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.

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.

1 Poisson

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.

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:

```python
#############
#
# Scripts for Q1 on Hand-in Assignment 1
#
#############

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)
```

```python
        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
```

λ	k	Self	Scipy height1
0	3.678794E-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	1.269727E-18	1.269531E-18 height	

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

```python
    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'

    with open('results/poisson_tab.txt', 'w') as file:
        file.write(table)


def main():
    compute_poisson_values()


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

../poisson.py

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 6 digits. When moving towards higher values we can see a slight discrepancy between "true" Scipy values, and our estimates on the order of $10^{\cdot\cdot}$

2 Vandermonde Matrix