NUR Hand-in 1

Berend Nieuwhof

February 26, 2025

Abstract

In this document, the solutions to the first hand-in exercise are presented.

1 Exercise 1: Poisson distribution

In this section, the solutions to the first exercise are presented.

Our script is given by:

```python
'''This file implements a function that returns the poisson distribution for an integer k.'''

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import poisson

def log_factorial(k):
    '''Computes the factorial of a number in log space.'''
    if k == 0:
        result = 0
    else:
        k_values = np.arange(k, 0, -1)
        result = np.sum(np.log(k_values))
    return np.float32(result)

def log_poisson(lamb, k):
    '''Computes the log of the poisson distribution for a positive mean lambda.'''
    assert lamb > 0
    k = np.int32(k)

    result = k * np.log(lamb) - lamb - log_factorial(k)
    return np.float32(result)

def poisson(lamb, k):
    return np.float32(np.exp(log_poisson(lamb, k)))

lamb_k_values = np.array([np.array([1, 5, 3, 2.6, 100, 101], dtype=np.float32),
                          np.array([0, 10, 21, 40, 5, 200], dtype=np.int32)])

np.set_printoptions(precision=8)

for i in range(len(lamb_k_values[0])):
    lamb = lamb_k_values[0, i]
    k = lamb_k_values[1, i]
    poisson_val = poisson(lamb, k)
    print(f'for lamb = {lamb:2f} and k = {k:1f}, P_lambda(k) = {poisson_val}'.format())
```

Poisson.py

As apparent from the code above, we have adopted a computation in log space. This avoids possible overflow errors and enables us to compute the Poisson probability for larger (λ, k) pairs. The output for a representative sample of pairs is given below, which demonstrates that the returned Poisson probability is accurate when compared to calculations done in online calculators.

```
for lamb = 1.000000 and k = 0.000000, P_lambda(k) = 0.3678794205188751
for lamb = 5.000000 and k = 10.000000, P_lambda(k) = 0.018132776021957397
for lamb = 3.000000 and k = 21.000000, P_lambda(k) = 1.019340254565515e-11
for lamb = 2.600000 and k = 40.000000, P_lambda(k) = 3.61510320423337e-33
for lamb = 100.000000 and k = 5.000000, P_lambda(k) = 3.1000581991562844e-36
for lamb = 101.000000 and k = 200.000000, P_lambda(k) = 1.2695239237170379e-18
```

poisson_output.txt

2 Exercise 2: Vandermonde Matrix

In this section we look at exercise 2, which is about Vandermonde matrices. The full script is presented below, and the parts of it specific to subquestions will be discussed in appropriate subsections.

Our script is given by:

```
# Main code body and plotting code adapted from Marcel van Daalen, at
# https://home.strw.leidenuniv.nl/~daalen/Handin_files/vandermonde.py

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

def crout(A):
    m, n = A.shape
    for i in range(m):
        for j in range(n):
            if i >= j:
                A[i, j] = A[i, j] - np.sum(A[i, :j]*A[:j, j])
            else:
                A[i, j] = (A[i, j] - np.sum(A[i, :i]*A[:i, j]))/A[i, i]

    return A

def forward_sub2(A, b):
    m, _ = A.shape
    y = copy.deepcopy(b)
    for k in range(m):
        y[k] = (b[k] - np.sum(A[k, :k]*y[:k]))/A[k, k]
    return y

def backward_sub2(A, b):
    _, n = A.shape
    x = np.zeros_like(b)
    for i in range(n-1, -1, -1):
        x[i] = b[i] - np.dot(A[i, i+1:], x[i+1:])
    return x

def get_coeffs(y, A):
    A_decomp = crout(A)
    fw_sub = forward_sub2(A_decomp, y)
    coeffs = backward_sub2(A_decomp, fw_sub)

    return coeffs, A_decomp

def polynomial(x, coeffs):
    y = np.zeros_like(x)
    for i in range(len(x)):
        for j in range(len(coeffs)):
            y[i] += coeffs[j] * x[i]**j
    return y

# Let's get started
data=np.genfromtxt(os.path.join(sys.path[0],"Vandermonde.txt"),comments='#',dtype=np.
    float64)
```

```
x=data[:,0]
y=data[:,1]

# Construct the matrix.
A = np.zeros([20, 20])
m, n = A.shape
for i in range(m):
    for j in range(n):
        A[i, j] = x[i]**j

A_orig = copy.deepcopy(A)
# now set up the polynomial function!

coeffs, crout_LU = get_coeffs(y, A)
xx=np.linspace(x[0],x[-1],1001) #x values to interpolate at
yya = polynomial(xx, coeffs)
ya= polynomial(x, coeffs)

# Now we'd like to implement neville's algorithm.

def neville(datax, datay, x_interp):
    y_interp = []
    error_est = []
    for x in x_interp:
        y = datay.copy()
        M = len(datax)

        for k in range(1, M):
            for i in range(M-k):
                y[i] = ((x - datax[i+k])*y[i] + (datax[i] - x)*y[i+1]) / (datax[i] - datax[i+k])
        y_interp.append(y[0])
        error_est.append(np.abs(y[0] - y[1]))
    return y_interp, error_est

yyb, yyb_err= neville(x, y, xx)
yb, yb_err= neville(x, y, x)


# Let's implement the error canceling algorithm.

def error_cancel(A_orig, crout_LU, y, coeffs, iterations):
    for _ in range(iterations):
        v = A_orig @ coeffs - y
        fw_result = forward_sub2(crout_LU, v)
        coeff_corr = backward_sub2(crout_LU, fw_result)
        coeffs = coeffs - coeff_corr
    return coeffs

c1_coeffs = error_cancel(A_orig, crout_LU, y, coeffs, iterations=1)
yyc1= polynomial(xx, c1_coeffs)
yc1= polynomial(x, c1_coeffs)

c10_coeffs = error_cancel(A_orig, crout_LU, y, coeffs, iterations=10)
yyc10= polynomial(xx, c10_coeffs)
yc10= polynomial(x, c10_coeffs)

#Don't forget to output the coefficients you find with your LU routine
print('Coefficients found with LU method (a):\n', coeffs)

print('Coefficients found with 1 iteration of LU correction (c):\n', c1_coeffs)

print('Coefficients found with 10 iterations of LU correction (c):\n', c10_coeffs)
#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)
```

```python
axs[0].set_ylim(-400,400)
axs[0].set_ylabel('$y$')
axs[1].set_ylim(1e-16,1e1)
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('my_vandermonde_sol_2a.png',dpi=600)

#For questions 2b and 2c, add this block
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('my_vandermonde_sol_2b.png',dpi=600)

#For question 2c, add this block too
line,=axs[0].plot(xx,yyc1,linestyle='dotted',color='red')
line.set_label('LU with 1 iteration')
axs[1].plot(x,abs(y-yc1),linestyle='dotted',color='red')
line,=axs[0].plot(xx,yyc10,linestyle='dashdot',color='purple')
line.set_label('LU with 10 iterations')
axs[1].plot(x,abs(y-yc10),linestyle='dashdot',color='purple')
axs[0].legend(frameon=False,loc="lower left")
plt.savefig('my_vandermonde_sol_2c.png',dpi=600)

#Don't forget to caption your figures to describe them/
#mention what conclusions you draw from them!

# Now, finally, time the different methods.

def problem_2a():
    coeffs, crout_LU = get_coeffs(y, A)
    yya = polynomial(xx, coeffs)

def problem_2b():
    yyb, yyb_err= neville(x, y, xx)

def problem_2c():
    coeffs, crout_LU = get_coeffs(y, A)
    c10_coeffs = error_cancel(A_orig, crout_LU, y, coeffs, iterations=10)
    yyc10= polynomial(xx, c10_coeffs)

number = 150
two_a_time = timeit.timeit(lambda : problem_2a(), number = number)

two_b_time = timeit.timeit(lambda : problem_2b(), number = number)

two_c_time = timeit.timeit(lambda : problem_2c(), number = number)

print(f'Time for {number} iterations of 2a:', two_a_time)
print(f'Time for {number} iterations of 2b:', two_b_time)
print(f'Time for {number} iterations of 2c:', two_c_time)
```

Vandermonde.py

2.1 2a: fitting by LU decomposition

The relevant code is given by:

```python
import numpy as np
import sys
import os
import matplotlib.pyplot as plt
import copy
import timeit
```

```python

def crout(A):
    m, n = A.shape
    for i in range(m):
        for j in range(n):
            if i >= j:
                A[i, j] = A[i, j] - np.sum(A[i, :j]*A[:j, j])
            else:
                A[i, j] = (A[i, j] - np.sum(A[i, :i]*A[:i, j]))/A[i, i]

    return A

def forward_sub2(A, b):
    m, _ = A.shape
    y = copy.deepcopy(b)
    for k in range(m):
        y[k] = (b[k] - np.sum(A[k, :k]*y[:k]))/A[k, k]
    return y

def backward_sub2(A, b):
    _, n = A.shape
    x = np.zeros_like(b)
    for i in range(n-1, -1, -1):
        x[i] = b[i] - np.dot(A[i, i+1:], x[i+1:])
    return x

def get_coeffs(y, A):
    A_decomp = crout(A)
    fw_sub = forward_sub2(A_decomp, y)
    coeffs = backward_sub2(A_decomp, fw_sub)

    return coeffs, A_decomp

def polynomial(x, coeffs):
    y = np.zeros_like(x)
    for i in range(len(x)):
        for j in range(len(coeffs)):
            y[i] += coeffs[j] * x[i]**j
    return y

# Let's get started
data=np.genfromtxt(os.path.join(sys.path[0],"Vandermonde.txt"),comments='#',dtype=np.
    float64)

x=data[:,0]
y=data[:,1]

# Construct the matrix.
A = np.zeros([20, 20])
m, n = A.shape
for i in range(m):
    for j in range(n):
        A[i, j] = x[i]**j

A_orig = copy.deepcopy(A)
# now set up the polynomial function!

coeffs, crout_LU = get_coeffs(y, A)
xx=np.linspace(x[0],x[-1],1001) #x values to interpolate at
yya = polynomial(xx, coeffs)
ya= polynomial(x, coeffs)
```

Vandermonde.py

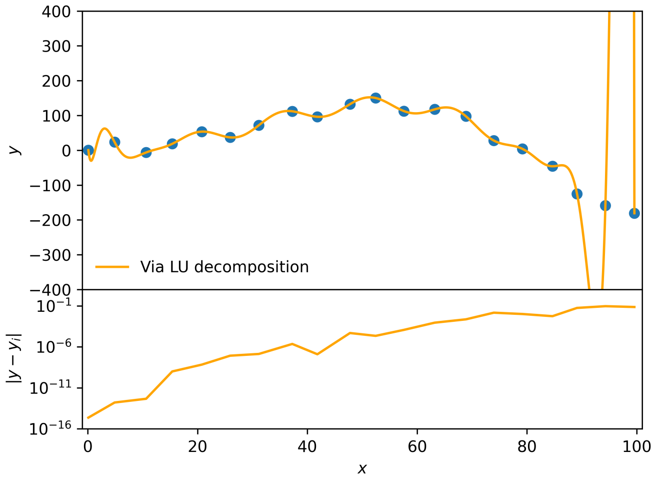
We start by reading in the data points and constructing the $N \times N$ Vandermonde matrix with each entry of the form $V_{ij} = x_i^j$, with i the row index and j the column index (both up to $N-1$). We then find the coefficients for the unique polynomial that goes through all given points by solving the system $\mathbf{Vc} = \mathbf{y}$ with y the given y-values for the points.

To do this, we overwrite the Vandermonde matrix by its LU decomposition using Crout's algorithm.

We then solve the system by applying forward substitution, followed by backward substitution.

The result is the unique polynomial given in figure 2.1.

Figure 1: Upper half: the given points in blue, with the polynomial interpolation using LU decomposition in orange. Bottom half: the absolute difference between the interpolation and the given y value at the given points.




```
Coefficients found with LU method (a):
 [ 1.70147517e+01 -1.86599980e+02  2.37038854e+02 -1.09410937e+02
  2.67337487e+01 -4.05711019e+00  4.16236505e-01 -3.03795406e-02
  1.63051551e-03 -6.57955383e-05  2.02524984e-06 -4.79493741e-08
  8.75580154e-10 -1.22942095e-11  1.31443054e-13 -1.05004562e-15
  6.06574150e-18 -2.39219555e-20  5.76088051e-23 -6.38865757e-26]
```

Vandermonde_output.txt

2.2 2b: fitting by Neville's algorithm

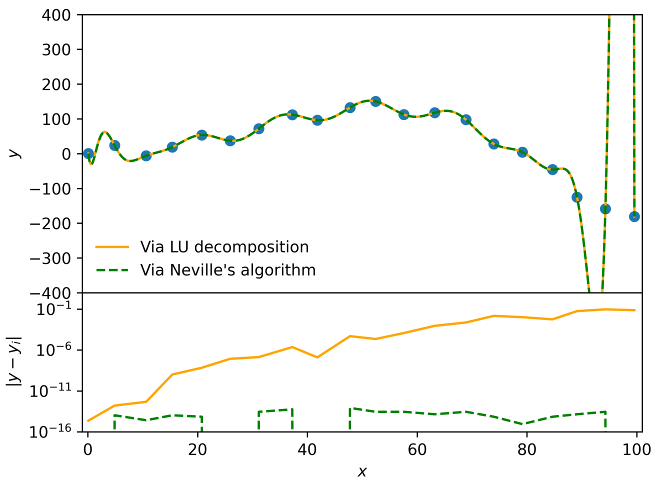
The relevant code is given by:

```python

# Now we'd like to implement neville's algorithm.

def neville(datax, datay, x_interp):
    y_interp = []
    error_est = []
    for x in x_interp:
        y = datay.copy()
        M = len(datax)

        for k in range(1, M):
            for i in range(M-k):
                y[i] = ((x - datax[i+k])*y[i] + (datax[i] - x)*y[i+1]) / (datax[i] - datax[i+k])
        y_interp.append(y[0])
        error_est.append(np.abs(y[0] - y[1]))
    return y_interp, error_est

yyb, yyb_err= neville(x, y, xx)
yb, yb_err= neville(x, y, x)
```

Vandermonde.py

Here, we simply pass an array of values to be interpolated into our Neville interpolation function. The resulting polynomial lies on top of the polynomial found in 2a. This is to be expected, as there is a single unique 19th order polynomial that goes through all of these points. See figure 2.2 below:

Figure 2: Upper half: the given points in blue, with the polynomial interpolation using LU decomposition in orange. The polynomial found by Neville's algorithm has been added, and we see that it lies on top of the previous one. Bottom half: the absolute difference between the interpolation and the given y value at the given points. The absolute difference for the second method is orders of magnitude smaller.




```
Coefficients found with 1 iteration of LU correction (c):
 [ 1.73261060e+01 -1.89903192e+02  2.39192506e+02 -1.10031275e+02
  2.68385894e+01 -4.06885525e+00  4.17169348e-01 -3.04342928e-02
  1.63295507e-03 -6.58795060e-05  2.02750625e-06 -4.79969536e-08
  8.76367347e-10 -1.23043727e-11  1.31544235e-13 -1.05080648e-15
  6.06991727e-18 -2.39377260e-20  5.76454210e-23 -6.39259513e-26]
```

Vandermonde_output.txt

In the next subsection, we shall discuss possible reasons for the difference in absolute error between methods.

2.3 2c: error-cancellation using LU decomposition

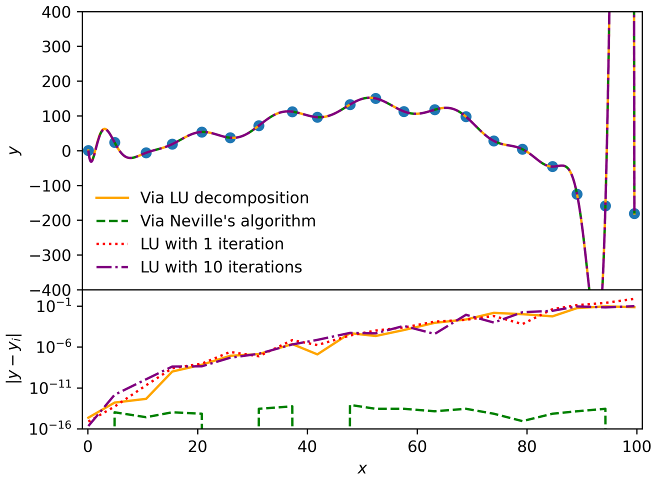
The relevant code is given by:

```

# Let's implement the error canceling algorithm.

def error_cancel(A_orig, crout_LU, y, coeffs, iterations):
    for _ in range(iterations):
        v = A_orig @ coeffs - y
        fw_result = forward_sub2(crout_LU, v)
        coeff_corr = backward_sub2(crout_LU, fw_result)
        coeffs = coeffs - coeff_corr
    return coeffs

c1_coeffs = error_cancel(A_orig, crout_LU, y, coeffs, iterations=1)
yyc1= polynomial(xx, c1_coeffs)
yc1= polynomial(x, c1_coeffs)

c10_coeffs = error_cancel(A_orig, crout_LU, y, coeffs, iterations=10)
yyc10= polynomial(xx, c10_coeffs)
yc10= polynomial(x, c10_coeffs)
```

Vandermonde.py

We now iterate on the solution of 2a. We can use the LU matrices calculated previously and solve a new system to find a correction on the coefficients. We do this once with 1 iteration, and then once more with 10 iterations of the error-cancelling algorithm. The results are shown in figure 2.3.

Figure 3: Upper half: the given points in blue, with the polynomial interpolation using LU decomposition in orange, as well as the polynomial found by Neville's algorithm. We have now added 2 additional interpolations, for different iterations of the LU error canceling algorithm. Bottom half: the absolute difference between the interpolation and the given y value at the given points. The absolute difference for the second method is orders of magnitude smaller. The error-canceled LU interpolations are very similar to the initial LU interpolation.




```
Coefficients found with 10 iterations of LU correction (c):
 [ 1.76004367e+01 -1.92813615e+02  2.41090086e+02 -1.10577877e+02
  2.69309767e+01 -4.07920689e+00  4.17991738e-01 -3.04825810e-02
  1.63510778e-03 -6.59536551e-05  2.02950069e-06 -4.80390585e-08
  8.77064968e-10 -1.23133950e-11  1.31634240e-13 -1.05148487e-15
  6.07365027e-18 -2.39518661e-20  5.76783589e-23 -6.39614985e-26]
```

Vandermonde_output.txt

2.4 2d: Timing the interpolation

Using the timeit module, we execute the code from 2a, 2b and 2c 150 times each. The results are given below:

```
Time for 150 iterations of 2a: 1.913271800003713
Time for 150 iterations of 2b: 28.733681085999706
Time for 150 iterations of 2c: 2.0703072220057948
```

Vandermonde_output.txt

We see that the LU decomposition method is quite quick, and the error cancelling doesn't add that much more time. Neville's algorithm is the clear loser when it comes to efficiency. When we take into account accuracy, though, it becomes apparent that Neville's algorithm does have significantly lower errors in this application. A possible cause for error with the LU decomposition method is roundoff.

We also see that iteration on the LU method