Numerical Recipes: Hand-In 1

Rens Kievit (s1948415)

March 8, 2023

Abstract

In this report we present the problems, solutions and scripts for the exercises from the first handout for the course Numerical Recipes.

Plotting styles in this report are set using the following code

```
def set_styles():
    """For consistent plotting scheme"""
    plt.style.use('default')
    mpl.rcParams['axes.grid'] = True
    plt.style.use('seaborn-darkgrid')
    mpl.rcParams['font.family'] = 'serif'
    mpl.rcParams['lines.linewidth'] = 1.5
    mpl.rcParams['font.size'] = 14
```

Listing 1: Matplotlib Plotting Styles

1 Poisson Distribution

The Poisson probability distribution for any given positive integer k and positive mean λ is given as

$$P_\lambda(k) = \frac{\lambda^k \exp^{-\lambda}}{k!}. \tag{1}$$

This distribution is normalized such that $\sum_{k=0}^{\infty} P_\lambda(k) = 1$. This distribution is implemented in an existing Python package as `scipy.stats.poisson.scipy`, but in this report we will implement this distribution using only pure Python, and `numpy.exponent`. For memory reasons we will limit the variables to only use 32 bits. As a test of the implementation, we will compute $P_\lambda(k)$ for the values presented in Table 1.

λ	k
1	0
5	10
3	21
2.6	40
101	200

Table 1: λ and k values at which $P_\lambda(k)$ is evaluated in this report.

Before we start programming, we can already see a potential memory issue in the parameters at which we want to evaluate the probability distribution. For $k = 200$ we have to compute $200! = 7.9 \times 10^{374}$ which is a lot larger than the maximum size of a 32-bit signed integer, $2^{31} = 2.1 \times 10^9$. This issue starts even earlier, the factorial function overtakes this maximum size already at $k \sim 12$. To combat this potential overflow error we will instead compute $\ln P_\lambda(k)$. To denote this in a smart way, we have to rewrite the factorial by realizing that $k! = \prod_{i=1}^{k} i$. Therfore $\ln k! = \ln \prod_{i=1}^{k} i = \sum_{i=1}^{k} \ln(i)$. We apply this trick only if $k > 5$ as a generous underlimit for when overflow starts becoming an issue.

$$\ln P_\lambda(k) = \ln\left(\frac{\lambda^k \exp^{-\lambda}}{k!}\right) = k \cdot \ln(\lambda) - \lambda - \sum_{i=1}^{k} \ln(i) \tag{2}$$

Combining all of the above we can code this as such:

```
import numpy as np
import scipy.stats

def factorial(k, dtype=np.int64):
    """Computes the factorial k! for any integer k"""
    if k == 0:
        return dtype(1)
    else:
        prod = dtype(1)
        for i in range(1, k+1):
            prod *= dtype(i)
        return prod

def log_factorial(k, dtype=np.int64):
    """Computes the factorial k! for any integer k in log-space"""
    if k == 0:
```

```
        return dtype(1)
    else:
        logsum = dtype(0)
        for i in range(1, k+1):
            logsum += np.log(i, dtype=dtype)
        return logsum


def poisson(lmda, k, dtype_int, dtype_float):
    """Returns the Poisson function with mean lamda, evaluated at
    the integer point k.

        P_lmda(k) = lmda^k * exp(-lmda) / k!

    If k is too large we compute the function evaluated at k in log
    -space first, and then return the exponent of the
    result to dodge overflow errors, which occur at k ~ 12. The
    converted function to log space looks like:

        ln(P_lmda(k)) = k*ln(lmda) - lmda - sum_1^k(ln(i))

    All function calls are wrapped in dtypes to limit the amount of
     memory usage
    """
    if k > 5:
        res = dtype_float(
            dtype_int(k) * np.log(lmda, dtype=dtype_float) -
    dtype_float(lmda) - log_factorial(k, dtype=dtype_float))
        res = np.exp(res, dtype=dtype_float)
    else:
        res = dtype_float(((lmda ** k) * np.exp(-lmda, dtype=
    dtype_float)) / factorial(k, dtype=dtype_int))
    return res


def compute_poisson_values(dtype_int=np.int32, dtype_float=np.
    float32):
    """Compute the Poisson values for the points provided in Q1 of
     hand-in assignment 1.
    For testing purposes we compare our values to those from an
    official library
    """

    values = [[1, 0], [5, 10], [3, 21], [2.6, 40], [101, 200]]
    poisson_prob_ar_self = np.zeros(len(values))
    poisson_prob_ar_scipy = np.zeros(len(values))
    for i, vals in enumerate(values):
        lmda = dtype_float(vals[0])
        k = dtype_int(vals[1])

        poisson_prob_ar_self[i] = poisson(lmda, k, dtype_int,
    dtype_float)
        poisson_prob_ar_scipy[i] = scipy.stats.poisson.pmf(k, lmda)

    # Print table in Latex ready format
    table = """$\lambda$ & k & Self & Scipy \\\\\n\hline\n"""

    for i in range(poisson_prob_ar_self.shape[0]):
        table += f'{values[i][0]} & {values[i][1]} & {
    poisson_prob_ar_self[i]:.6E} & {poisson_prob_ar_scipy[i]:.6E}
    \\\\\n'
    table = table[:-4] # Remove the last two \\ for a nicer Latex
```

```
    file
67
68
69      with open('results/poisson_tab.txt', 'w') as file:
70          file.write(table)
71
72
73  def main():
74      compute_poisson_values()
75
76
77  if __name__ == '__main__':
78      main()
```

Listing 2: All code for Poisson Distribution calculations

As visible in the code we compute both the Poisson distribution evaluated by our code, and by the Scipy implementation and compare them in a table. We present these results in Table 2. We can see that for low values of λ, k the results match up exactly up to siz digits. When moving towards higher values we can see that the "true" Scipy values and our estimates are the same up to the fourth decimal digit. In both cases we only consider the number, not the exponent.

λ	k	Self	Scipy
1	0	3.678795E-01	3.678794E-01
5	10	1.813280E-02	1.813279E-02
3	21	1.019340E-11	1.019340E-11
2.6	40	3.615103E-33	3.615119E-33
101	200	1.269728E-18	1.269531E-18

Table 2: Results of the Poisson distribution code presented in this work, and the implementation from `scipy.stats.poisson.pmf`.

2 Vandermonde Matrix

We start by presenting the background code and miscelaneous functions for matrices and linear algebra.

```
class Matrix():
    """Matrix Class for linear algebra"""

    def __init__(self, values=None, num_rows=None, num_columns=None
    , dtype=np.float64):
        """Check inputs and create a corresponding matrix or vector
    """
        if values is not None:
            self.num_rows = values.shape[0]
            try:
                self.num_columns = values.shape[1]
            except IndexError:
                self.num_columns = 1
                #print(f'Warning! Values has dim=1. Making vector
    with shape ({self.num_rows}, {self.num_columns})')
            if type(values) == np.ndarray:
                self.matrix = np.array(values, dtype=dtype)
            else:
                print(f'Datatype of values {type(values)} not
    recognized. Initializing matrix with zeros.')
                self.matrix = np.zeros((num_rows, num_columns),
    dtype=dtype)
        else:
            self.num_rows = num_rows
            self.num_columns = num_columns
            self.matrix = np.zeros((num_rows, num_columns))

        # Use row order to track rows that have been shuffled
        self.row_order = np.arange(self.num_rows)

    def swap_rows(self, idx1, idx2):
        """Extract rows from a matrix, and switch them. Track the
    change in row_order"""
        self.matrix[[idx1, idx2]] = self.matrix[[idx2, idx1]]
        self.row_order[[idx1, idx2]] = self.row_order[[idx2, idx1]]

    def scale_row(self, idx, scalar):
        """Multiply all elements of row {idx} by a factor {scalar}
    """
        self.matrix[idx] *= scalar

    def add_rows(self, idx1, idx2, scalar):
        """Add row {idx2} multiplied by scalar to row {idx1}"""
        self.matrix[idx1] += scalar * self.matrix[idx2]
```

Listing 3: Code for the Matrix class used throughout this work

```
def check_solution(A, x, b, epsilon=1e-10):
    """Checks a proposed solution to a system of linear equations
    by computing Ax - b and checking
       if all elements are below some threshold"""
    return (np.abs(mat_vec_mul(A, x) - b) < epsilon).all()
def mat_vec_mul(mat, vec):
    """Computes the product between a matrix of shape MxN and a
    vector of shape Nx1.

    Inputs:
```

```
        mat: ndarray of shape MxN
        vec: ndarray of shape Nx1

    Outputs
        res: The result of mat x vec, ndarray of shape Mx1

    """

    res = np.zeros_like(vec)
    for i in range(mat.shape[0]):
        for j in range(mat.shape[1]):
            res[i] += mat[i, j] * vec[j]

    return res


def mat_mat_mul(A, B):
    """Computes the product between a matrix of shape MxN and another matrix of shape NxL.

    Inputs:
        A: ndarray of shape MxN
        B: ndarray of shape NxL

    Outputs
        res: The result of A x B, ndarray of shape MxL
             If both M and L are one, the result is a single float

    """

    try:
        equality = (A.shape[0] == B.shape[1])
        if not equality:
            raise ValueError(f"Shape of A ({A.shape}) does not match shape of B ({B.shape}) for matrix multiplication.")
    except IndexError:
        pass

    res = np.zeros((A.shape[1], B.shape[0]))
    for i in range(res.shape[0]):
        for j in range(res.shape[1]):
            for k in range(A.shape[0]):
                res[i, j] += A[i, k] * B[k, j]

    return res
```

Listing 4: Matrix and vector multiplication functions, and a function to check if a solution x to the system of equations Ax = b is correct.

```
def import_data():
    """Import the Vandermonde data and place it in a Vandermonde matrix"""
    data = np.genfromtxt(os.path.join(sys.path[0], "Vandermonde.txt"), comments='#', dtype=np.float64)
    x = data[:, 0]
    y = data[:, 1]
    x_interp = np.linspace(x[0], x[-1], 1001)  # x values to interpolate at

    V = np.zeros((len(x), len(x)))
    # Fill out the Vandermonde Matrix as V_ij = x_i^j
    for j in range(len(x)):
```

```
        V[:, j] = x**j

    return V, x, y, x_interp


def compute_polynomial(x, coeff):
    """Computes a polynomial evaluated at all x, with coefficients coeff"""
    y = np.zeros_like(x)
    for i in range(len(coeff)):
        y += coeff[i] * x**i
    return y
    """Code for Assignment 2 from Handout 1"""
    V, x, y, x_interp = import_data()

    # a. LU Decomposition
    LU_y, LU_y_at_x = LU_decomposition(V, x, y, x_interp)

    # b. Neville's Algorithm
    neville_y, neville_y_at_x = neville_fit(x, y, x_interp)

    # c. Iterative LU improvement
    LU_y_iterative, LU_y_at_x_iterative = LU_decomposition(V, x, y, x_interp, num_LU_iterations)

    # Plot Data
    set_styles()
    fig, (ax0, ax1) = plt.subplots(2,1, sharex=True, figsize=(10,8))
    ax0.plot(x, y, marker='o', linewidth=0)
    ax0.plot(x_interp, LU_y, c='C1', label='LU Decomposition')
    ax0.plot(x_interp, neville_y, c='C2', ls='dashed', label="Neville's Algorithm")
    ax0.plot(x_interp, LU_y_iterative, c='C3', ls='dotted', label=f'{num_LU_iterations} Times Iterated LU')
    ax0.legend()

    ax1.plot(x, np.abs(y - LU_y_at_x), c='C1', marker='o', lw=0)
    ax1.plot(x, np.abs(y - neville_y_at_x), c='C2', marker='o', lw=0)
    ax1.plot(x, np.abs(y - LU_y_at_x_iterative), c='C3', marker='o', lw=0)
    ax1.axhline(y=0, c='black', ls='--', alpha=0.6)
    ax1.set_yscale('log')

    plt.xlim(-1, 101)
    ax0.set_ylim(-400, 400)
    ax1.set_xlabel('$x$')
    ax0.set_ylabel('$y$')
    ax1.set_ylabel(r'Absolute $\Delta y$')

    plt.suptitle("Lagrange Polynomial Estimations"          )
    plt.savefig('results/vandermonde_fitresults.png', bbox_inches='tight')
    #plt.show()
```

Listing 5: Code to import the data and apply the algorithms described throughout this section

In this section we will work with a Vandermonde matrix V, which is defined as an NxN matrix with elements $V_{i,j} = x_i^j$ for rows i and columns j ranging from 0 to $N-1$. For any set with N (x_i, y_i) points, we can populate the

Vandermonde matrix and solve the set of linear equations $Vc = y$ to find the coefficients c of the unique polynomial passing through all (x_i, y_i) points. Here, we will apply two different methods to find this polynomial and compare the results.

We will begin by introducing both methods, and providing the code used to implement them in Python. Afterwards we present the results of our implementation, and a discussion of these results.

2.1 LU Decomposition

Solving an equation of the form $Vc = y$ using the LU decomposition method means first decomposing the matrix V into an L matrix, which only has non-zero elements in the lower triangle, and a U matrix, which only has non-zero elements in the upper triangle, such that $LU = V$. We perform the decomposition using our implementation of Crout's Algorithm, described by the `lu_decomposition` function given below. We first pivot the matrix column-wise from left to right by swapping rows such that the elements on the diagonal are as large as possible, chosen from the values on or below the diagonal. For determining the maximum value we use the implicit pivotting method, which means we weigh each element by the absolute maximum value of its row. With each row swap of the matrix V, we also swap the indices in `V.row_order` which later will be used to swap the elements of the constraints y.

After pivotting the matrix V, we apply the equations of LU decomposition as presented in the lecture to transform V into the matrix containing the elements of L, $\alpha_{i,j}$, below the diagonal and the elements of U, $\beta_{i,j}$, on and above the diagonal. With this LU matrix we then use the forward substitution method introduced in the lectures to solve $Lz = y$, and then backward substitution to solve $Uc = z$ to find the coefficients c describing the 19th order polynomial going through all 20 data points.

```
1  def LU_decomposition(V, x, y, x_interp, num_iterations=0):
2      """Performs LU decomposition on the Vandermonde matrix V to
       find the matrices L and U such that L*U = V
3      It then uses these decomposed matrices to solve for the
       coefficient of the polynomial that goes through
4      all points x_i, y_i. If num_iterations > 0, we reapply the LU
       matrix to \delta y = Vc' - y. """
5      LU = lu_decomposition(V)
6      LU_coefficients = solve_lineqs_lu(LU, y)
7
8
9      # Evaluate the Vandermonde polynomial on the whole smooth range
       , and at the 20 data points
10     LU_polynomial = compute_polynomial(x_interp, LU_coefficients.
       matrix)
11     LU_poly_at_xdata = compute_polynomial(x, LU_coefficients.matrix
       )
12
13     # Iterative Improvement
14     for i in range(num_iterations):
15         delta_y = LU_poly_at_xdata - y
16         delta_coefficients = solve_lineqs_lu(LU, delta_y)
17         LU_coefficients.matrix -= delta_coefficients.matrix
18
```

```
        LU_polynomial = compute_polynomial(x_interp, LU_coefficients.matrix)
        LU_poly_at_xdata = compute_polynomial(x, LU_coefficients.matrix)

    return LU_polynomial, LU_poly_at_xdata
```

Listing 6: Code to apply LU decomposition using Crout's Algorithm to the provided data

```
def determine_implicit_pivot_coeff(mat):
    """Determines the coefficients for implicit pivotting in Crout's Algorithm. It does this by finding
       the absolute maximum value of each row in the matrix, and storing its inverse.

       NOTE: Requires a Matrix object (this script) as input. This ensures correspondence with row_order
    """
    row_max_inverse = np.zeros(mat.num_rows)
    for i in range(mat.num_rows):
        row = mat.matrix[i]
        row_max = row[np.argmax(np.abs(row))]
        row_max_inverse[i] = 1. / row_max

    return row_max_inverse


def lu_decomposition(coefficients, implicit_pivoting=True):
    """Decomposes a matrix into:
        -L: A matrix with non-zero elements only in the lower-triangle, and ones on the diagonal
        -U: A matrix with non-zero elements only in the upper-triangle, including the diagonal
       These matrices are presented and stored into one.
       The decomposition is done using Crout's Algorithm
    """
    A = Matrix(values=coefficients)
    if implicit_pivoting:
        row_max_inverse = determine_implicit_pivot_coeff(A)

    imax_ar = np.zeros(A.num_columns)
    # First pivot the matrix
    for i in range(A.num_columns):
        # A.matrix[i:, i] selects all elements on or below the diagonal
        if implicit_pivoting:
            pivot_candidates = A.matrix[i:, i] * row_max_inverse[i:]
        else:
            pivot_candidates = A.matrix[i:, i]

        pivot_idx = i + np.argmax(np.abs(pivot_candidates))
        imax_ar[i] = pivot_idx
        A.swap_rows(i, pivot_idx)

    for i in range(A.num_columns):
        # A.matrix[i:, i] selects all elements on or below the diagonal
        diag_element = A.matrix[i, i]  # Use to scale alpha factors

```

```
44          for j in range(i + 1, A.num_rows):  # This leaves a zero at
        the end, not the best fix this!
45              A.matrix[j, i] /= diag_element
46              for k in range(i + 1, A.num_rows):  # j+1):
47                  A.matrix[j, k] -= A.matrix[j, i] * A.matrix[i, k]
48
49      return A
50  def solve_lineqs_lu(LU, b):
51      """Performs the steps to solve a system of linear equations
        after a matrix A has been LU decomposed. It
52      does this by first applying forward substitution to solve Ly =
        b, and then applies backward subsituttion
53      to solve Ux = y.
54
55      Inputs:
56          LU: The decomposed L and U matrices, stored in a single
        Matrix instance
57          b: The constraints of the linear equations, ndarray
58
59      Outputs:
60          x: Matrix instance containing the solution such that Ax = b
61      """
62
63      x = Matrix(values=b)
64      # Begin by swapping the x's in the right order
65      x.matrix = x.matrix[LU.row_order]
66
67      # Forward Subsitutions. Solves Ly = b
68      for i in range(0, x.num_rows):
69          x.matrix[i] -= np.sum(LU.matrix[i, :i] * x.matrix[:i])
70
71      # Backward Substitutions. Solves Ux = y
72      for i in range(x.num_rows-1, -1, -1):
73          x.matrix[i] = (1./LU.matrix[i,i])*(x.matrix[i] - np.sum(LU.
        matrix[i, i+1:]*x.matrix[i+1:]))
74
75      return x
```

Listing 7: Algorithms for LU decomposition using Crout's Algorithm and a linear equation solver

2.2 Iterative LU Decomposition

In theory the coefficients c we find using the LU decomposition method described above should exactly solve the equation $Vc = y$. In reality however, there is a small error on the machine estimates of the coefficients, giving us $c' = c + \delta c$ instead. This means we actually found $Vc' = V(c + \delta c) = y + \delta y$ and therefore we can write $V\delta c = \delta y = Vc' - y$. This shows us that we can solve $V\delta c = \delta y$ to find an improvement to our solution c'.

The great thing is that we do not have to decompose V again because we have already done that for the initial estimate of c. Therefore, this iterative improvement can be done for relatively little computing power. This iterative improvement process is encoded in the for loop in the function `LU_decomposition` in `vandermonde.py`. In this report we will apply this iterative improvement process ten times, and included it alongside our other results.

2.3 Neville's Algorithm

Due to the similiarity of the polynomial based on the Vandermonde matrix to the Lagrange polynomial, we can also use a polynomial interpolator to find the coefficients of the Lagrange polynomial that goes through all data points (x_i, y_i). To find this polynomial we have implemented Neville's Algorithm on all 20 data points simultaneously as presented below. In this specific implementation we start with a bisection algorithm to find the neighbours of the x values we want to interpolate. This is not necessary in this specific implementation because we always use all available data, however it is left in for the sake of generality.

```
def split_list(a):
    """Splits a list in two halves and returns both.
        -If the length of the array is even, both halves are equally long.
        -If the length of the array is odd, the second half is one longer"""
    half = len(a) // 2
    return a[:half], a[half:]


def bisection(x, y, t, M):
    """Applies the bisection algorithm at order M in 1D.

    Inputs:
        x: The x-values of the data points. Assumed strictly increasing
        y: The y-values of the data points
        t: The x-value of the point to identify sample points for
        M: Number of sample points to return

    Returns:
        interp_points: The M points nearest to t along the x-axis
        extrapolated: boolean, True if t lies outside the given range of the x data
    """
    # Use another variable than x to make sure we don't accidentally mess with
    # the real xdata list due to Python shenanigans
    adjacent_idxs = np.arange(0, len(x))

    # Check if we're dealing with extrapolation
    if t < x[0]:
        #print(f"Warning: Extrapolation. {t} lies below x data points")
        return np.arange(0, M), True
    elif t > x[-1]:
        #print(f"Warning: Extrapolation. {t} lies above x data points")
        return np.arange(len(x) - M, len(x)), True

    # Keep doing the below steps until we find the two points adjacent to t
    while True:
        left, right = split_list(adjacent_idxs)
        if (t >= x[left[0]]) & (t < x[right[0]]):
            # Check if we're exactly on the boundary between two sets
            if t > x[left[-1]]:
                adjacent_idxs = [left[-1], right[0]]
                break
```

```
42              adjacent_idxs = left
43          else:
44              adjacent_idxs = right
45
46          if len(adjacent_idxs) < 3:  # then adjacent points are identified
47              break
48
49      # In the following if-statements we check if the point t is "too close to the edge". This is the
50      # case if there are fewer than M/2 points to the left or right of t.
51      # If the point is in the middle and M is odd, we add an extra point to the right
52      if adjacent_idxs[0] < M//2:
53          min_idx = 0
54          max_idx = M-1
55      elif adjacent_idxs[0] > (len(x) - M):
56          min_idx = len(x) - M
57          max_idx = len(x) - 1
58      else:
59          min_idx = int(adjacent_idxs[0] - np.floor((M-2)/2))
60          max_idx = int(adjacent_idxs[1] + np.ceil((M-2)/2))
61
62      # Check if bisection worked properly
63      if not ((x[min_idx] <= t) & (t <= x[max_idx])):
64          raise ValueError(f'Mistake in bisection. {t} not between {x[min_idx]} and {x[max_idx]}')
65
66      interp_points = np.arange(min_idx, max_idx+1)
67      return interp_points, False  # +1 to include max_idx
68
69
70  def nevilles_equation(t, P1, P2, x1, x2):
71      """Given two P and x values, computes Neville's Equation. This is used for the polynomial
72      interpolation function. The equation is:
73
74      H(x) = ((x_j - x) * F_i(x) + (x - x_i) * G_j(x)) / (x_j - x_i)
75
76      Inputs:
77          t : x, point to be interpolated
78          P1: F_i(x)
79          P2: G_j(x)
80          x1: x_i
81          x2: x_j
82
83      Outputs:
84          H_t: H(x)
85      """
86      top = (x2 - t) * P1 + (t - x1) * P2
87      return top / (x2 - x1)
88
89
90  def poly_interpolator(xdata, ydata, t, M):
91      """This function first makes use of the bisection algorithm to identify M data points surrounding
92      the point to be interpolated, and then applies Neville's Algorithm to estimate its y-value and
93      the uncertainty on that estimation.
94
95      Note: While this function is called 'polynomial interpolator',
```

```
    it does accept values outside of the
96          provided range of xdata, and therefore can extrapolate.
    But the reported accuracy of the
97          estimation drops off quickly outside the range.
98
99      Inputs:
100         xdata: N data points along the x-axis
101         ydata: N data points along the y-axis
102         t    : The points along the x-axis we want to interpolate,
    should always be provided as a
103                 list or ndarray for indexing purposes
104         M    : The order of the polynomial to fit
105
106     Outputs:
107         y_inter:   Array of the estimated y-values corresponding to
     t
108         unc_inter; Array of the estimated uncertainties on y_inter.
     These are estimated as the absolute
109                    difference between P_array[0] just after, and
    just before the final loop.
110 """
111     y_inter = np.zeros_like(t)
112     unc_inter = np.zeros_like(t)
113     for i in range(len(t)):
114         interp_points, interpolated = bisection(xdata, ydata, t[i],
     M)
115         P_array = np.array(ydata)[interp_points]
116         x_points = np.array(xdata)[interp_points]
117
118         for k in range(1, M):
119             for j in range(M - k):
120                 P_array[j] = nevilles_equation(t[i], P_array[j],
    P_array[j + 1], x_points[j], x_points[j + k])
121             if k == M - 2:
122                 previous_val = P_array[0]  # This is e.g. P012, use
     its diff. with P0123 as dy
123         y_inter[i] = P_array[0]
124         unc_inter[i] = np.abs(P_array[0] - previous_val)
125     return y_inter, unc_inter
```

Listing 8: Algorithms for polynomial interpolation using Neville's Algorithm

2.4 Results

We have introduced all the algorithms which we have used to find the polynomial going through all of the provided data points. We plot these data points and each of the three polynomials in the top panel of Figure 1. In the bottom panel we present the base 10 logarithm of the absolute difference between the estimations of each of the three implementations and the true data values evaluated at each data point i. We note that a few points are missing for Neville's Algorithm. This is because at these points the polynomial derived using Neville's Algorithm and the true values exactly match (up to machine error) and therefore their difference is zero. Taking the logarithm of zero is impossible and therefore returns no result.

Initially we can see that each of the three polynomials go through each of the twenty data points in almost exactly the same manner. It is impossible to see any difference by eye on a linear scale. However from the bottom panel we can see that there is a difference between the LU Decomposition and Neville's

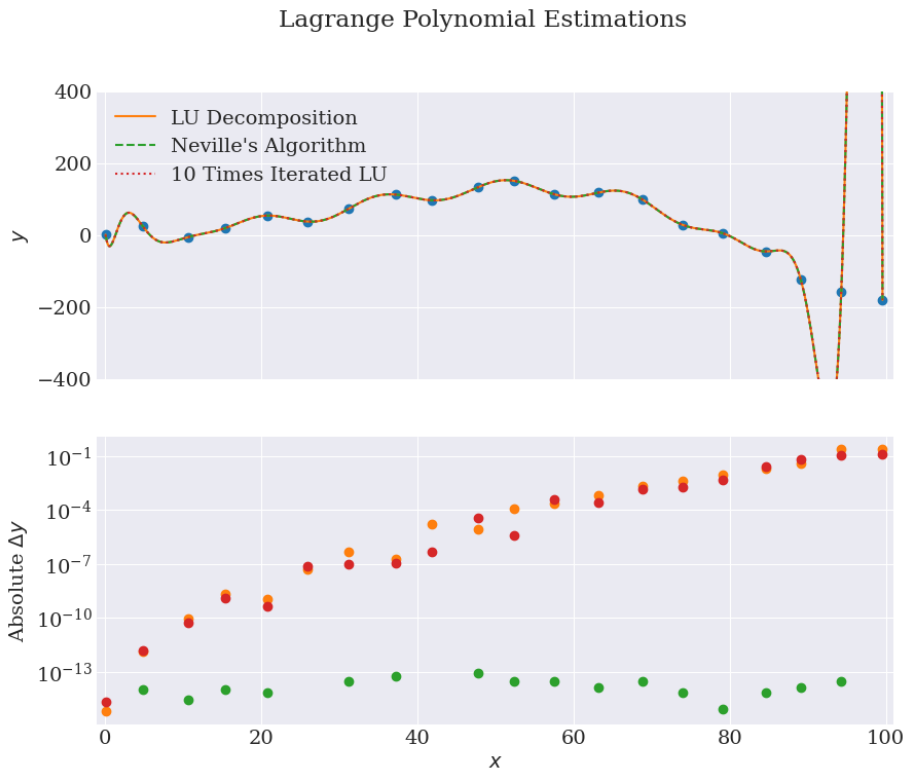

Figure 1: Results of the polynomial estimations of the data. A full description of the algorithms and a full discussion of the text is given in text. *top*: Provided twenty data points combined with the three estimated polynomials going through each of these points. *bottom*: Logarithmic absolute difference between the estimated polynomials and the true data values evaluated at each x_i.

Algorithm method when evaluated at the x_i. The error for the polynomial interpolator is significantly lower for all x, and the error for the LU decomposition implementation increases of approximately thirtheen orders of magnitude throughout the entire dataset. Additionally we see virtually no improvement in error after attempting to iteratively improve the estimate with LU decomposition.

The consistently low values for absolute difference through Neville's Algorithm make sense due to the definition of the equation used for the estimations. Neville's equation is mostly based on the true values, y_i and the distance between the corresponding x_i and the point we want to interpolate. If that distance is very small, then the deviation from y_i will by definition be very small as well.

The pattern of increasing absolute difference in both LU decomposition implementations seems weird at first sight. But this most likely arises from the increasingly 'weird' behaviour of the polynomial that is clearly visible in the top panel. It has immensly steep slopes surrounding the final three data points, which means the estimation of a y value can be very sensitive to a tiny variation in x, leading to larger uncertainties. The absence of any improvement for the iterated LU decomposition method could have arised from the large scale difference between the y and Δy. After one iteration we already have a realtively good estimation which at its maximum has an uncertainty of ~ 0.1 on a value of 200. Adding additional improvement with this iterative process just adds extra negligble terms that do not significantly change our estimations. We confirmed this by looking at the δc for the last point, which are usually on the order of $\sim 10^{-3}$, too small to be noticed after ten iterations.

2.5 Implementation Speed

As a final comparison between each of the three methods we measure their execution times. To keep the run time of the complete assignment small we set a limit of one minute for this section, which allows 100 iterations of each algorithm. The results are presented in Table 3 and the associated code is presented at the end of this section.

Algorithm	Runtime (ms)
LU Decomposition (1x)	7.11
LU Decomposition (10x)	26.87
Neville's Algorithm	592.62

Table 3: Runtime of each of the three algorithms described in the text expressed in milli-seconds, averaged over 100 iterations each.

We see that the LU decomposition, while slightly less precise as we saw in the previous section, is faster than Neville's Algorithm by almost two orders of magnitude. Additionaly, we can very clearly see the fact that subsequent iterations of solving a system of linear equations with an LU matrix can be done very quickly. The LU decomposition iterated ten times is only ~ 5 times slower than the non-iterated version. However as we saw in the last section, in this specific case this additional computing time did not result in extra accuracy.

```
def vandermonde_timeit(num_iter=100):
    """Time the execution time of the code snippets above"""
    V, x, y, x_interp = import_data()

    time_LU = timeit.timeit(f"LU_decomposition(V, x, y, x_interp, num_iterations=0)",
                            setup='from __main__ import LU_decomposition, import_data \
                                   \nV, x, y, x_interp = import_data()',
                            number=num_iter)/num_iter

    time_LU_iterative = timeit.timeit("LU_decomposition(V, x, y, x_interp, num_iterations=10)",
                                      setup='from __main__ import LU_decomposition, import_data \
                                             \nV, x, y, x_interp = import_data()',
                            number=num_iter)/num_iter

    time_neville = timeit.timeit("neville_fit(x, y, x_interp)",
                                 setup='from __main__ import neville_fit, import_data \
                                        \nV, x, y, x_interp = import_data()',
                                 number=num_iter)/num_iter

    table = f"""Algorithm & Runtime (ms) \\\\
\hline
LU Decomposition (1x) & {time_LU*1e3:.2f} \\\\
LU Decomposition (10x) & {time_LU_iterative*1e3:.2f} \\\\
Neville's Algorithm & {time_neville*1e3:.2f}"""

    with open('results/vandermonde_timetab.txt', 'w') as f:
        f.write(table)

    #print('almost done.')
    #input()

def main():
    vandermonde_fit()
    print("\tTiming Algorithms..")
    vandermonde_timeit()


if __name__ == '__main__':
    main()
```

Listing 9: Code for the algorithm execution time estimation