NUR Hand-in Exercise 1

Angèl Pranger

February 25, 2025

Abstract

This document shows my solutions to hand-in exercise 1 of numerical methods in astrophysics.

1 Poisson distribution

The script for question 1 is given by:

```python
#!/usr/bin/env python

import numpy as np

# overflow caused by the factorial in the traditional way for large k

# using log conversion
def poisson1(lamda:np.float32, k:np.float32) -> np.float32:
    j = np.float32(0)
    for i in range(np.int32(k)):
        i = np.float32(i)
        j += np.float32(np.log(k-i))
    P = np.float32(np.exp(k*np.log(lamda)-lamda-j))
    return P

# using different order of operations
def poisson2(lamda:np.float32, k:np.float32) -> np.float32:
    P = np.float32(np.exp(-lamda))
    for i in range(np.int32(k)):
        i = np.float32(i)
        P *= np.float32(lamda/(k-i))
    return P

# using different order of operations
def poisson3(lamda:np.float32, k:np.float32) -> np.float32:
    P = np.float32(1)
    for i in range(np.int32(k)):
        i = np.float32(i)
        P *= np.float32(lamda/(k-i))
    P = np.float32(np.exp(np.log(P) - lamda))
    return P

lamda_k = np.array([[1,0],[5,10],[3,21],[2.6,40],[100,5],[101,200]], dtype=np.float32)
print("[lamda k] P1 P2 P3")
for values in lamda_k:
    P1 = poisson1(values[0], values[1])
    P2 = poisson2(values[0], values[1])
    P3 = poisson3(values[0], values[1])
    print(values, P1, P2, P3)

# TODO: why do the results for the last two values differ? seems like the exponent gives
    a large round off error
# version 3 is the most accurate according to wolframalpha, but why? Im not sure
```

1_poisson_distribution.py

The results are given by:

```
[lamda k] P1 P2 P3
[1. 0.] 0.36787942 0.36787942 0.36787942
[ 5. 10.] 0.018132769 0.018132787 0.018132787
[ 3. 21.] 1.0193364e-11 1.0193401e-11 1.01934025e-11
[ 2.6 40. ] 3.6150753e-33 3.615119e-33 3.6151308e-33
[100. 5.] 3.1000582e-36 3.1529214e-36 3.1000582e-36
[101. 200.] 1.269529e-18 1.0743388e-05 1.2695337e-18
```

poisson_distribution.txt

2 Vandermonde matrix

The code of any shared modules for question 2 is given by:

```python
#!/usr/bin/env python

import numpy as np
import timeit
import sys
import os
import matplotlib.pyplot as plt

######################################################################

# Importing data

data=np.genfromtxt(os.path.join(sys.path[0],"Vandermonde.txt"),comments='#',dtype=np.
    float64)
x=data[:,0]
y=data[:,1]
xx=np.linspace(x[0],x[-1],1001) # x values to interpolate at

######################################################################

# Functions used

def LU_decomposition(matrix:np.ndarray):
    """TODO"""
    # Assume alpha_ii = 1
    # Loop over the columns j
    for j in range(matrix.shape[0]):
        # Keep beta_0j = a_0j
        # Update beta_ij where 0<i<=j
        for i in range(1, j+1):
            term = np.float64(0)
            for k in range(0, i):
                term += matrix[i,k]*matrix[k,j]
            matrix[i,j] -= term
        # Update alpha_ij where i>j
        beta_jj = 1/matrix[j,j]
        for i in range(j+1, matrix.shape[0]):
            term = np.float64(0)
            for k in range(0, j):
                term += matrix[i,k]*matrix[k,j]
            matrix[i,j] = (matrix[i,j]-term)*beta_jj
    return np.float64(matrix)

def solving_system_with_LU(LU:np.ndarray, b:np.array) -> np.array:
    """TODO"""
    # Knowing that alpha_ii = 1
    # Forward substitution
    # Keep y_0 = b_0 as alpha_00 = 1
    for i in range(1, LU.shape[0]):
        term = np.float64(0)
        for j in range(i):
            term += LU[i,j]*b[j]
        b[i] -= term
```

```
    # Backward substitution
    b[-1] /= LU[-1,-1]
    for i in range(LU.shape[0]-1):
        i = LU.shape[0]-1 - i
        term = np.float64(0)
        for j in range(i+1, LU.shape[0]):
            term += LU[i,j]*b[j]
        b[i] = (b[i]-term)/LU[i,i]
    return np.float64(b)

def bisection(M:np.int64, x_samp:np.array, x_interp:np.array) -> np.ndarray:
    """TODO"""
    # Assumes x_samp are uniformly spaced
    if (M > x_samp.shape[0]):
        print("M is too large for the array x_samp")
        return
    for (i,x) in enumerate(x_interp):
        l_idx = np.int64(0)
        r_idx = np.int64(x_samp.shape[0]-1)
        while (r_idx - l_idx > 1):
            m_idx = np.int64(np.floor((l_idx + r_idx)*0.5))
            if (x <= x_samp[m_idx]):
                r_idx = m_idx
            elif (x > x_samp[m_idx]):
                l_idx = m_idx
        m = (x_samp[l_idx] + x_samp[r_idx])*0.5
        # Computing the left boundary index (ignoring range)
        l_idx -= np.int64(np.ceil(0.5*M-1))
        # Correcting left boundary if x is closer to the right boundary
        # (in which case the right boundary is seen as the first point instead of the second)
        if ((x > m) & (M % 2 == 1)):
            l_idx += np.float64(1)
        # Checking range
        if l_idx < 0:
            l_idx = 0
        if (l_idx + M >= x_samp.shape[0]):
            l_idx -= l_idx + M - x_samp.shape[0]
        x_interp[i] = l_idx
    return np.int64(x_interp)

def neville(M:np.int64, x_samp:np.array, y_samp:np.array, x_interp:np.array) -> np.ndarray:
    """TODO"""
    if (M < 1):
        print("M is too small")
        return
    # Finding left index of M tabulated points using bisection
    l_indices = bisection(M, x_samp.copy(), x_interp.copy()) # correct!
    # Initializing matrix P
    Ps = np.zeros((x_interp.shape[0], M), dtype=np.float64)
    for i in range(M):
        Ps[:,i] = y_samp[l_indices+i] # correct!
    # Looping over orders k and current intervals
    for k in range(1, M):
        for i in range(M-k):
            j = i+k # note this j is only for the x_samp, as Ps is overwritten so just j=i+1 as index
            Ps[:,i] = ((x_samp[l_indices+j]-x_interp)*Ps[:,i]+(x_interp-x_samp[l_indices+i])*Ps[:,i+1])/(x_samp[l_indices+j]-x_samp[l_indices+i])
    if (M < 2):
        return Ps[:,0], 0
    error_estimate = np.absolute(Ps[:,0]-Ps[:,1])
    return Ps[:,0], error_estimate

def polynomial(c:np.array, x:np.array) -> np.array:
    """Constructing the values y of the polynomial with coefficients given by c at points x.
    Returns y values."""
    y = np.float64(0)
```

```
    for j in range(c.shape[0]):
        y += c[j] * np.power(x,j)
    return np.float64(y)
```

2_vandermonde_matrix.py

2.1 a

TODO(For all routines you write,explainhow they work in the comments of your code andargueyour-choices) TODO(This includes discussing your plots in their caption) TODO(Per main question, the code of any shared modules. Per sub-question, an explanation of what you did. Per sub-question, the code specific to it. Per sub-question, the output(s) along with discussion/captions) TODO(Add plotting code.)

```

def q2a(x:np.array, y:np.array, xx:np.array) -> np.ndarray:
    """Constructing Vandermonde matrix V from data x, LU-decomposing V and solving Vc=y for c.
    Returns c and the values of the polynomial, yya and ya, corresponding to xx and x.
    """
    # Constructing V
    V = np.zeros((x.shape[0], x.shape[0]), dtype=np.float64)
    for j in range(V.shape[0]):
        V[:,j] = np.power(x,j)
    # LU-decomposing V
    LU_V = LU_decomposition(V.copy())
    # Solving for c
    c0 = solving_system_with_LU(LU_V, y.copy())
    # Finding the polynomial values y
    yya = polynomial(c0, xx)
    ya = polynomial(c0, x)
    return V, LU_V, c0, yya, ya

V, LU_V, c0, yya, ya = q2a(x, y, xx)

# Printing the values of c
print("The coefficients of the polynomial found with LU-decomposition are")
print(c0)
```

2_vandermonde_matrix.py

```
The coefficients of the polynomial found with LU-decomposition are
[ 5.21896930e-01  3.53817596e+03 -2.20652879e+03  6.00128574e+02
 -9.43549171e+01  9.66286617e+00 -6.87646085e-01  3.53596337e-02
 -1.34619705e-03  3.84953383e-05 -8.32127939e-07  1.35821060e-08
 -1.65786631e-10  1.48156916e-12 -9.31152862e-15  3.80148410e-17
 -8.28186823e-20  2.55562475e-23  1.78743492e-25 -1.28961270e-56]
```

vandermonde_matrix.txt

2.2 b

```
# Plot of points with absolute difference shown on a log scale (question 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)
```

2_vandermonde_matrix.py

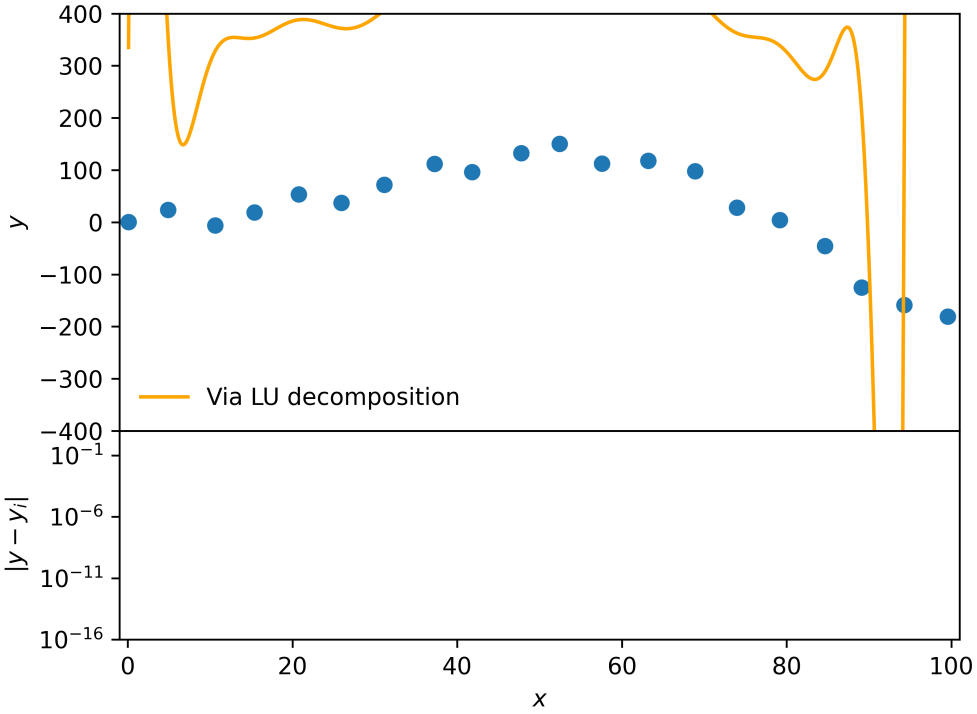

Figure 1:

2.3 c

```
159 axs[1].set_yscale('log')
160 line,=axs[0].plot(xx,yya,color='orange')
161 line.set_label('Via LU decomposition')
162 axs[0].legend(frameon=False,loc="lower left")
163 axs[1].plot(x,abs(y-ya),color='orange')
164 plt.savefig('my_vandermonde_sol_2a.png',dpi=600)
165
166 ######################################################################
167
168 # 2b
169
170 def q2b(x:np.array, y:np.array, xx:np.array, M:np.int64):
171     """Computing the lagrange polynomial through the given data points (x,y) using
    Neville's algorithm (iff M=x.shape[0]).
172     Returns values yyb and yb of the polynomial corresponding to the points in xx and x
    respectively."""
173     yyb = neville(M, x, y, xx)[0]
174     yb = neville(M, x, y, x)[0]
```

2_vandermonde_matrix.py

2.4 d

```
177 yyb, yb = q2b(x, y, xx, 20)
178
179 # For questions 2b and 2c, add this block
180 line,=axs[0].plot(xx,yyb,linestyle='dashed',color='green')
181 line.set_label('Via Neville\'s algorithm')
182 axs[0].legend(frameon=False,loc="lower left")
```

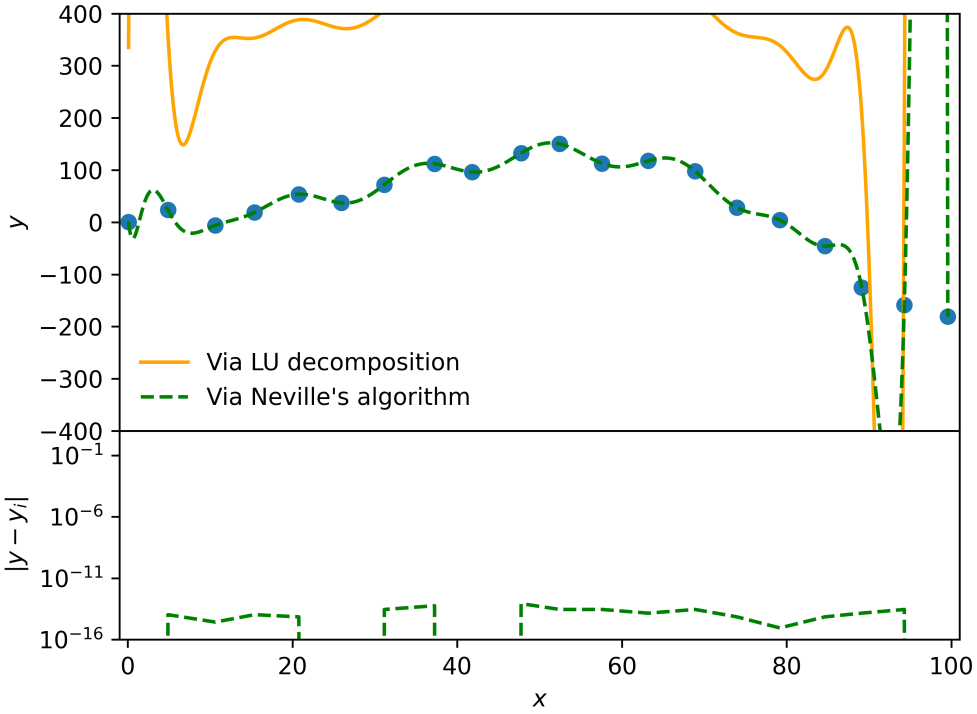

Figure 2:

```
183 axs[1].plot(x,abs(y-yb),linestyle='dashed',color='green')
```

2_vandermonde_matrix.py

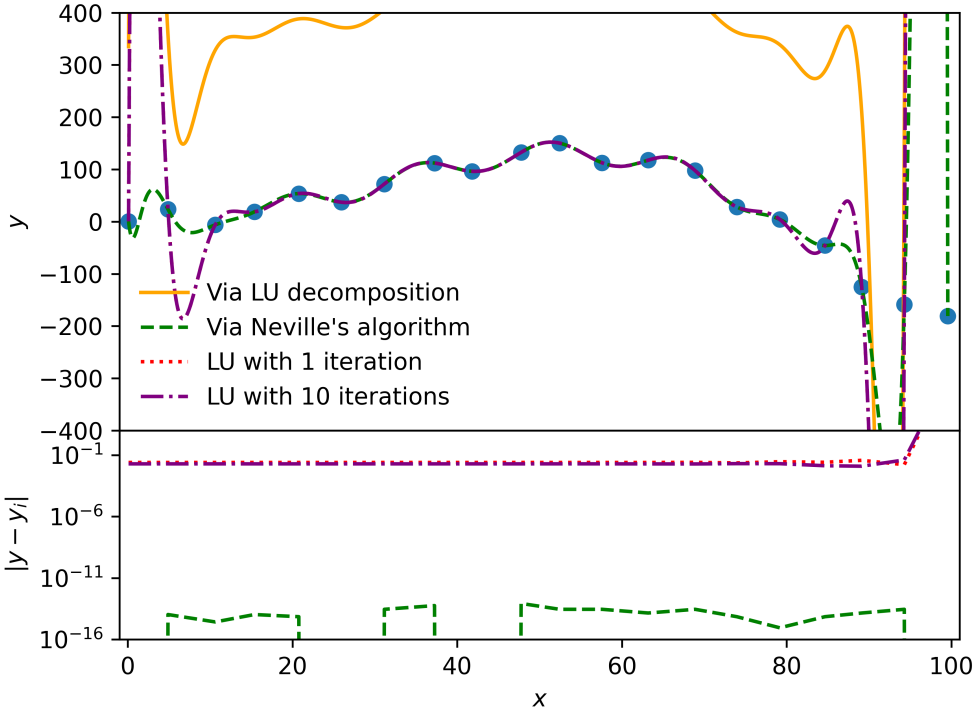
400
300
200
100
0
−100
−200
−300
−400
y
Via LU decomposition
Via Neville's algorithm
LU with 1 iteration
LU with 10 iterations
10^{-1}
10^{-6}
10^{-11}
10^{-16}
$|y - y_i|$
0
20
40
60
80
100
x

Figure 3: