NUR Assignment I: solutions

Tina Neumann

March 9, 2023

Abstract

This document displays my solution for the first assignment in the course numerical recipes for astrophysics, summer term 2023.

1 Exercise 1: Poisson distribution

In this exercise the poisson probability distribution for integer k should be calculated with the given values.

The poisson probability distribution is given as:

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

While calculations a straight-forward derivation with 32-bit values would run into overflow-problems for the k-factorial, the given equation was converted into log-space which gives the following equation:

The result is calculated the following way:

$$log(P_\lambda(k)) = log(\lambda) * k - \lambda - \sum(k) \tag{2}$$

The $P_\lambda(k)$ for the given values are calculated for 32-bit and 64-bit values with into log-space converted equation and as comparison the same is derived using *scipy* via the given equation.

```
#!/usr/bin/env python
# coding: utf-8

# # Numerical recipies
# Assignment (09.03.23)
#
# Tina Neumann

# In[28]:


import numpy as np
import timeit


# In[40]:


### Exercise 1: Poisson distribution
# define given distribution
def poiss_32(lam, k):
    '''Function determines the poisson distribution P(k) of
    input: a positive mean (lam) and an integer (k)
    output: P(k)'''
    lam = np.float32(lam) #redefine as 32-bit integers
    k = np.int32(k)
    # to decrease the calculation time:
    # multiply by inverse instead of dividing
    # the values are considered to be in log-scale: log(products)--> sums
    # define k!
```

```
    f_frac = 1.
    for f in range(1,k): #range leaves out last element
        f_frac += np.log(f)

    log_p = np.int32(k)*np.log(np.float32(lam)) - np.float32(lam) - np.int32(f_frac) # logarithmic P(k)
    p = np.exp(np.float32(log_p)) #revert log-scale
    print('k! = ', f_frac)
    print('log(P(k)) = ', log_p)
    return p


# In[41]:


def poiss_64(lam, k):
    '''Function determines the poisson distribution P(k) of
    input: a positive mean (lam) and an integer (k)
    output: P(k)'''
    lam = np.float64(lam) #redefine as 64-bit integers
    k = np.int64(k)
    # define k!
    f_frac = 1.
    for f in range(1,k): #range leaves out last element
        f_frac *= f

    log_p = k*np.log(lam) - lam - np.log(f_frac) #logarithmic P(k)
    p = np.exp(log_p) #revert log-scale
    print('k! = ', f_frac)
    print('log(P(k)) = ', log_p)
    return p


# In[42]:


from scipy.special import factorial
def poiss(lam, k):
    '''Function determines the poisson distribution P(k) of
    input: a positive mean (lam) and an integer (k)
    output: P(k)'''
    lam = float(lam)
    k = np.int64(k)
    #with numpy functions
    # define k!
    try:
        f_frac = factorial(k)
        p = lam**k*np.exp(-lam)*f_frac**(-1)
    except:
        print('OverflowError (34, Numerical result out of range')
        p = 'nan(OverflowError)'
    return p


# In[43]:
# Save output as a text file
with open("1_PoissonDistribution.txt", "w") as file:
    file.write('# The given lambda, k \n# \t \t results for 32-bit data types for the Poisson-distribution P(k) \n# \t \t \t \t P(k) for 64-bit types \n# \t \t \t \t  \t \t P(k) calculated with scipy \n')
# read-in given values
with open('input_1a.txt') as f:
    lines = f.readlines()[2:]
    for line in lines:
            #define lambda, k
            mean_lam, k_int = line.split('\t')
            print(mean_lam,k_int)

            k_int = k_int[:-1] #str has additional '\n'
            print('The given values are lam & k: '+ mean_lam, ' & '+ k_int)
```

```

            # add values to output file
            with open("1_PoissonDistribution.txt", "a+") as file:
                file.write(str(mean_lam+'\t'+k_int+'\t'+f'{poiss_32(mean_lam,k_int):.6E}'+'\t'+f'{poiss_64(mean_lam,k_int):.6E}'+'\t'+f'{poiss(mean_lam,k_int):.6E}'+'\n'))
            print('For 32 bits this results in a poisson distribution of P(k) = ', f'{poiss_32(mean_lam,k_int):.6E}') #print 6 relevant digits
            print('For 64 bits this results in a poisson distribution of P(k) = ', f'{poiss_64(mean_lam,k_int):.6E}') #print 6 relevant digits
            print('For numpy-functions this results in a poisson distribution of P(k) = ', f'{poiss(mean_lam,k_int):.6E}') #print 6 relevant digits
```

1_PoissonDistribution.py

The result of $P_\lambda(k)$ are given in:

```
# The given lambda, k
#       results for 32-bit data types for the Poisson-distribution P(k)
#               P(k) for 64-bit types
#                       P(k) calculated with scipy
1    0    1.353353E-01     3.678794E-01     3.678794E-01
5    10   1.487303E-01     1.813279E-01     1.813279E-02
3    21   1.101540E-10     2.140614E-10     1.019340E-11
2.6  40   1.000591E-31     1.446050E-31     3.615124E-33
101  200  2.376060E-16     0.000000E+00     NAN
```

1_PoissonDistribution.txt

2 Exercise 2: Vandemonde matrix

The Vandermonde-matrix can be obtained to find an unique solution for a given Langrangian polynomial . The coefficients of the Langrangian polynomial are calculated as:

$$y_i = \sum_{j=0}^{N-1} g_j x_i^j \tag{3}$$

2.1 2.a) Approximation with LU-decomposition

The entries of the Vandermonde-matrix ($V_{ij} = x_i^j$) are given and can read in by the following script:

```
# # Numerical recipies
# Assignment I, exercise 2 (09.03.23)
#
# Tina Neumann

# In[1]:
import numpy as np
import sys
import os
import matplotlib.pyplot as plt

# copied vandermonde.py #script of van Daalen
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


# In[7]:
# approximate values via Nevilles algorithm (Ex 2,2) on the basis of bi-section
from nevilles_algo import *
from lu_decomposition import *

# plotting given data
```

```python
plt.plot(x,y, '+', label = 'Given data points')

## Exercise 2.a) solve matrix via LU-decomposition
# create vandemonde-matrix as polynomial
vdM = []
for i in range(len(x)): #create rows
    row_x = []
    for j in range(len(x)): #create columns
        row_x.append(x[j]**j) #define polynomial for vandemonde-matrix

    vdM.append(row_x) #append polynomial for row elements, of same x value

lmat, umat, a_matrix = LU_decomp(vdM) #use code from Tut3
xsol, ysol = solve_LU(y,lmat,umat,len(y))
plt.plot(x, xsol, ':', alpha = 0.6, label = 'Interpolation with LU-decomposition')

## Exercise 2.b) interpolation with nevilles algorithm
y_nev = []
err_nev = []
for a in xx:
    y2, derr = nevilles_algo(a,x,y) #cal create function of Tut2
    y_nev.append(y2)
    err_nev.append(derr)

plt.plot(xx, y_nev, '-', alpha = 0.6, label = 'Interpolation with Nevilles algo')

## Exercise 2.c) solve matrix via LU-decomposition for 10 iterations
xsol10 = y
for l in range(10):
    #repeat solver 10-times while implementing the difference between the obtained result and the given y-values
    x10, y10 = solve_LU(xsol10,lmat,umat,len(xsol10))
    xsol10 = np.subtract(y,x10)

plt.plot(x, xsol10, ':', alpha = 0.6, label = 'Interpolation with 10xLU')

plt.xlabel('x')
plt.ylabel(r'$y_i = \sum_{j=0}{N-1} g_j x_i^j$')
plt.ylim(-400,400)
plt.title('Interpolation of Lagrangian polynom')
plt.legend()
plt.savefig('./plots/2_Interpolation_Algos.pdf')
plt.show()

### calculate the y-error produced (variance from given values) by
# 2.a) LU-decomposition
err_lu = []
err_lu10 = []
for m in range(len(xsol)):
    err_lu.append(abs(y[i]-xsol[i]))
    err_lu10.append(abs(y[i]-xsol10[i]))

plt.plot(x, err_lu, marker = '*', color = 'orange', label = 'Error of LU-decomposition')
plt.plot(x, err_lu10, marker = 'x', color = 'red', label = 'Error of 10xLU-decomposition')

# 2.b) through nevilles algo
# check diffence in y-value between given values and of nevilles algo produced
for i in range(len(x)):
    # write workaround (for .index()-to find the closest index) through smallest difference
    ind = np.abs(xx-x[i]).argmin()
    dy_nev = abs(y[i]-y_nev[ind])
    plt.plot(x[i],dy_nev, marker = '+', color = 'green') #, label = 'Error of Nevilles algo')
plt.plot(x[-1],dy_nev, marker = '+', color = 'green', label = 'Error of Nevilles algo') #to obtain label

plt.xlabel('x')
plt.ylabel(r'|$\Delta$ y|')
```

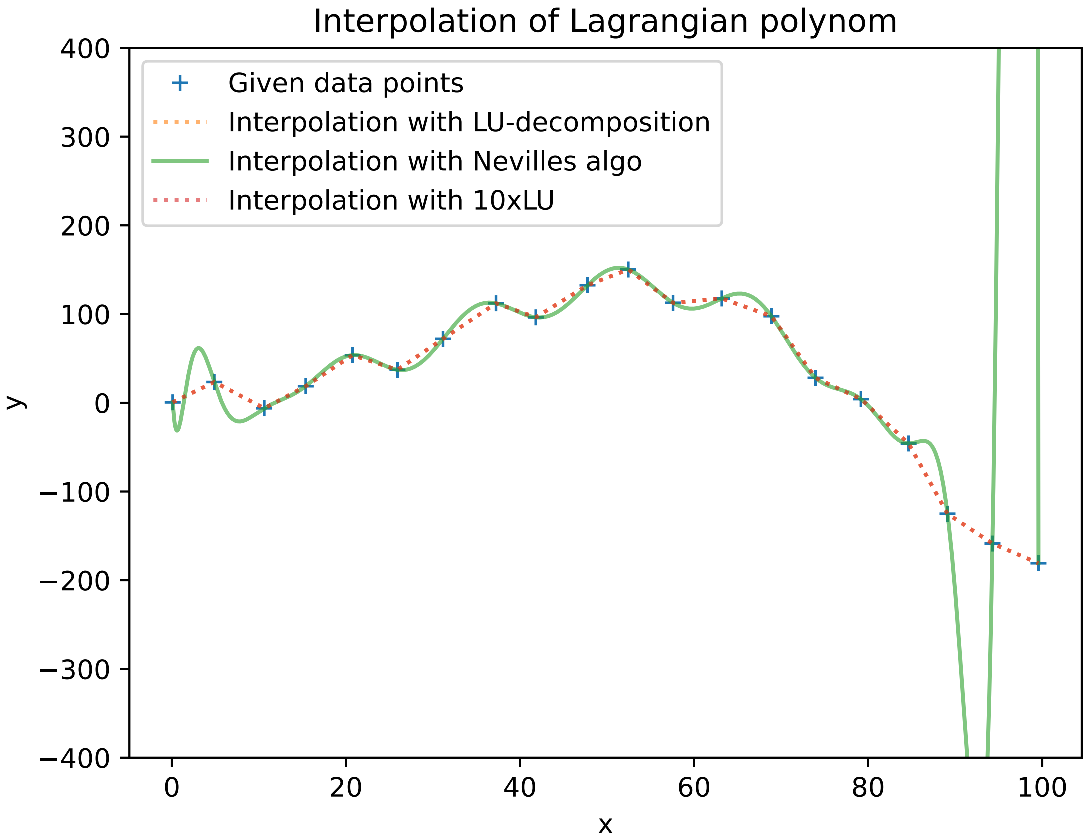

Figure 1: Goodness of fitting for each interpolation method

```
plt.yscale('log')
plt.title('Error comparison of Lagrangian polynom')
plt.legend()
plt.savefig('./plots/2_interpolation_error.pdf')
plt.show()
```

2_Vandemonde.py

With the code the following result for **c** is obtained, which is plotted as dashed line in figure **??**. The distribution of the data points is vizualized in the plot above.

2.2 2.b) Approximation with Nevilles algorithm

The result of $P_{\lambda}(k)$ for Nevilles algorithm vary from the LU-decomposition results because Nevilles algorithm focuses on fitiing the actual given data points while LU-decomposition adjusts/ optimizes also the fit in between theses points, which implies a loss of precision at the given values.

2.3 2.c) LU-decomposition of 2.a)

incomplete

2.4 2.d) Time previous sub-exercises

Within *timeit* the number of repititions considered to determine the runtime can be adjustested via the *repeat*-option. The default is set to 1e7. In the script below the number of repetitions are choosen as 1e2

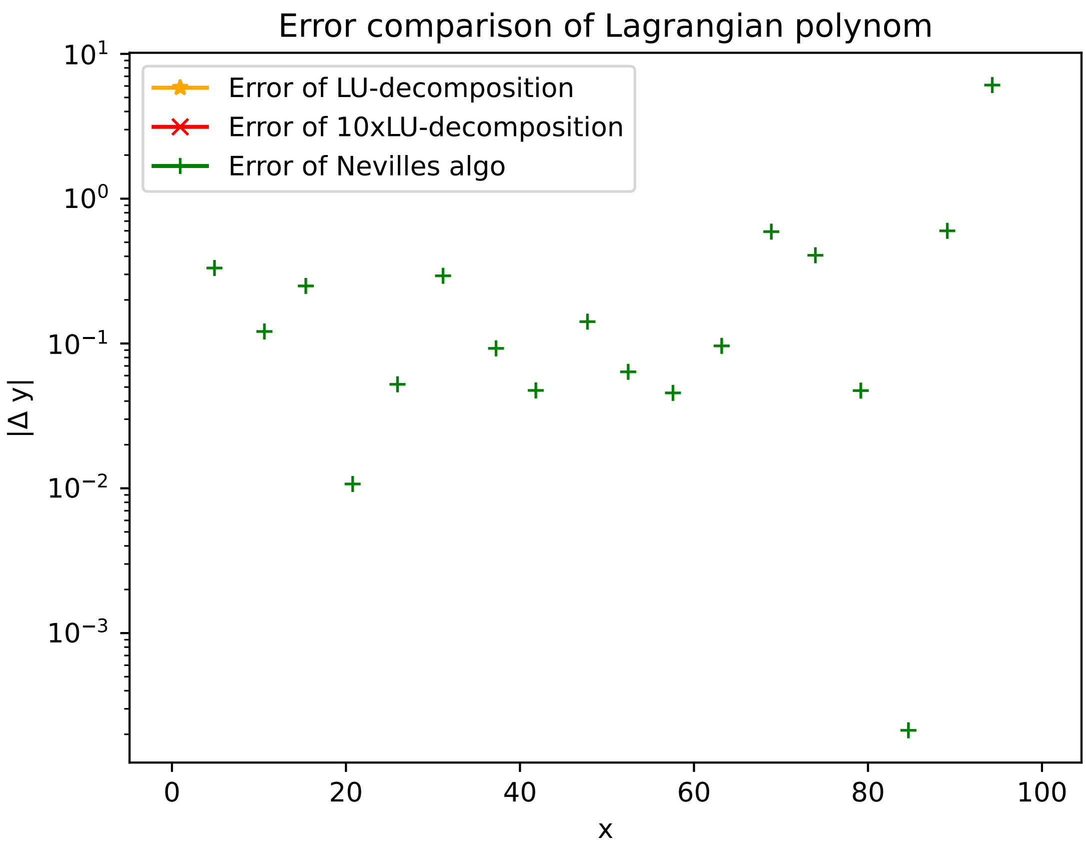

Figure 2: Error of each interpolation method

so that code runs completely within less than 1min. Nevilles algorithm takes for 1e2 repition 30 s which is approximately 30-time smore than the LU-decomposition. This means that LU-decomposition with one iteration is more efficient (faster running) but as it can be seen in 2 it also generates a bigger error. This can be decreased by more iterations (x10) of the LU-decomposition, which is still not reaching the precision as Neville's algorithm.

```python
# # Numerical recipies
# Assignment I, exercise 2 (09.03.23)
#
# Tina Neumann

# In[1]:
import timeit
import os
import matplotlib.pyplot as plt
import numpy as np
import sys

## Exercise 2.d) time routines with appropiate 'parameters'
repeat = int(1e2) #default 1e7: choose so that code runs below <1min

## Exercise 2.a) solve matrix via LU-decomposition
# create vandemonde-matrix as polynomial
print('The following time is needed to apply LU-decomposition and solve with the
    vandemonde-matrix the equation for c takes [sec]:')
print(timeit.timeit('lmat, umat, a_matrix = LU_decomp(vdM);xsol, ysol = solve_LU(y,lmat,
    umat,len(y))', setup = 'from readin_data import x,y,vdM,LU_decomp,solve_LU',number =
     repeat))


## Exercise 2.b) interpolation with nevilles algorithm
print('The time required to interpolate the matrix with Nevilles algorithm is [sec]:')
print(timeit.timeit('y_nev=[];err_nev=[];[nevilles_algo(a,x,y) for a in xx]', setup = '
    from readin_data import nevilles_algo,xx,x,y',number = repeat))

## Exercise 2.c) solve matrix via 10xLU-decomposition
print('The time required to interpolate via 10-times LU-decomposition solving [sec]:')
print(timeit.timeit('lmat, umat, a_matrix = LU_decomp(vdM);xsol10 = y;[(y- solve_LU(
    xsol10,lmat,umat,len(xsol10))[0]) for l in range(10)]', setup = 'from readin_data
    import xx,x,y,vdM,LU_decomp,solve_LU',number = repeat))
```

2d_timeit.py

The retrieved times are for Nevilles algorithm 23.268 s was to interpolate the matrix and and 0.663 s for the LU-decomposition and a few seconds slower than that for 10x-LU-decomposition.