NUR A - Assignment 1

Taotao Yang (s4866835)

Dated: February 28, 2026

1 Poisson Distribution

We start with the general expression for Poisson distribution for some positive mean λ and integer k:

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

To prevent overflow/underflow in general, we circumvent factorial calculations via mapping to log space. We therefore rewrite Poisson distribution as:

$$\begin{aligned} \ln\left(P_\lambda(k)\right) &= \ln\left(\frac{\lambda^k e^{-\lambda}}{k!}\right) \\ &= k\ln\lambda - \lambda - \ln k! \\ &= k\ln\lambda - \lambda - \sum_{i=1}^{k+1} \ln i \end{aligned} \tag{2}$$

Therefore, the actual distribution is recovered via:

$$P_\lambda(k) = \exp\left(\ln\left(P_\lambda(k)\right)\right) \tag{3}$$

To ensure input parameter λ and k follows the correct `dtype`, we add error handling to function via simple if-else statements. This step is enforced alongside some value checker for λ and k, where $\lambda <= 0$ or $k <=$ would both result in function raising errors, as the Poisson distribution given would become ill-defined or invalid for these λ and k values. We enforce correct `dtype` via local `dtype` conversion in function. Additionally, for normalized Poisson distribution, we set special case where $k = 0$. That is, Poisson distribution simplifies to:

$$P_\lambda(k) = \frac{\lambda^k e^{-\lambda}}{k!} = \frac{\lambda^0 e^{-\lambda}}{0!} = e^{-\lambda}\ ,\ k = 0 \tag{4}$$

These steps further simplify the calculations. That is, we can recover the actual value of Possion distribution from the log space results via a simple exponential. This is on top of converting a factorial into a summation of terms.

The full code used for this question:

```python
"""
Scripts for assignment 1 question 1
"""

import numpy as np


def Poisson(k: np.int32, lmbda: np.float32) -> np.float32:
    """Calculate the Poisson probability for k occurrences with mean lmbda.
    Parameters:
        k (np.int32): The number of occurrences.
        lmbda (np.float32): The mean number of occurrences.
    Returns:
        np.float32: The probability of observing k occurrences given the mean
            lmbda.
    """
    # we start with the gen form of posson distro P
    # top = lmbda**k * np.exp(-lmbda)
    # bot = np.prod(np.arange(1, k, 1))
    # P = top / bottom

    # we then rewrite P with log space to prevent over/underflow
    # ln(P) = k ln(lmbda) - lmbda - ln(k!)
    # with ln(k!) = sum(ln(i) for i in range(1, k+1))
    # P <-> ln(P) through np.exp()

    # break if k and lmbda doesn't match wanted dtype
    if k.dtype != np.int32 or lmbda.dtype != np.float32:
        raise TypeError(
            f"No matching dtype with k.dtype {k.dtype}, lmbda.dtype {lmbda.dtype}"
        )

    # local dtype enforcement
    k = np.int32(k)
    lmbda = np.float32(lmbda)

    # special case if k = 0 -> k! = 1
    if k == np.int32(0):
        result = np.exp(-lmbda)
    # break if k is neg -> undefined factorial in source fn
    elif k < np.int32(0):
        raise ValueError(f"Invalid k when k < 0, k={k}.")

    # break if lmbda is non negative
    if lmbda <= np.int32(0):
        raise ValueError(f"Invalid lmbda when lmbda <= 0, k={k}.")

    # log space rewrite P -> ln(P)
    # enforce generated k as int32
    # enforce ln(k) in float32
    # np.sum uses the dtype of passed in array as default, outputting float32
    log_factorial = np.sum(
        np.log(np.arange(1, k + 1, 1, dtype=np.int32), dtype=np.float32)
    )
    log_distro = k * np.log(lmbda, dtype=np.float32) - lmbda - log_factorial
    result = np.exp(log_distro, dtype=np.float32)

    return result


def main() -> None:
    # (lambda, k) pairs:
```

```
    values = [
        (np.float32(1.0), np.int32(0)),
        (np.float32(5.0), np.int32(10)),
        (np.float32(3.0), np.int32(21)),
        (np.float32(2.6), np.int32(40)),
        (np.float32(100.0), np.int32(5)),
        (np.float32(101.0), np.int32(200)),
    ]
    with open("./output/a1q1_poisson_output.txt", "w") as file:
        for i, (lmbda, k) in enumerate(values):
            P = Poisson(k, lmbda)
            if i < len(values) - 1:
                file.write(f"{lmbda:.1f} & {k} & {P:.6e} \\\\ \\hline \n")
            else:
                file.write(f"{lmbda:.1f} & {k} & {P:.6e} \n")


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

The results of $P_\lambda(k)$ for selected values of k and λ are shown in Table 1.

λ	k	$P_\lambda(k)$
1.0	0	3.678794e-01
5.0	10	1.813280e-02
3.0	21	1.019340e-11
2.6	40	3.615103e-33
100.0	5	3.100058e-36
101.0	200	1.269418e-18

Table 1: Poisson probability distribution for selected k and λ.

2 Vandermonde Matrix and Interpolation

The $N \times N$ Vandermonde matrix $\mathbf{V}_{i,j}$ has general form for some row i and column j:

$$\mathbf{V}_{i,j} = x_i^j \tag{5}$$

For some data array x, we can obtain the associated elements of $\mathbf{V}_{i,j}$. Therefore, we can find some coefficient array c by solving $\mathbf{V}c = y$, where c forms some unique polynomials that passes through all y points. We can compute y from:

$$y_i = \sum_{j=0}^{N-1} c_j x_i^j \ , \ i \in \{0, 1, \ldots, N-1\} \tag{6}$$

2.1 (2a) LU decomposition

Let us first consider some matrix $\mathbf{A}_{i,j}$. For LU decomposition, we require $\mathbf{A}_{i,j}$ to be square and non-singular. By passing the matrix through shape checks, we can proceed to performing LU decomposition in situ. That is, instead of aving two separate matricies for storing $\mathbf{L}_{i,j}$ and $\mathbf{U}_{i,j}$, we compute for the combined $\mathbf{LU}_{i,j}$ to replace source $\mathbf{A}_{i,j}$.

Following Court's algorithm, we create nested loops where we loop over the column j, with row i looped inside the column loops. Observe the following equation for $\mathbf{LU}_{i,j}$, where we define $\mathbf{L}_{i,j}$ to be unity along its diagonal:

$$\mathbf{L}_{i,j} = \frac{\mathbf{A}_{i,j}}{\mathbf{A}_{j,j}} \ , \ i > j \tag{7}$$

We then perform Gaussian elimination on rows with the computed $\mathbf{L}_{i,j}$ terms. On each row of $\mathbf{A}_{i,j}$, where $i \in \{1, \ldots, i\}$, we subtract the values of said row with $\mathbf{L}_{i,j}$ times of the first row. For example:

$$\mathbf{A}_{i,j} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 3 & 9 & 27 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 6 \\ 0 & 0 & 6 \end{bmatrix} = \mathbf{L}_{i,j}\mathbf{U}_{i,j} \tag{8}$$

Effectively, we decompose a source matrix $\mathbf{A}_{i,j}$ into $\mathbf{LU}_{i,j}$, where $\mathbf{LU}_{i,j}$ is the combined matrix of $\mathbf{L}_{i,j}$ and $\mathbf{U}_{i,j}$. Notice that the row reduction happens at each column iterations across all relavant rows. These matricies follow:

$$\mathbf{A}_{i,j} = \mathbf{L}_{i,j}\mathbf{U}_{i,j} \tag{9}$$

For efficient computation, we want to do operations on the source matrix and return a combined LU matrix instead of two distinct matrices.For some source matrix $\mathbf{A}_{i,j}$, we obtain:

$$\begin{aligned} \mathbf{A}_{i,j} &= \begin{bmatrix} \mathbf{A}_{0,0} & \ldots & \mathbf{A}_{0,N-1} \\ \ldots & \ddots & \vdots \\ \mathbf{A}_{N-1,0} & \ldots & \mathbf{A}_{N-1,N-1} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{L}_{0,0} & & \\ \ldots & \ddots & \\ \mathbf{L}_{N-1,0} & \ldots & \mathbf{L}_{N-1,N-1} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{0,0} & \ldots & \mathbf{U}_{0,N-1} \\ & \ddots & \vdots \\ & & \mathbf{U}_{N-1,N-1} \end{bmatrix} = \mathbf{L}_{i,j}\mathbf{U}_{i,j} \end{aligned} \tag{10}$$

Recall that $\mathbf{L}_{i,j}$ is defined with its diagonal set to unity, we now overwrite $\mathbf{L}_{i,j}$ with $\mathbf{U}_{i,j}$. This creates a combined matrix for LU decomposition results, with diagonal axis of $\mathbf{L}_{i,j}$ overwritten with the diagonal entry of $\mathbf{U}_{i,j}$:

$$\begin{bmatrix} 1 & & \\ \dots & \ddots & \\ \mathbf{L}_{N-1,0} & \dots & 1 \end{bmatrix} \begin{bmatrix} \mathbf{U}_{0,0} & \dots & \mathbf{U}_{0,N-1} \\ & \ddots & \vdots \\ & & \mathbf{U}_{N-1,N-1} \end{bmatrix} \Rightarrow \begin{bmatrix} \mathbf{U}_{0,0} & \dots & \mathbf{U}_{0,N-1} \\ \dots & \ddots & \vdots \\ \mathbf{L}_{N-1,0} & \dots & \mathbf{U}_{N-1,N-1} \end{bmatrix} = \mathbf{LU}_{i,j} \tag{11}$$

Consider source matrix $\mathbf{A}_{i,j}$, we have general expression for the linear system with variable vector $\vec{x}$ and value vector b:

$$\mathbf{A}_{i,j}\vec{x} = \vec{b} \Rightarrow \mathbf{LU}_{i,j}\vec{x} = \vec{b} \tag{12}$$

We can solve for coefficient c using the following subsitution methods. First, we set $\mathbf{U}_{i,j}\vec{x} = \vec{y}$ and solve for $\vec{y}$. Then, we set $\mathbf{L}_{i,j}\vec{y} = \vec{b}$ and solve for $\vec{x}$. Effectively, we use:

$$\mathbf{LU}_{i,j}\vec{x} = \vec{b} \Rightarrow \begin{cases} \mathbf{L}_{i,j}\vec{x} = \vec{y} \\ \mathbf{U}_{i,j}\vec{y} = \vec{x} \end{cases} \tag{13}$$

Recall how $\mathbf{LU}_{i,j}$ is the combined matrix of $\mathbf{L}_{i,j}$ and $\mathbf{U}_{i,j}$. We can therefore extract component matrix values via careful indexing rules:

$$\mathbf{L}_{i,j} \ , \ i > j \ \ ; \ \ \mathbf{U}_{i,j} \ , \ i \leq j \tag{14}$$

We extract the terms for forward and backward substitution via:

$$\begin{aligned} \vec{y}_i &= \frac{1}{\mathbf{L}_{i,i}} \left(\vec{b}_i - \sum_{j=0}^{i-1} \vec{y}_j \mathbf{L}_{i,j} \right) \\ \vec{x}_i &= \frac{1}{\mathbf{U}_{i,i}} \left(\vec{y}_i - \sum_{j=i+1}^{n-1} \mathbf{U}_{i,j} \vec{x}_j \right) \end{aligned} \tag{15}$$

Therefore, we obtain $\vec{x}$ as c. For more information, see listed comments in code lisitng sections. In the provided plotting functions, we plot the polynomial fit from LU decomposition against source data points, with residual terms represented as absolute error on log scale.

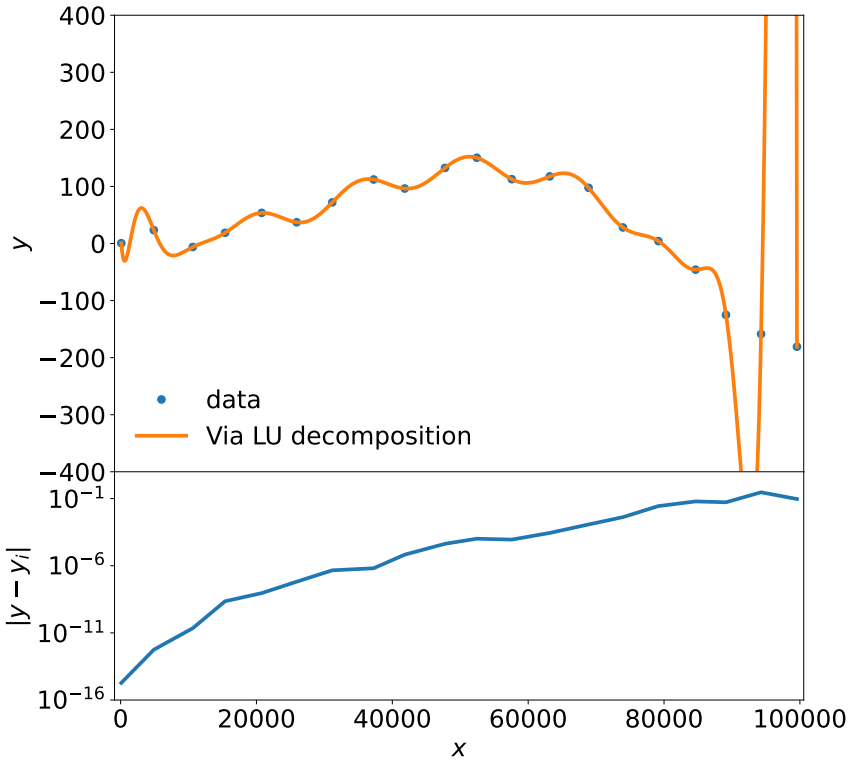

Figure 1: Polynomial fit evaluated using LU decomposition. Top: data points and interpolated curve. Bottom: absolute error at the data points on a log scale.

2.2 (2b) Neville's algorithm

Moving onward, we implement Neville's algorithm for interpreting $\vec{y}$ from $\vec{x}$. The aim of Neville's algorithm is to interpolate the values of the entire fucntion from some given (x, y) data points. To achieve this, we generate a table $\mathbf{p}_{i,j}$ to store the polynomials. Let us again consider a 3×3 matrix for intuition:

$$\mathbf{p}_{i,j} = \begin{bmatrix} \mathbf{p}_{0,0} & \mathbf{p}_{0,1} & \mathbf{p}_{0,2} \\ & \mathbf{p}_{1,1} & \mathbf{p}_{1,2} \\ & & \mathbf{p}_{2,2} \end{bmatrix} \tag{16}$$

Intuitivly, we see that $\mathbf{p}_{i,i}$ terms match with source (x, y) pais. That is, $\mathbf{p}_{i,i} = y_i$. Therefore, we can interpolate for some $x = k$ as:

$$\mathbf{p}_{i,j} = \frac{(k - x_i)\, p_{i+1,j} - (k - x_j)\, p_{i,j-1}}{x_j - x_i} \ , \ k \in (\min(x), \max(x)) \tag{17}$$

However, we quickly notice that enforcing loops over rows is not ideal, as we would be computing for $\mathbf{p}_{0,2}$ prior to its dependencies $\mathbf{p}_{0,1}$ and $\mathbf{p}_{1,2}$are resolved. That is, we simply shift the matrix

to the following form such that:

$$\mathbf{p}_{i,j} = \begin{bmatrix} \mathbf{p}_{0,0} & \mathbf{p}_{0,1} & \mathbf{p}_{0,2} \\ \mathbf{p}_{1,0} & \mathbf{p}_{1,1} & \\ \mathbf{p}_{2,0} & & \end{bmatrix} \tag{18}$$

We can now interpolate $\mathbf{p}_{0,1}$ from $\mathbf{p}_{0,0}$ and $\mathbf{p}_{1,0}$. Similarly, we obtain $\mathbf{p}_{1,1}$ from $\mathbf{p}_{1,0}$ and $\mathbf{p}_{2,0}$. The indexing is now shifted to:

$$\mathbf{p}_{i,j} = \frac{(k - x_i)\mathbf{p}_{i+1,j-1} - (k - x_{i+j})\mathbf{p}_{i,j-1}}{x_{i+j} - x_i} \tag{19}$$

Naturally, we index out $\mathbf{p}_{0,-1}$ term of the resulting polynomial table to obtain the evaluated y value at some $x = k$. And by performing this opearation repeatedly, we can construct some evaluate $\vec{y}$ from $\vec{k}$, which would be used for residual calculations.

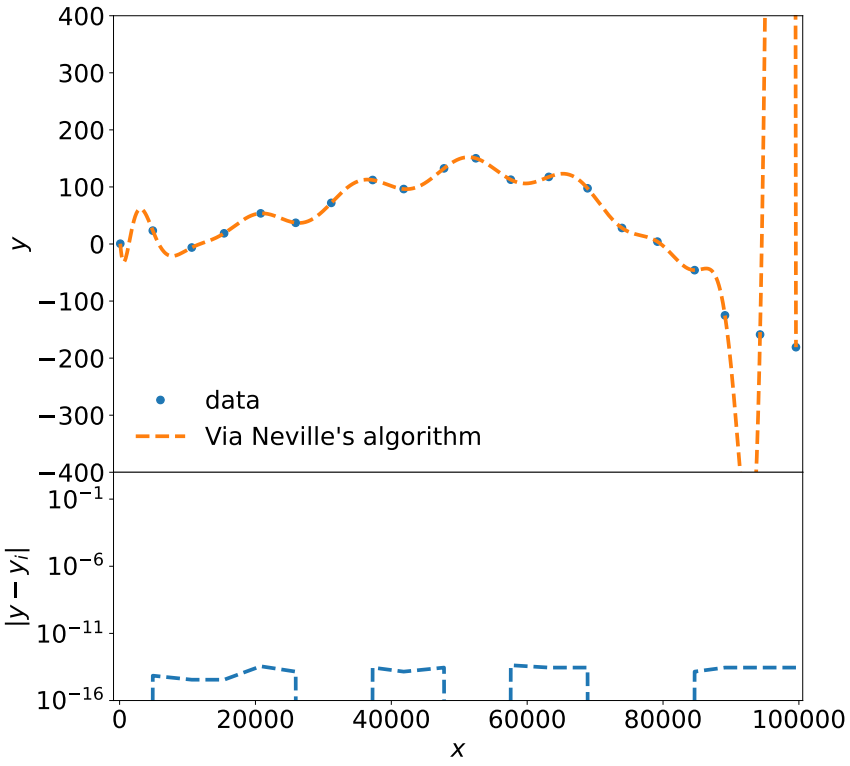

Figure 2: Interpolation using Neville's algorithm. Top: data points and interpolated curve. Bottom: absolute error at the data points on a log scale.

2.3 (2c) Improving the LU decomposition

We now come to the part where we realize that LU decomposition methods for solving c can be problematic when we followed Court's algorithm. Namely, for source matrix $\mathbf{A}_{i,j}$, when we perform pivoting opeartions, we might run into cases where the values on each row are in extreme ratio. Since we enforce `dtype=numpy.float64` on matrix entries, the error from numerical opeataions increases as the number of operations increases. Therefore, by introducing partial pivoting with pivot value storage in a dedicated vector $\vec{p}$.

This step is important for organizing the source matrix before pivoting is performed. Consider the following matrix with arbitrarily massive value differences:

$$\mathbf{A}_{i,j} = \begin{bmatrix} 10^{2} & 10^{9} & 10^{21} \\ 10^{-20} & 10^{21} & 10^{-1} \\ 10^{3} & 10^{1} & 10^{-2} \end{bmatrix} \tag{20}$$

With the previous implementatin of Court's algorithm, we get pivot term 10^{-22} on row $i = 1$, This lead to iterative Gaussian elimination steps on the rows, negelecting potential numerical stability issue with repeated divition. Alternativly, by adding implicit pivoting, we observe that a max value of $(i, j) = (1, 1)$ is found in $\mathbf{A}_{i,j}$. We can therefore first compute the pivot for this $(i, j) = (1, 1)$ pair for Gaussian elimination. The core idea is to prevent division by extremely small numbers.

In our code, we loop over indecies to find the max value of some $\mathbf{A}_{i,j}$. After which, we rearange $\mathbf{A}_{i,j}$ through row swapping if applicable for Gaussian elimination preparation. Thus, we obtain $\mathbf{LU}_{i,j}$ with implicit pivoting included. And as usual, we now have the necessary ingredients to solve for c through forward and backward substitution. Recall the expression for $\mathbf{V}_{i,j}c = \vec{y}$, we can compute the residual r, which leads to the correction term δ_y and δ_c for $\vec{y}$ and c, since we aim to minimize r.

$$\begin{aligned} r &= \vec{y}_{\text{computed}} - \vec{y}_{\text{data}} \\ \delta_y &\Rightarrow \mathbf{LU}_{i,j}\vec{r} \\ \delta_c &\Rightarrow \mathbf{LU}_{i,j}\vec{\delta_y} \end{aligned} \tag{21}$$

Thus, we obtain the new coefficient c as $c + \delta_c$. We then iterate over to minimize the residual r. In the figures, r live within a smaller range at higher iterations. Howeverm it is observed that these errors do converge when a large number of data points x are given.

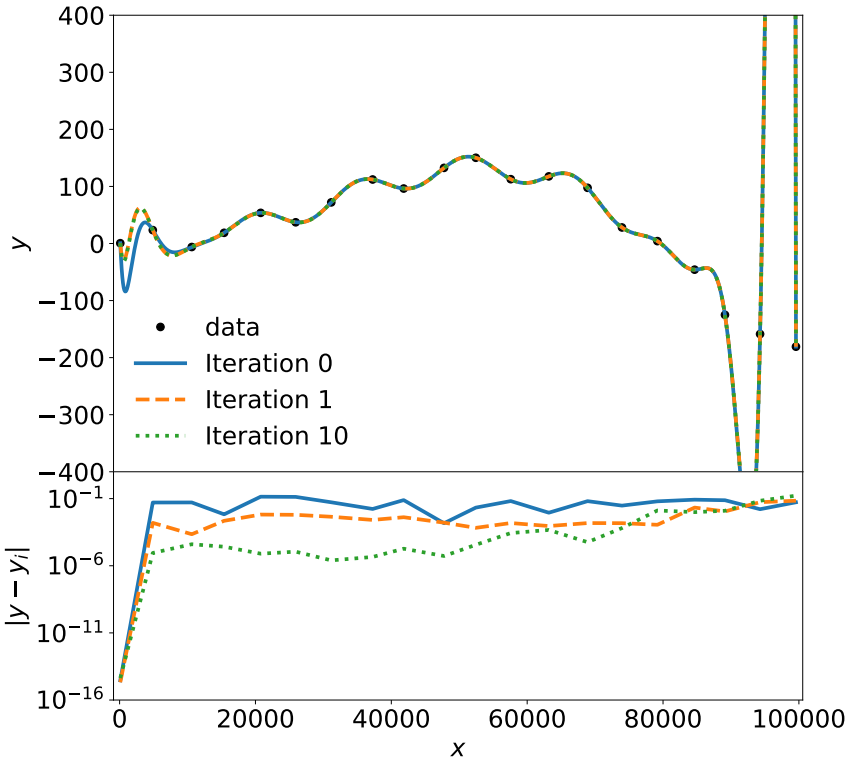

Figure 3: LU-based solution with iterative refinement (showing iterations 0, 1 and 10). Top: interpolated curves. Bottom: absolute error at the data points on a log scale.

2.4 (2d) Timing

The function shipped with the template indicated the time calculation from average time taken for ten function calls. We see that basic LU decomposition without implicit pivoting uses the least amount of time when generating results while Neville's algorithm implmentation takes significantly longer to complete the same coefficient calculations. LU decomposition with implicit pivoting seems to sit in between the two abovementioned methods in terms of time taken.

Timing results (average per run):

- Execution time for part (a): 0.00027 seconds
- Execution time for part (b): 0.08381 seconds
- Execution time for part (c): 0.00170 seconds

2.5 Code for Question 2

The following code was used for parts (2a)–(2d):

```
"""
Scripts for assignment 1 question 2
"""

import os
import sys
import timeit

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.rcParams["font.size"] = 20
mpl.rcParams["axes.labelsize"] = 20
mpl.rcParams["xtick.labelsize"] = 20
mpl.rcParams["ytick.labelsize"] = 20


def load_data():
    """
    Function to load the data from Vandermonde.txt.

    Returns
    ------------
    x (np.ndarray): Array of x data points.

    y (np.ndarray): Array of y data points.
    """
    data = np.genfromtxt(
        "./data/vandermonde.txt",
        comments="#",
        dtype=np.float64,
    )
    x = data[:, 0]
    y = data[:, 1]
    return x, y


def construct_vandermonde_matrix(x: np.ndarray) -> np.ndarray:
    """
    Construct the Vandermonde matrix V with V[i,j] = x[i]^j.

    Parameters
    ----------
    x : np.ndarray, x-values.

    Returns
    -------
    V : np.ndarray, Vandermonde matrix.
    """
    # init vandermonde mat at shape (len(x), len(x))
    # use consistent float64 dtype as load_data()
    n = len(x)
    v_mat = np.zeros((n, n), dtype=np.float64)

    # assign val at elements with V(i,j) as x_i^j, first col == 1 by def
    # 1st col -> 0 idx == [1, 1, ...] Transpose -> V(0,0)=1.0
    # 2nd col -> 1 idx == [x_0, x_1, ...] Transpose -> V(1,0)=1.0*x_0
```

```
    # 3rd col -> 2 idx == [x_0**2, x_1**2, ...] Transpose -> V(2,0)=(1.0*x_0)*x_0
    # efficient through col assignment by doing col_(j) = x**(j-1)
    # alternative -> col_(j) = x * col_(j-1)
    # with j=0 col assignment -> reducing n(operations) on each row with
    v_mat[:, 0] = np.float64(1.0)
    # now loop over the rest of the col
    for j in range(1, n):
        v_mat[:, j] = x * v_mat[:, j - 1]

    return v_mat


def LU_decomposition(A: np.ndarray) -> np.ndarray:
    """
    Perform LU decomposition.

    The lower-triangular matrix (L) is stored in the lower part of A (the diagonal elements are assumed =1),
    while the upper-triangular matrix (U) is stored on and above the diagonal of A.

    Parameters
    ----------
    A : np.ndarray
        Matrix to decompose.

    Returns
    -------
    A : np.ndarray
        Decomposed array.
    """
    n_row, n_col = A.shape
    # error if not square
    if n_row != n_col:
        raise ValueError(
            f"Abort with non-square matrix. Current shape ({A.shape[0]}, {A.shape[1]})."
        )
    # error if singular
    if n_row == n_col == 1:
        raise ValueError(
            f"Abort with a singular matrix. Current shape ({A.shape[0]}, {A.shape[1]})."
        )

    # do gaussian elimination with mat element A(i,j)
    # 1st row, 1st col item is A(1, 1) in mat, code at 0 idx equivalent to A[0, 0]
    # L -> terms with i > j
    # U -> terms with i <= j
    # LU -> combined L and U into one mat
    # LU through crout's -> pivot -> loop over col k -> then row i
    # last diag term does not need pivot
    for k in range(n_col - 1):  # k in [0, n_col-1)
        pivot = A[k, k]
        # break if zero pivot
        if pivot == np.float64(0.0):
            raise ValueError(f"Found zero pivot, pivot({k}, {k}) = {pivot}.")
        # do operation on whole row
        for i in range(k + 1, n_col):  # i in [1, n_col)
            # L(i,k) = A(i,k)/A(k,k), i>k
            A[i, k] /= pivot
            # update row -< reduce
```

```
            A[i, k + 1 :] -= A[i, k] * A[k, k + 1 :]

    return A

    # general form of A for LU follows
    # A = mat(A_ij)
    # if A x = b, with A = LU
    # LU x = b
    # we can sub with y = U x -> L y = b
    # we now have 2 sets of eqs 1) y = U x, 2) L y = b
    # we can now solve for y and x


def forward_substitution_unit_lower(LU: np.ndarray, b: np.ndarray) -> np.ndarray:
    """
    Solve L*y = b using forward substitution,
    where L is the lower-triangular matrix.

    Parameters
    ----------
    LU : np.ndarray
        LU matrix from LU_decomposition.
    b : np.ndarray
        Right-hand side.

    Returns
    -------
    y : np.ndarray
        Solution vector.
    """
    # forward sub
    # lecture 3 p11
    # L * y = b
    # y has shape agreement with b
    y = np.zeros(len(b), dtype=np.float64)

    # y_i = (1 / L_ii) * (b_i - sum_{j=0}^{i-1}(y_j * L_ij))
    # because of LU mat instead of L and U mat, we can safely set L_ii = 1
    #
    # y[0] = b[0]
    # y[1] = b[1] - (LU[1,0] * y[0])
    # y[2] = b[2] - (LU[2,0] * y[0] + LU[2,1] * y[1])
    for i in range(len(y)):
        the_sum = np.float64(0.0)
        for j in range(i):
            the_sum += LU[i, j] * y[j]
        y[i] = b[i] - the_sum

    return y


def backward_substitution_upper(LU: np.ndarray, y: np.ndarray) -> np.ndarray:
    """
    Solve U*c = y using backward substitution,
    where U is the upper-triangular matrix.

    Parameters
    ----------
    LU : np.ndarray
        LU matrix from LU_decomposition.
    y : np.ndarray
        Right-hand side.
```

```
179
180     Returns
181     -------
182     c : np.ndarray
183         Solution vector.
184     """
185     # backward sub
186     # lecture 3 p11 -> U*x = y -> use c notations here
187     # make results ary like in forward sub
188     c = np.zeros(len(y), dtype=np.float64)
189
190     # c_i = (1/U_ii)*(y_i-sum_{j=i+1}^{n-1}(U_ij*c_j))
191     # for j in [i+1, n-1) -> n terms -> idx n-1 = idx -1 -> idx j as [i+1, -1, -1]
192     #
193     # just like in forward sub, we use a LU mat instead of a separate U
194     # U_ij = LU_ij since j >= i
195     #
196     # set n=3
197     # c[0] = (1/LU[0,0])*(y[0]-(LU[0,2]*c[2] + LU[0,1]*c[1]))
198     # c[1] = (1/LU[1,1])*(y[1]-(LU[1,2]*c[2]))
199     # c[2] = (1/LU[2,2])*(y[2])
200     #
201     # loop from last i, i = n-1 = 2 -> loop 2, 1, 0 -> range(n-1, -1, -1)
202     # for j val
203     # when at i = 2 -> empty j loop
204     # when at i = 1 -> loop j at 2
205     # when at i = 0 -> loop j at 1, 2
206     for i in range(len(c) - 1, -1, -1):
207         the_sum = np.float64(0.0)
208         for j in range(i + 1, len(c)):
209             the_sum += LU[i, j] * c[j]
210         c[i] = (1 / LU[i, i]) * (y[i] - the_sum)
211     return c
212
213
214 def vandermonde_solve_coefficients(x: np.ndarray, y: np.ndarray) -> np.ndarray:
215     """
216     Solve for polynomial coefficients c from data (x,y) using the Vandermonde
        matrix.
217
218     Parameters
219     ----------
220     x : np.ndarray
221         x-values.
222     y : np.ndarray
223         y-values.
224
225     Returns
226     -------
227     c : np.ndarray
228         Polynomial coefficients.
229     """
230     # get vandermonde mat
231     v_mat = construct_vandermonde_matrix(x)
232     # get LU from v mat
233     LU = LU_decomposition(v_mat)
234     # we want c
235     # first solve forward sub -> Ly=b -> returns y
236     # y ary in arg is actually b
237     y_ary = forward_substitution_unit_lower(LU, y)
238     # then solve backward sub -> Uc=y_ary -> returns c
239     c = backward_substitution_upper(LU, y_ary)
```

```
240
241     return c
242
243
244 def evaluate_polynomial(c: np.ndarray, x_eval: np.ndarray) -> np.ndarray:
245     """
246     Evaluate y(x) = sum_j c[j] * x^j.
247
248     Parameters
249     ----------
250     c : np.ndarray
251         Polynomial coefficients.
252     x_eval : np.ndarray
253         Evaluation points.
254
255     Returns
256     -------
257     y_eval : np.ndarray
258         Polynomial values.
259     """
260     # we want y_eval = sum_j (c[j] * x_eval**j)
261     # y_eval[i] = sum_j (c[j] * x_eval[i]**j)
262     # x_eval.shape = y_eval.shape
263     y_eval = np.zeros(x_eval.shape, dtype=np.float64)
264
265     # we need to loop j for c.shape times at each i -> len(c)
266     # at i = 0, say we set len(c) = 3
267     # y[0] = c[0] * 1 + c[1] * x[0] + c[2] * x[0]**2
268     # y[1] = c[0] * 1 + c[1] * x[1] + c[2] * x[1]**2
269     #
270     # y[i] = c[0] * 1 + c[1] * 1 * x[j] + c[2] * 1 * x[j] * x[j]
271     # the prod behind each c can be computed by
272     # prod *= prod, with init at 1.0
273     for i in range(len(x_eval)):
274         y_val = np.float64(0.0)
275         x_val = np.float64(1.0)
276         for j in range(len(c)):
277             y_val += c[j] * x_val
278             x_val *= x_eval[i]
279         y_eval[i] = y_val
280
281     return y_eval
282
283
284 def neville(x: np.ndarray, y: np.ndarray, k: float) -> float:
285     """
286     Function that applies Nevilles algorithm to calculate the function value at k.
287
288     Parameters
289     ------------
290     x (np.ndarray): Array of x data points.
291     y (np.ndarray): Array of y data points.
292     k (float): The x value at which to interpolate.
293
294     Returns
295     ------------
296     float: The interpolated y value at k.
297     """
298     # lecture 4 p12 -> consider romberg
299     # x_data.shape should match y_data.shape -> same len()
300     # need polynomial table p -> upper trig form
301     n = len(x)
```

```
    p = np.zeros((n, n), dtype=np.float64)

    # consult gen form when 0<=i<=j<=n
    # p[i,i] = y[i]
    # p[i,j] eval at x = ((x-x[i])p[i+1,j]-(x-x[j])p[i,j-1])/(x[j]-x[i])
    # for code implementation, loop over diag is complicated
    #
    # p00 p01 p02     p00 p01 p02
    #     p11 p12  => p10 p11
    #         p22     p20
    #
    # loop over col at each iter
    # at each col, loop over rows
    # p00 = y0, p10 = y1, p20 = y2
    # p01 = y01, p11 = y12
    # p02 = y012
    #
    # at niter0, col0 is y
    p[:, 0] = y
    # loop over col to get the p[:j] needed for next col
    for j in range(1, n):
        for i in range(n - j):
            top = (k - x[i]) * p[i + 1, j - 1] - (k - x[i + j]) * p[i, j - 1]
            bot = x[i + j] - x[i]
            p[i, j] = top / bot

    result = np.float64(p[0, -1])

    return result


# you can merge the function below with LU_decomposition to make it more efficient
def run_LU_iterations(
    x: np.ndarray,
    y: np.ndarray,
    iterations: int = 11,
    coeffs_output_path: str = "./output/a1q2_coefficients_output.txt",
):
    """
    Iteratively improves computation of coefficients c.

    Parameters
    ----------
    x : np.ndarray
        x-values.
    y : np.ndarray
        y-values.
    iterations : int
        Number of iterations.
    coeffs_output_path : str
        File to write coefficient values per iteration.

    Returns
    -------
    coeffs_history :
        List of coefficient vectors.
    """
    # get initial conditions
    n = len(x)
    # get base v_mat
    v_mat = construct_vandermonde_matrix(x)
    # create LU decomposition pivot vec p
```

```
    LU = v_mat.copy()
    p = np.arange(n)
    # init history
    coeffs_history = []

    # find max pivot and record row idx
    for k in range(n - 1):

        # get initial max val
        pivot_max = abs(LU[k, k])
        pivot_max_row = k

        # check over other pivots
        for i in range(k + 1, n):
            pivot_val = abs(LU[i, k])
            if pivot_val > pivot_max:
                pivot_max = pivot_val
                pivot_max_row = i

        # row swap if max pivot not on 0,0
        if pivot_max_row != k:
            LU[[k, pivot_max_row]] = LU[[pivot_max_row, k]]
            p[[k, pivot_max_row]] = p[[pivot_max_row, k]]

        # do LU decomp
        pivot = LU[k, k]
        for i in range(k + 1, n):
            LU[i, k] /= pivot
            LU[i, k + 1 :] -= LU[i, k] * LU[k, k + 1 :]

    # y swap according to pivot vec p
    y_new = y[p]

    # find initial c
    y0 = forward_substitution_unit_lower(LU, y_new)
    c = backward_substitution_upper(LU, y0)
    coeffs_history.append(c.copy())

    # write to file
    # from niter1 onwards
    # get residual res = y - y_estimated = y - v_mat * c
    # delta_c = vandermonde solve(x, dy), dy ~ res
    # c += delta_c -> add to history
    # next niter
    with open(coeffs_output_path, "w", encoding="utf-8") as f:
        f.write(f"niter=0\n")
        for i, coef in enumerate(c):
            f.write(f"c_{i}={coef:.3e}\n")

        for it in range(1, iterations):
            y_est = np.zeros(n, dtype=np.float64)
            for i in range(n):
                for j in range(n):
                    y_est[i] += v_mat[i, j] * c[j]

            res = y - y_est
            res_new = res[p]
            delta_y = forward_substitution_unit_lower(LU, res_new)
            delta_c = backward_substitution_upper(LU, delta_y)

            c += delta_c
            coeffs_history.append(c.copy())
```

```
426
427             f.write(f"niter={it}\n")
428             for i, coef in enumerate(c):
429                 f.write(f"c_{i}={coef:.3e}\n")
430
431     return coeffs_history
432
433
434 def plot_part_a(
435     x_data: np.ndarray,
436     y_data: np.ndarray,
437     coeffs_c: np.ndarray,
438     plots_dir: str = "./plots",
439 ) -> None:
440     """
441     Ploting routine for part (a) results.
442
443     Parameters
444     ----------
445     x_data : np.ndarray
446         x-values.
447     y_data : np.ndarray
448         y-values.
449     coeffs_c : np.ndarray
450         Polynomial coefficients c.
451     plots_dir : str
452         Directory to save plots.
453
454     Returns
455     -------
456     None
457     """
458     xx = np.linspace(x_data[0], x_data[-1], 1001)
459     yy = evaluate_polynomial(coeffs_c, xx)
460     y_at_data = evaluate_polynomial(coeffs_c, x_data)
461
462     fig = plt.figure(figsize=(10, 10))
463     gs = fig.add_gridspec(2, hspace=0, height_ratios=[2.0, 1.0])
464     axs = gs.subplots(sharex=True, sharey=False)
465
466     axs[0].plot(x_data, y_data, marker="o", linewidth=0)
467     axs[0].plot(xx, yy, linewidth=3)
468     axs[0].set_xlim(
469         np.floor(xx[0]) - 0.01 * (xx[-1] - xx[0]),
470         np.ceil(xx[-1]) + 0.01 * (xx[-1] - xx[0]),
471     )
472     axs[0].set_ylim(-400, 400)
473     axs[0].set_ylabel("$y$")
474     axs[0].legend(["data", "Via LU decomposition"], frameon=False, loc="lower left
        ")
475
476     axs[1].set_ylim(1e-16, 1e1)
477     axs[1].set_yscale("log")
478     axs[1].set_ylabel(r"$|y - y_i|$")
479     axs[1].set_xlabel("$x$")
480     axs[1].plot(x_data, np.abs(y_data - y_at_data), linewidth=3)
481
482     plt.savefig(os.path.join(plots_dir, "a1q2_vandermonde_sol_2a.pdf"))
483     plt.close()
484
485
486 def plot_part_b(
```

```
    x_data: np.ndarray,
    y_data: np.ndarray,
    plots_dir: str = "./plots",
) -> None:
    """
    Ploting routine for part (b) results.

    Parameters
    ----------
    x_data : np.ndarray
        x-values.
    y_data : np.ndarray
        y-values.
    plots_dir : str
        Directory to save plots.

    Returns
    -------
    None
    """
    xx = np.linspace(x_data[0], x_data[-1], 1001)
    yy = np.array([neville(x_data, y_data, x) for x in xx], dtype=np.float64)
    y_at_data = np.array([neville(x_data, y_data, x) for x in x_data], dtype=np.
        float64)

    fig = plt.figure(figsize=(10, 10))
    gs = fig.add_gridspec(2, hspace=0, height_ratios=[2.0, 1.0])
    axs = gs.subplots(sharex=True, sharey=False)

    axs[0].plot(x_data, y_data, marker="o", linewidth=0)
    axs[0].plot(xx, yy, linestyle="dashed", linewidth=3)
    axs[0].set_xlim(
        np.floor(xx[0]) - 0.01 * (xx[-1] - xx[0]),
        np.ceil(xx[-1]) + 0.01 * (xx[-1] - xx[0]),
    )
    axs[0].set_ylim(-400, 400)
    axs[0].set_ylabel("$y$")
    axs[0].legend(["data", "Via Neville's algorithm"], frameon=False, loc="lower
        left")

    axs[1].set_ylim(1e-16, 1e1)
    axs[1].set_yscale("log")
    axs[1].set_ylabel(r"$|y - y_i|$")
    axs[1].set_xlabel("$x$")
    axs[1].plot(x_data, np.abs(y_data - y_at_data), linestyle="dashed", linewidth
        =3)

    plt.savefig(os.path.join(plots_dir, "a1q2_vandermonde_sol_2b.pdf"))
    plt.close()


def plot_part_c(
    x_data: np.ndarray,
    y_data: np.ndarray,
    coeffs_history: list[np.ndarray],
    iterations_num: list[int] = [0, 1, 10],
    plots_dir: str = "./plots",
) -> None:
    """
    Ploting routine for part (c) results.

    Parameters
```

```
    ----------
    x_data : np.ndarray
        x-values.
    y_data : np.ndarray
        y-values.
    coeffs_history : list[np.ndarray]
        Coefficients per iteration.
    iterations_num : list[int]
        Iteration numbers to plot.
    plots_dir : str
        Directory to save plots.

    Returns
    -------
    None
    """

    linstyl = ["solid", "dashed", "dotted"]
    colors = ["tab:blue", "tab:orange", "tab:green"]

    xx = np.linspace(x_data[0], x_data[-1], 1001)

    fig = plt.figure(figsize=(10, 10))
    gs = fig.add_gridspec(2, hspace=0, height_ratios=[2.0, 1.0])
    axs = gs.subplots(sharex=True, sharey=False)

    axs[0].plot(x_data, y_data, marker="o", linewidth=0, color="black", label="
        data")

    for i, k in enumerate(iterations_num):
        if k >= len(coeffs_history):
            continue
        c = coeffs_history[k]
        yy = evaluate_polynomial(c, xx)
        y_at_data = evaluate_polynomial(c, x_data)
        diff = np.abs(y_at_data - y_data)

        axs[0].plot(
            xx,
            yy,
            linestyle=linstyl[i],
            color=colors[i],
            linewidth=3,
            label=f"Iteration {k}",
        )
        axs[1].plot(x_data, diff, linestyle=linstyl[i], color=colors[i], linewidth
            =3)

    axs[0].set_xlim(
        np.floor(xx[0]) - 0.01 * (xx[-1] - xx[0]),
        np.ceil(xx[-1]) + 0.01 * (xx[-1] - xx[0]),
    )
    axs[0].set_ylim(-400, 400)
    axs[0].set_ylabel("$y$")
    axs[0].legend(frameon=False, loc="lower left")

    axs[1].set_ylim(1e-16, 1e1)
    axs[1].set_yscale("log")
    axs[1].set_ylabel(r"$|y - y_i|$")
    axs[1].set_xlabel("$x$")

    plt.savefig(os.path.join(plots_dir, "a1q2_vandermonde_sol_2c.pdf"))
```

```
606     plt.close()
607
608
609 def main():
610     os.makedirs("./plots", exist_ok=True)
611     x_data, y_data = load_data()
612
613     # compute times
614     number = 10
615
616     t_a = (
617         timeit.timeit(
618             stmt=lambda: vandermonde_solve_coefficients(x_data, y_data),
619             number=number,
620         )
621         / number
622     )
623
624     xx = np.linspace(x_data[0], x_data[-1], 1001)
625     t_b = (
626         timeit.timeit(
627             stmt=lambda: np.array(
628                 [neville(x_data, y_data, x) for x in xx], dtype=np.float64
629             ),
630             number=number,
631         )
632         / number
633     )
634
635     t_c = (
636         timeit.timeit(
637             stmt=lambda: run_LU_iterations(x_data, y_data, iterations=11),
638             number=number,
639         )
640         / number
641     )
642
643     # write all timing
644     with open("./output/a1q2_execution_times.txt", "w", encoding="utf-8") as f:
645         f.write(f"\\item Execution time for part (a): {t_a:.5f} seconds\n")
646         f.write(f"\\item Execution time for part (b): {t_b:.5f} seconds\n")
647         f.write(f"\\item Execution time for part (c): {t_c:.5f} seconds\n")
648
649     c_a = vandermonde_solve_coefficients(x_data, y_data)
650     plot_part_a(x_data, y_data, c_a)
651
652     formatted_c = [f"{coef:.3e}" for coef in c_a]
653     with open("./output/a1q2_coefficients_output.txt", "w", encoding="utf-8") as f
        :
654         for i, coef in enumerate(formatted_c):
655             f.write(f"c$_{i+1}$ = {coef}, ")
656
657     plot_part_b(x_data, y_data)
658
659     coeffs_history = run_LU_iterations(
660         x_data,
661         y_data,
662         iterations=11,
663         coeffs_output_path="./output/a1q2_coefficients_per_iteration.txt",
664     )
665     plot_part_c(x_data, y_data, coeffs_history, iterations_num=[0, 1, 10])
666
```

```
667
668 if __name__ == "__main__":
669     main()
```

2.6 Conclusions

That is, we observe basic LU decomposition through Court's algorithm to be the fasted but least accurate. This is because we are gambling on the source matrix to be somewhat compatible for this direct approach without introducing too much numerical instabilities. And Neville's algorithm is the slowest by almost 300 orders of magnitude in terms of time cosumed while producing the smallest error. This is because Neville's algorithm interpolate the Vandermonde matrix for results in an almost bruteforce way. Even with a permutation table present, the number of operations needed to fill the permutation table is huge. Comparing the order of operations. Basic LU decomposition is the heaviest when LU matrix operations are called, as Gaussian elimination requires opeations $O(n^3)$. While Neville's algorithm, though relying on permutation table, which echoes to order of $O(n^2)$, is linearly dependent on the number of points supplied.

Iterative LU decomposition with implicit pivoting gives reasonably small error terms while being significantly faster than Neville's algorithm. Foundamentally, the iterative approach is very similar to the basic LU decomposition solving through Court's algorithm, with key difference in using implicit pivoting for maintaining numerical stability. By direct comparison, the iterative process takes on more operations, but after the first iteration where initial c is known, the iterative process only need to solve for δ_y and δ_c, instead of constructing a fresh source data matrix at each iteration.

Overall, we must consider how computers handle numbers. For this assigmnet, most arrays and numbers are computed with `numpy.float64` precision. And with each operations, rounding errors compound. Assume high iterations for solving a system, higher number of opearations reflects a higher error in numerical roundoff. But in terms of balancing speed and precision, an iterative approach is ideal.

Acknowledgement

This article is rendered off of source files at git@github.com:Yang-Taotao/strw-nur-scripts.