PHY407 Lab3

Genevieve Beauregard (1004556045)
Anna Shirkalina (1003334448)

September 2020

Question 1

The code submissions this question is `Q1a.py`, `Q1b.py` for question 1a) and 1b) respectively. We also require that `Functions.py` to be in the same directory, as well as the provided `gaussint.py` and `gaussxw.py`.

a)

i)

We define a function `D_gauss` that calculates the Dawson function using a Gauss routine for a given value of x and number of slice N. We then created an array of of N -values which are powers of 2 from 8 to 2048, we called N. We looped over these and obtain the numerical integrals for each N[i] slices and printed them. Our output is in the following subsubsection, but you can view our formatted results to 3 s.f. in the table below. We used functions copied from the previous lab to calculate the Simpson's and trapezoidal numerical integrals.

N	x	Trapezoidal	Simpson's	Gauss
8	4	0.262	0.183	0.183
16	4	0.168	0.137	0.137
32	4	0.140	0.130	0.130
64	4	0.132	0.129	0.129
128	4	0.130	0.129	0.129
256	4	0.130	0.129	0.129
512	4	0.129	0.129	0.129
2048	4	0.129	0.129	0.129

RAW OUTPUT

```
Dawsons actual=> 0.1293480012360051

For N = 8 , x = 4
Our trap Dawson => 0.26224782053479523
```

```
Our simp Dawson => 0.18269096459712164
Our gauss Dawson => 0.18269096459712164

For N = 16 , x = 4
Our trap Dawson => 0.16828681895583716
Our simp Dawson => 0.13696648509618445
Our gauss Dawson => 0.13696648509618445

For N = 32 , x = 4
Our trap Dawson => 0.1395800909267732
Our simp Dawson => 0.13001118158375186
Our gauss Dawson => 0.13001118158375186

For N = 64 , x = 4
Our trap Dawson => 0.13194038496790617
Our simp Dawson => 0.12939381631495048
Our gauss Dawson => 0.12939381631495048

For N = 128 , x = 4
Our trap Dawson => 0.1299983024925397
Our simp Dawson => 0.12935094166741756
Our gauss Dawson => 0.12935094166741756

For N = 256 , x = 4
Our trap Dawson => 0.12951071531441982
Our simp Dawson => 0.1293481862550465
Our gauss Dawson => 0.1293481862550465

For N = 512 , x = 4
Our trap Dawson => 0.12938868844305068
Our simp Dawson => 0.129348012819261
Our gauss Dawson => 0.129348012819261

For N = 1024 , x = 4
Our trap Dawson => 0.12935817358096138
Our simp Dawson => 0.1293480019602649
Our gauss Dawson => 0.1293480019602649

For N = 2048 , x = 4
Our trap Dawson => 0.12935054435619742
Our simp Dawson => 0.1293480012812761
Our gauss Dawson => 0.1293480012812761
```

ii)

We plot our computed relative errors (to the scipy routine) on a scatter plot set to a log y and log x axis. We also consider the empirical error estimate in Equation 1 of the lab handout, where $\epsilon = I_{2N} - I_N$ for Gauss' method. We plot this on the same graph in Fig 1. Our results are to be expected: we have the relative error of the Gaussian method dropping at a far faster rate than its simpson's or trapezoidal methods as a function of N. We also note the failure of the equation at certain points (giving a zero error for example): this is not surprising as is simply an estimate that relies on the face that $\epsilon_{2N} << \epsilon_N$. If the errors become close to machine error, this condition begins to fail as they become equal resulting in the formula becoming inacurate or occasionally failing.

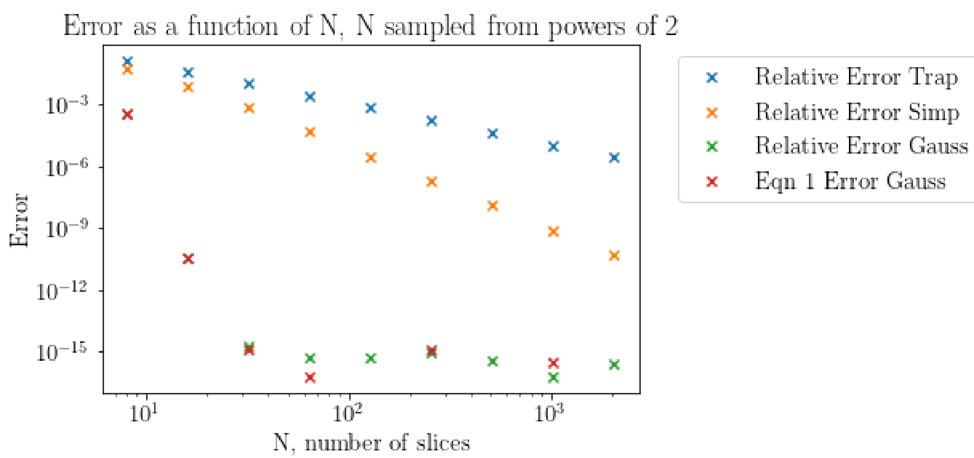

Figure 1: Relative errors of the three functions and the empirical error estimated plotted

b)

We plot $P(u_{10}, T_a, t_h)$ as per equation (2) in the handout, as a function of T_a, for different combinations of $u_{10} = (6, 8, 10)$ and $t_h = (24, 48, 72)$. We consider T_a from $-40°C$ to $30°C$ (basically the extreme range of temperatures one would consider probable in the Prairies) at a step of $1°C$. Our plot can be seen in Fig 2. We associate line style with the the value of u_{10} and line colour with the value of t_h used.

We see that for higher u_{10}'s, the higher average hourly windspeed, we get a higher probability for blowing snow. For example, the red curves where $t_h = 24$ hours. We have that the curve for $u_{10} = 10$ is higher than $u_{10} = 8$ which is then higher than $u_{10} = 6$. Similar observations can be made for other fixed t_h values. This dependence makes sense: higher wind speeds would pick up more snow.

We do a similar analysis for t_h. We see the snow surface age, the lower of the probability of blowing snow. If we clue on $u_{10} = 10$ (dashed lines), for example, we see that the probability is becomes lower as t_h rises. In this case, we see the curve is highest for $t_h = 24$, then $t_h = 48$ and then $t_h = 72$. This dependence also makes sense: older snow tends to pack more and thus does not get carried by the winds.

As the wind strength increases, the maxima of the probability curve moves more to the left as observed in Fig 2. Thus, the mostly likely temperature would decrease.

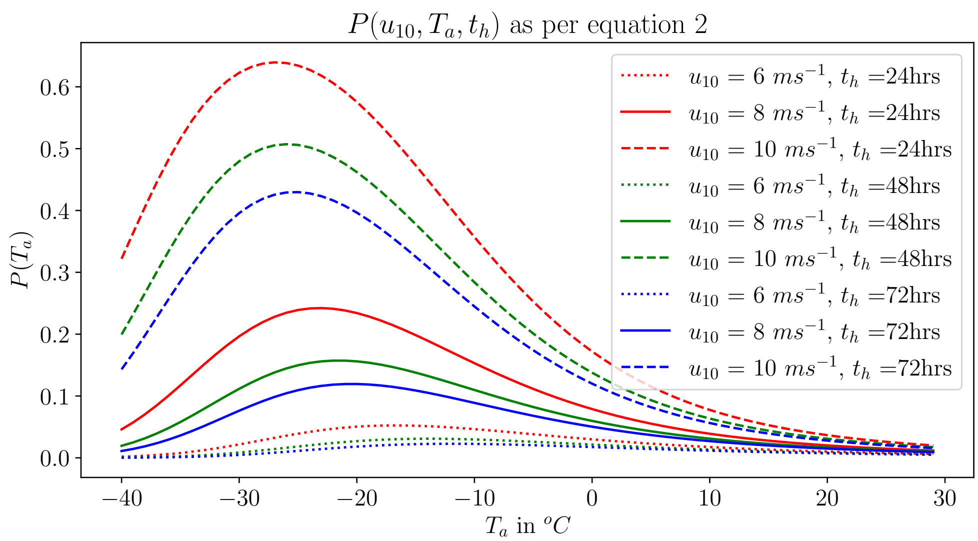

Figure 2: The probability of blowing snow in the Canadian Prairies

Question 2

The code submissions this question is `Lab3_Q2.py`. We also require that the provided `gaussxw.py` is located in the same directory.

a)

We define a function `H(n, x)` which takes as input the n and x, and outputs the $H_n(x)$. Originally `H(n, x)` was defined recursively but that proved to be computationally expensive since 2B took too long to run. Therefore we instead used a loop, and an array to store the values of the Hermite polynomials less than n.
We define the function `harmonic_oscillator(n, x)` to as given in equation 5 in the Physics Background. We can see the results displayed in 4.

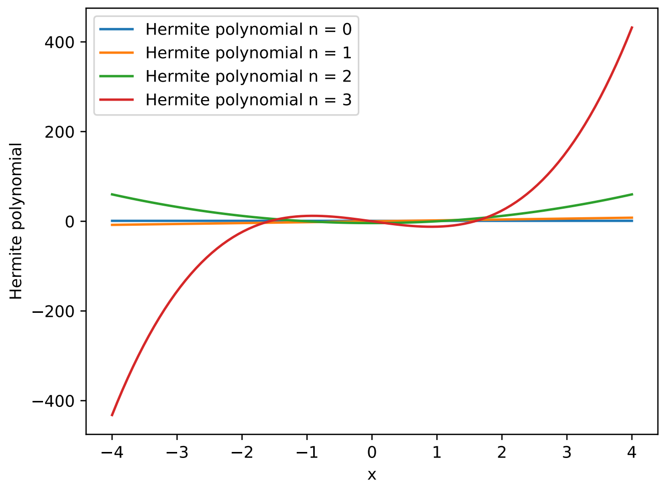

Figure 3: The Hermite polynomial for n = 0, 1,2,3

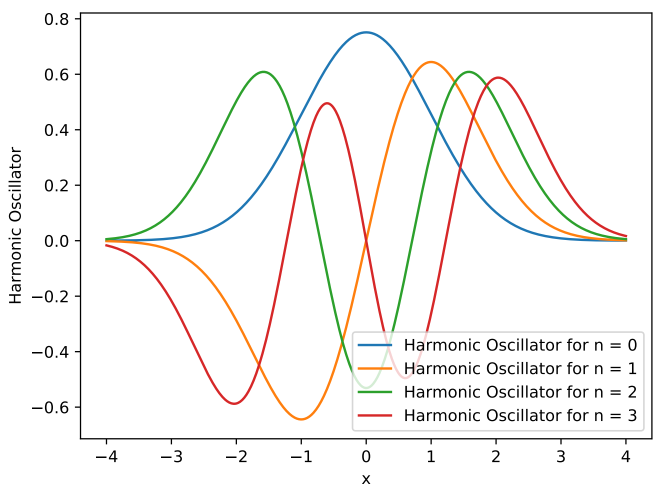

Figure 4: The wave function for n = 0, 1,2,3

b)

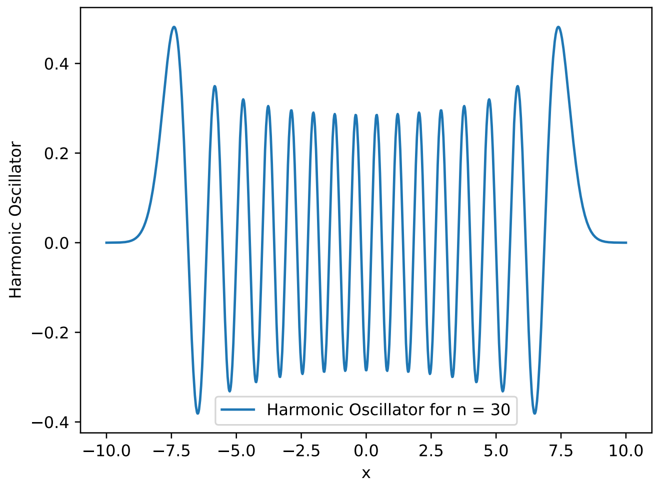

Figure 5: The wave function for n = 30

c)

To answer the question, the following algorithm was used:

```
#define the following functions
d_harmonic_oscillator_dx(n, x):
    """Returns the derivative of the quantum harmonic oscillator"""

x_mean_square_integrand(n, x):
    """Return the integrand of the quantum  position uncertainty of the nth
    level of a
    quantum oscillator, with the change of variable transformation z = tan(x)
    """


p_mean_square_integrand(n, x):
    """Return the integrand of the quantum  momentum uncertainty of the nth
    level of a quantum oscillator, with the change of variable transformation z = tan(x)
    """

energy_oscillator(x_ms, p_ms):
    Return the total energy of the oscillator 0.5 * (x_ms + p_ms)

# calculate the sample points and weights
```

```
gauss_weights(N, a, b):
    """Return the sample points, sample weights for guassian integration, use textbook code


guassian_integration(xp, wp, func):
    """
    Return the value of the integation using Guassian quadrature, use textbook code
    """

# get the weights from guass_weights, -pi /2 to pi /2

#initialize arrays for n, <x^2>, <p^2>

# set up for loop for over n:
    # guassian_intergration (lambda : x_mean_square_integrand)
    # guassian_intergration (lambda : p_mean_square_integrand)
    # append to <x^2>
    # append to <p^2>
    # calculate and append energy
```

The results are displayed below:

n	$< x^2 >$	$< p^2 >$	E	$\sqrt{< x^2 >}$	$\sqrt{< p^2 >}$
0	0.5000000000	0.5000000000	0.5000000000	0.7071067812	0.7071067812
1	1.5000000000	1.5000000000	1.5000000000	1.2247448714	1.2247448714
2	2.5000000000	2.5000000000	2.5000000000	1.5811388301	1.5811388301
3	3.5000000000	3.5000000000	3.5000000000	1.8708286934	1.8708286934
4	4.4999999998	4.4999999999	4.4999999998	2.1213203435	2.1213203435
5	5.4999999994	5.5000000000	5.4999999997	2.3452078798	2.3452078799
6	6.5000000112	6.5000000122	6.5000000117	2.5495097590	2.5495097592
7	7.5000000887	7.5000000408	7.5000000648	2.7386128037	2.7386127950
8	8.4999998407	8.4999995867	8.4999997137	2.9154759201	2.9154758766
9	9.4999963901	9.4999976286	9.4999970093	3.0822064159	3.0822066168
10	10.4999935533	10.5000057146	10.4999996339	3.2403693545	3.2403712310
11	11.5000617013	11.5000605130	11.5000611071	3.3911740889	3.3911739137
12	12.5002619278	12.5000026864	12.5001323071	3.5355709479	3.5355342858
13	13.4996550320	13.4991458209	13.4994004264	3.6741876697	3.6741183733
14	14.4961018266	14.4988404046	14.4974711156	3.8073746633	3.8077342875
15	15.4966147449	15.5071699928	15.5018923689	3.9365739857	3.9379144217

Table 1: Print out of the mean square of the position and moment, the energy total, and the root-mean-square for position and momentum

We can see in table 1 that the uncertainty in momentum and the uncertainty in position are roughly equivalent, for every value of n. We can observe that the function for energy in the oscillator is approximately equal to $E = n + 0.5$, and that $< x^2 >=< p^2 >= E$.

Note: the tabular environment was used to format the printout of the table, but has been commented out for the convience of the markers.

Question 3

The code submission for this question is `Question3.py`.

b)

We used file `N46E006.hgt`. Our pseudocode is as follows:

```
# define functions for dv/dx calculated using central, forward
# and backward differences:
    # forward and backward will use a h width and subtract over
    # one index. The central difference will use double width
    # and subtract over a difference of two indices.

# define Intensity function as per textbook

# Open file
# Set n size of grid to be 1201
# Set h value to be 420, set phi value to np.pi/6
# Initialize empty w array, I array, dwdy and dwdx nxn arrays

# Loop over indices of w array and fill array with data from file
# Plot w on nxn grid using imshow(). Use latitude and longtitude
# for x,y axis

# Initialize empty nxn array for dw/dx and dw/dy

# # Populate dwdy and dwdx arrays by doing the following,

# For loop over i,j indices of w array:

    # # We place conditionals to account for edge row cases
    # # where the central difference scheme is not applicable
    # # for dw/dy

    # if row index corresponds to the first row:
        # Set dwdx[i,j] using forward difference function
    # elif row index last row:
        # Set dwdx[i, j] using backward difference function
```

```
    # else:
        # Set dwdx[i,j] using central difference function

    # # We place conditionals to account for edge column
    # # cases where the central difference scheme is not
    # # applicable for dw/dy

    # if column index corresponds to the first column:
        # Set dwdy[i,j] using forward difference function
    # elif comlumn index the last row:
        # Set dwdy[i, j] using backward difference function
    # else:
        # Set dwdy[i, j] using central difference function

# For loop over i, j indices of I array:
    # Set I[i, j] to be intensity using intensity function
    # and corresponding derivatives

# Plot I on nxn grid using imshow(). Use latitude and
# longtitude for x,y axis.
```

b)

We implemented this programme with $\phi = \frac{\pi}{6}$ and $h = 420m$. We used file Our plots can be see in Fig 6 and 7. The biggest difference between the two plots is the clarity of the topography to the naked eye - the addition of shadows and bright patches allows us to discern the height differences easily. For the altitude map, we set vmin $\tilde{3}00$ m to be able to see some details (or as much as we can see). For the light intensity map, we set vmin = -0.05 and vmax = +0.05 in order to get visible shadows.

A quick glance allows us to identify the location of Lake geneva as the flat zone in the centre as seen in Fig 8. We can also also see the Alps south of Geneva as well as the French Jura Mountains to the north west. Other major features identified/found can be seen in Fig 9, which include Lake Neuchâtel as indicated by the dotted line.

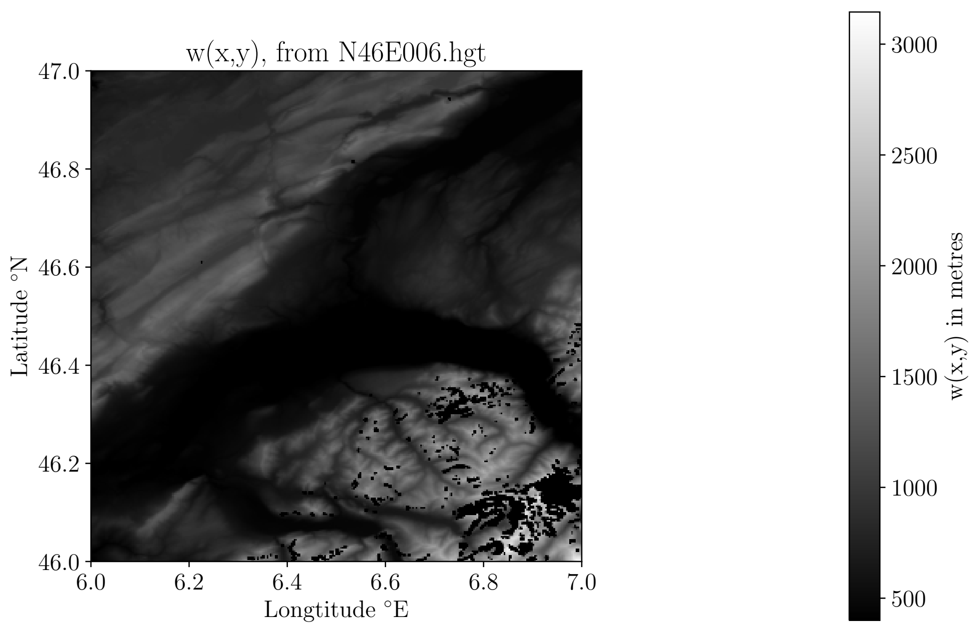

Figure 6: The raw data topographical $w(x, y)$ on tile `N46E006.hgt` around Lake Geneva.

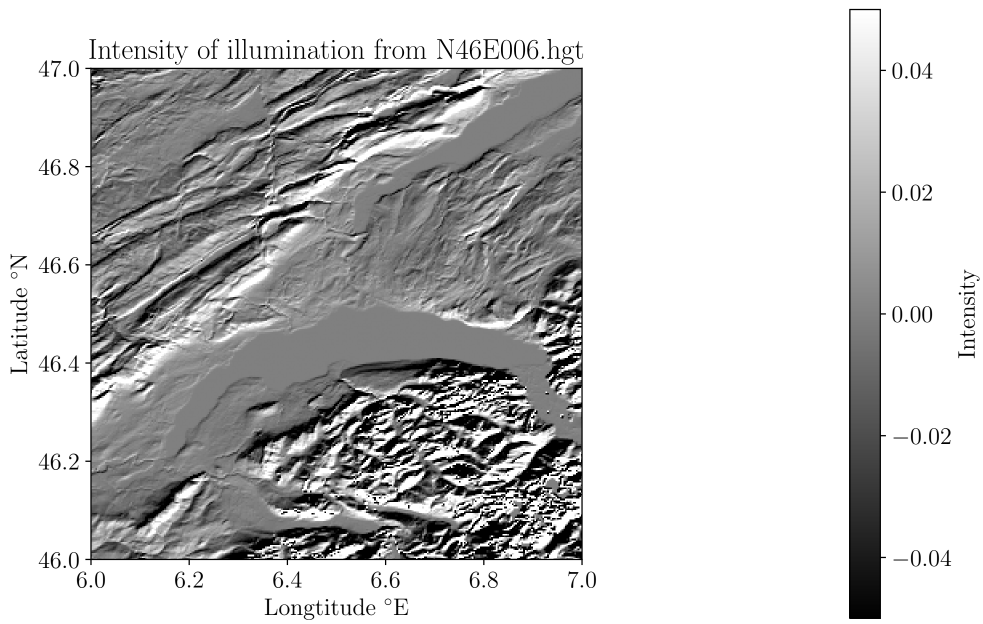

Figure 7: The intensity of illumination, as per the equation in the textbook, on tile `N46E006.hgt` around Lake Geneva.

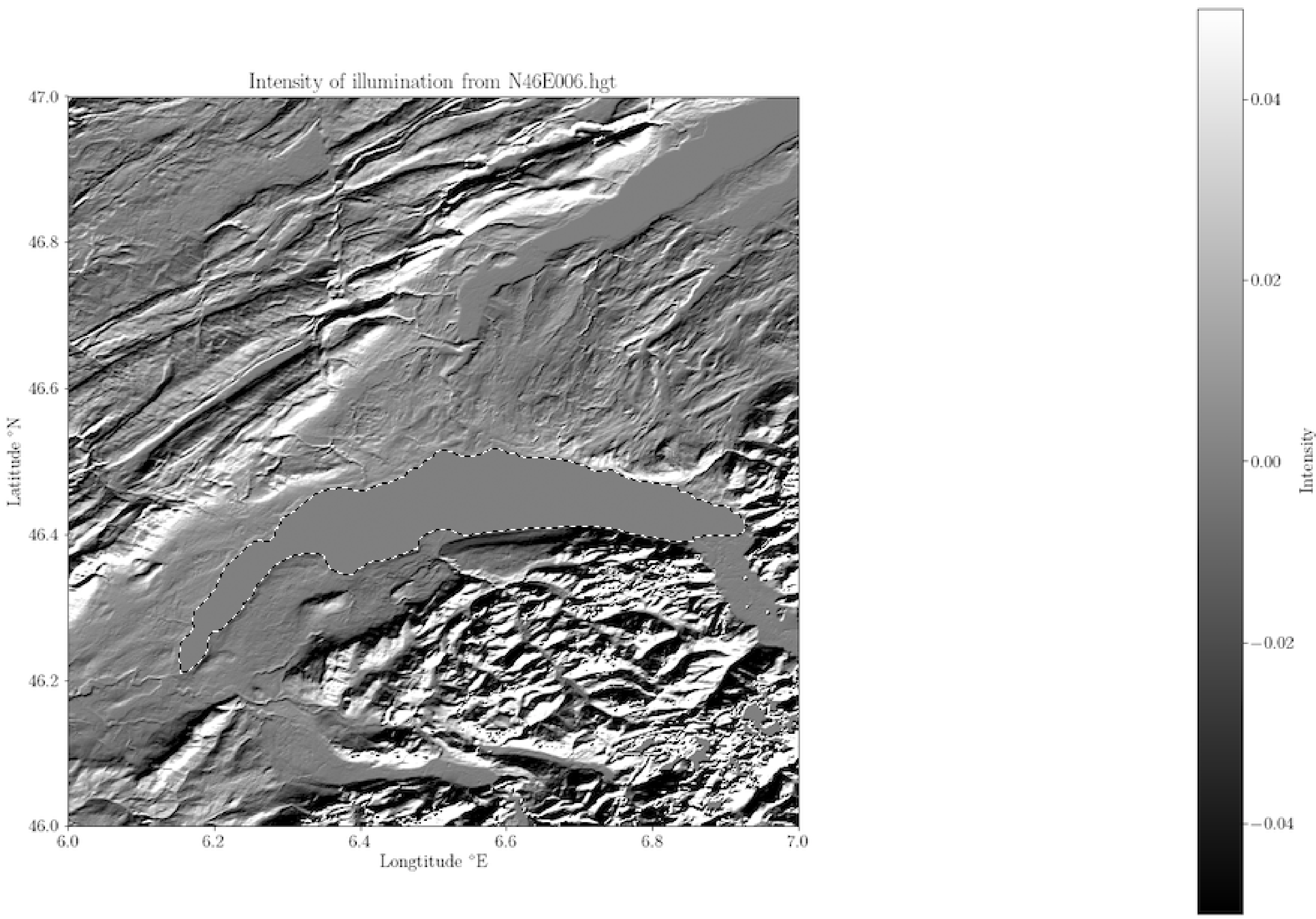

Figure 8: The border of Lake Geneva identified identified by the dotted line.

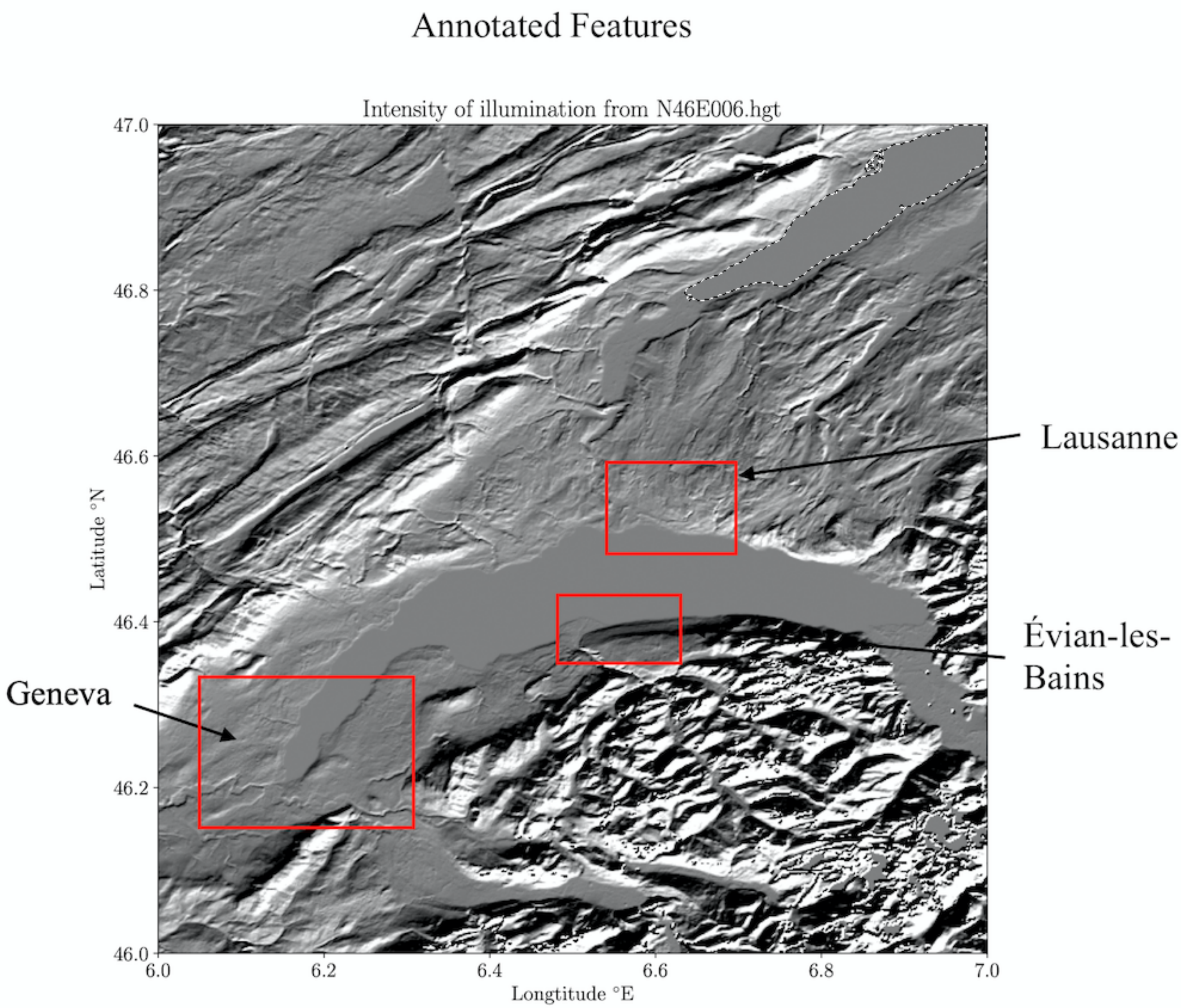

Figure 9: More locations of features. Google Maps was referenced. We also identified Lake Neuchâtel in the top right annotated by the dotted line. We omitted the intensity colour bar in order to fit the image, refer to the previous colour bar for reference.