya),color='orange')
plt.savefig('./plots/my_vandermonde_sol_2a.png',dpi=600)


'''
Point b):
We implement Neville's algorithm to interpolate the data points.
1) We start by identifying the M tabulated points closest to the point x0
To do so, we sort the points by their distance to x0 (using the function argsort) and select the M closest points.
2) We set the initial p_i = y_i for i in 0 to M.
3) We loop over each order of the interpolation from 1 to M-1
For each order, we loop over each interval of the current order from 0 to M-k-1
and update the p value for the interval, overwriting the previous orders' values.
4) We return the first element of p, which is the interpolated value.


'''

def argsort(seq):
    # Pair each element in the sequence with its index
    indexed_seq = [(val, i) for i, val in enumerate(seq)]
    # Sort the indexed sequence
    sorted_indexed_seq = sorted(indexed_seq)
    # Extract the indices from the sorted indexed sequence
    sorted_indices = [i for val, i in sorted_indexed_seq]
    return sorted_indices


def neville(x, y, x0, M):
    '''
    Parameters:
    x: x-coordinates of the data points
    y: y-coordinates of the data points
    x0: x-coordinate to interpolate at
```

```
    M: number of points to use for interpolation
    Returns:
    p[0]: interpolated value at x0
    '''
    # Sort the points by their distance to x0 and select the M closest points
    sorted_indices = argsort(np.abs(x - x0))
    x = x[sorted_indices[:M]]
    y = y[sorted_indices[:M]]

    # Set p = y
    p = np.copy(y)

    # Loop over each order of the interpolation
    for k in range(1, M):
        # Loop over each interval of the current order
        for i in range(M-k):
            # Update the p value for the interval
            p[i] = ((x0 - x[i+k])*p[i] + (x[i] - x0)*p[i+1]) / (x[i] - x[i+k])
    return p[0] # Return the first element which is the interpolated value


# Values for interpolation
xx = np.linspace(0, 100, 1000)

# Interpolation using Neville's algorithm
yyb = np.array([neville(x, y, xi, 20) for xi in xx])
yb = np.array([neville(x, y, xi, 20) for xi in x])

# Plotting
fig = plt.figure()
gs = fig.add_gridspec(2, hspace=0, height_ratios=[2.0, 1.0])
axs = gs.subplots(sharex=True, sharey=False)
axs[0].plot(x, y, marker='o', linewidth=0)
plt.xlim(-1, 101)
axs[0].set_ylim(-400, 400)
axs[0].set_ylabel('$y$')
axs[1].set_ylim(1e-30,1e1)
axs[1].set_ylabel('$|y-y_i|$')
axs[1].set_xlabel('$x$')
axs[1].set_yscale('log')
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('./plots/my_vandermonde_sol_2b.png', dpi=600)

'''
Point c):
Iterative version of LU decomposition.
To do so, we start by saving the value of the initial guess x0
(this passage is necessary to avoid overwriting the initial guess during iterations).

We then loop over the number of iterations and perform the following steps:
1) Calculate delta_b from Ax'=b+delta_b where x'=x0+delta_x (the imperfect solution)
2) Perform LU decomposition
3) Solve Ly = Î´b for y
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)

'''
# Function for iterative solution using LU decomposition
def iterative_improvement(A, b, x0, iterations):
    '''
    Inputs:
    A: matrix
    b: vector
    x0: initial guess for the solution
    iterations: number of iterations to perform
    Returns:
    x: improved solution
```

```
    '''

    x = np.copy(x0)
    for _ in range(iterations):
        # Difference between actual b and b calculated using current solution x
        delta_b = np.dot(A, x) - b
        # Perform LU decomposition
        L, U = lu_decomposition(A)
        # Solve Ly = Î´b for y
        y = forward_substitution(L, delta_b)
        # Solve U Î´x = y for Î´x
        delta_x = backward_substitution(U, y)
        # Improve the solution
        x -= delta_x
    return x


# Perform 1 LU iteration
c1 = iterative_improvement(V, y, c, 1)
yya1 = np.array([np.sum(c1 * xi ** np.arange(len(c1))) for xi in xx])
ya1 = np.array([np.sum(c1 * xi ** np.arange(len(c1))) for xi in x])

# Perform 10 LU iterations
c10 = iterative_improvement(V, y, c, 10)
yya10 = np.array([np.sum(c10 * xi ** np.arange(len(c10))) for xi in xx])
ya10 = np.array([np.sum(c10 * xi ** np.arange(len(c10))) for xi in x])

# Plot 2c
fig=plt.figure()
gs=fig.add_gridspec(2,hspace=0,height_ratios=[2.0,1.0])
axs=gs.subplots(sharex=True,sharey=False)
axs[0].plot(x,y,marker='o',linewidth=0)
plt.xlim(-1,101)
axs[0].set_ylim(-400,400)
axs[0].set_ylabel('$y$')
axs[1].set_ylim(1e-16,1e3)
axs[1].set_ylabel('$|y-y_i|$')
axs[1].set_xlabel('$x$')
axs[1].set_yscale('log')
line,=axs[0].plot(xx,yya1,linestyle='dotted',color='red')
line.set_label('LU with 1 iteration')
axs[1].plot(x,abs(y-ya1),linestyle='dotted',color='red')

line,=axs[0].plot(xx,yya10,linestyle='dashdot',color='purple')
line.set_label('LU with 10 iterations')
axs[1].plot(x,abs(y-ya10),linestyle='dashdot',color='purple')

axs[0].legend(frameon=False,loc="lower left")
plt.savefig('./plots/my_vandermonde_sol_2c.png',dpi=600)

'''
Point d): Exection time comparison

To measure the execution time of the three methods, we use the timeit module.
We run each method 100 times (specified by 'numnber') and save the results to a text
    file.

 - globals() is used to pass the variables defined in the code to the timeit function
 - the first instance of timeit is used to measure the execution time
 - the second instance of timeit is used to display the results obtained in previous
    step

'''

# (a) LU decomposition
lu_time = timeit.timeit("lu_decomposition(V)", globals=globals(), number=100)

# (b) Neville's algorithm
neville_time = timeit.timeit("np.array([neville(x, y, xi, 20) for xi in xx])", globals=
    globals(), number=100)
```

```python

# (c) Iterative improvement with LU decomposition (10 iterations)
iterative_time = timeit.timeit("iterative_improvement(V, y, c, 10)", globals=globals(),
    number=100)

# Write results to a text file
with open('timing_results.txt', 'w') as file:
    file.write(f"LU Decomposition Time: {lu_time:.6f} seconds\n")
    file.write(f"Neville's Algorithm Time: {neville_time:.6f} seconds\n")
    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 ≈ 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

```
1.745449629462546071e+01
-1.912658070469094298e+02
2.400858901688944798e+02
-1.102904824261681966e+02
2.688278188786916090e+01
-4.073856572355006200e+00
4.175711801925817701e-01
-3.045818340884471134e-02
1.634034746488180218e-03
-6.591724714078017072e-05
2.028537534785111589e-06
-4.801909239978146986e-08
8.767406678983276242e-10
-1.230929039236915523e-11
1.315942367469549149e-13
-1.051190828510772090e-15
6.072075154696671321e-18
-2.394606872271598716e-20
5.766526114820814136e-23
-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

```
LU Decomposition Time: 0.007013 seconds
Neville's Algorithm Time: 9.799903 seconds
Iterative Improvement Time: 0.104062 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.

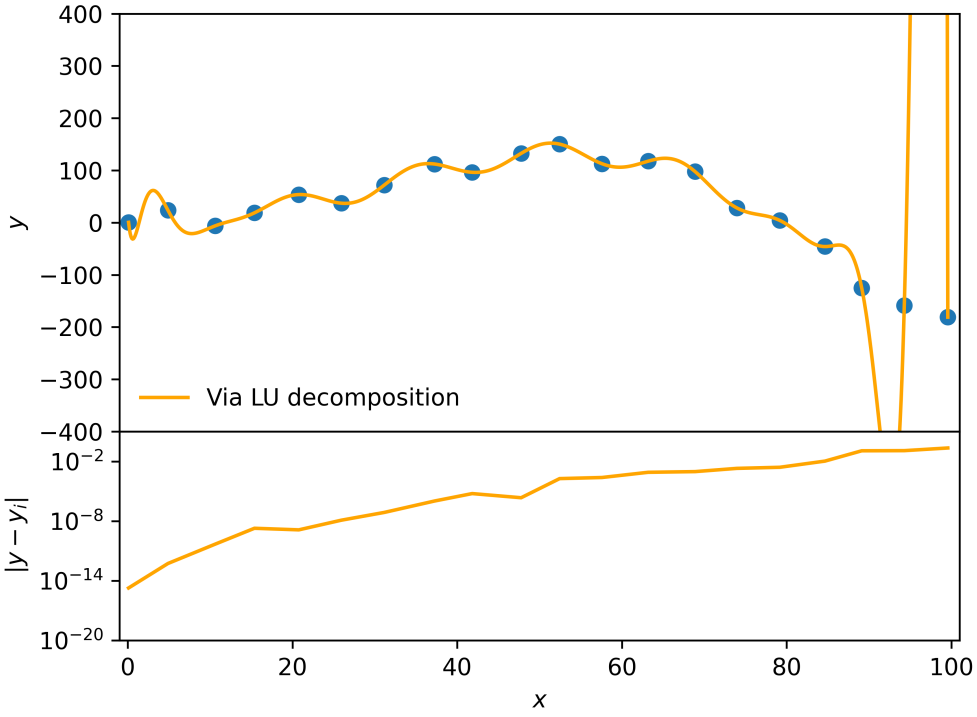

Figure 1: Upper panel: Interpolation on a set of given data points via LU decomposition. The fit is going through all the data points exactly. Nevertheless, it has an oscillating behaviour in correspondence of the last two points. Bottom panel: Absolute difference between the given points y_i and our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

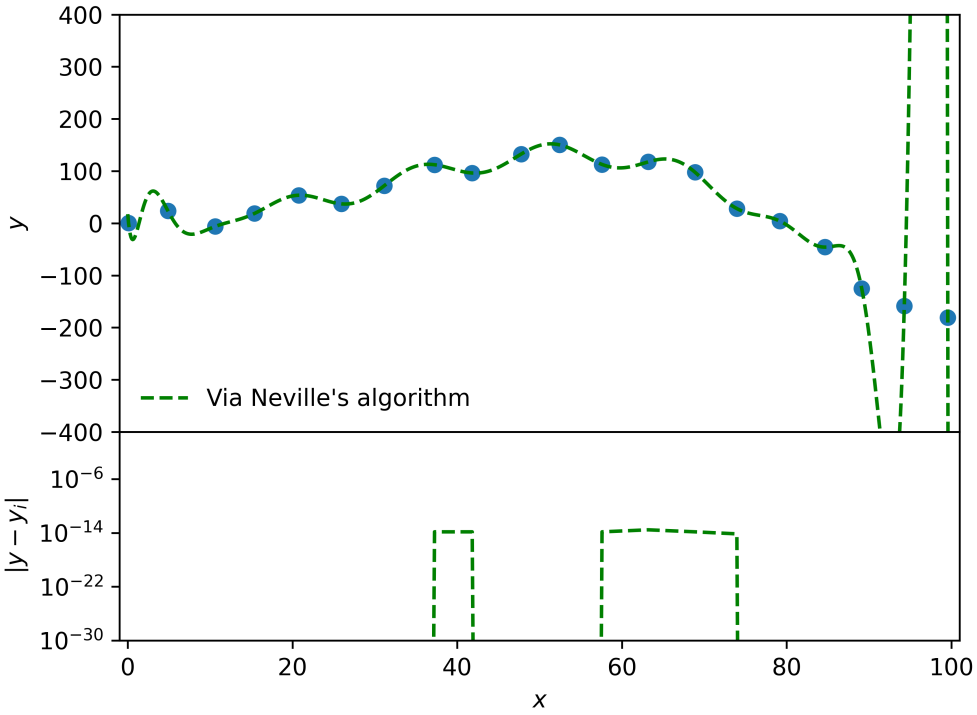

Figure 2: Upper panel: Interpolation on a set of given data points via Neville's algorithm. The polynomial obtained fits the data points well in the range considered, in the same way the interpolation of LU decomposition does. Bottom panel: Absolute difference between the given points y_i and our result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

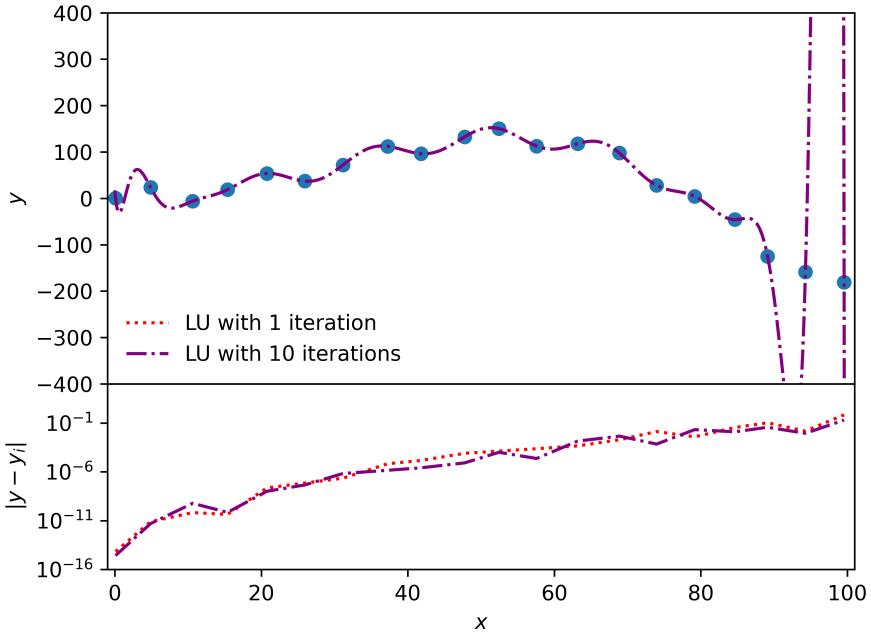

Figure 3: Upper panel: Interpolation on a set of given data points via LU decomposition, with iterative improvement of the found solution (done with 1 and 10 iterations). Both the fits are going through all the data points exactly, hence the overlap. Bottom panel: Absolute difference between the given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