PHY407 (Lab 3: Gaussian Quadrature and Numerical Differentiation)

Utkarsh Mali[1] and Aslesha Pokhrel[1,2]

[1]Department of Physics, University of Toronto
[2]Department of Computer Science, University of Toronto

October 1, 2020

Contents

Work Allocation:
Aslesha: Question 1, Question 3
Utkarsh: Question 2, Question 3

Question 1

(a) Here we evaluate the Dawson function from Lab 2 by adding one more integration method called Gaussian quadrature.

(i) We reused the code to evaluate the Dawson function using Trapezoidal and Simpson's rule from the last week's lab and added the code for Gaussian quadrature from the book. Then, the Dawson function was evaluated at $x = 4$ using all three methods for a range of N slices/sample points between $N = 8$ and $N = 2048$ which is shown in the table below:

N	Trapezoidal rule	Simpson's rule	Gaussian Quadrature
8	0.26224782053479523	0.18269096459712167	0.12901067881706976
16	0.1682868189558372	0.13696648509618448	0.12934800119977766
32	0.1395800909267732	0.13001118158375186	0.12934800123600343
64	0.13194038496790617	0.12939381631495048	0.12934800123600468
128	0.12999830249253974	0.12935094166741756	0.12934800123600468
256	0.12951071531441982	0.12934818625504652	0.1293480012360042
512	0.12938868844305074	0.12934801281926095	0.1293480012360053
1024	0.12935817358096138	0.12934800196026486	0.1293480012360051
2048	0.12935054435619744	0.12934800128127627	0.12934800123600504

Table 1: Table showing the theoretical and practical difference between $\tilde{c}_i$ and c_i.

(ii) Here we evaluate the accuracy of all three methods by plotting the practical error estimation as well as absolute relative error. For the practical error estimation, Equation (5.28) is used for the Trapezoidal rule, Equation (5.29) is used for the Simpson's rule and Equation (5.66) is used for the Gaussian Quadrature from the textbook. Similarly, to evaluate the absolute relative error, `scipy.special.dawsn` is taken as the true value.

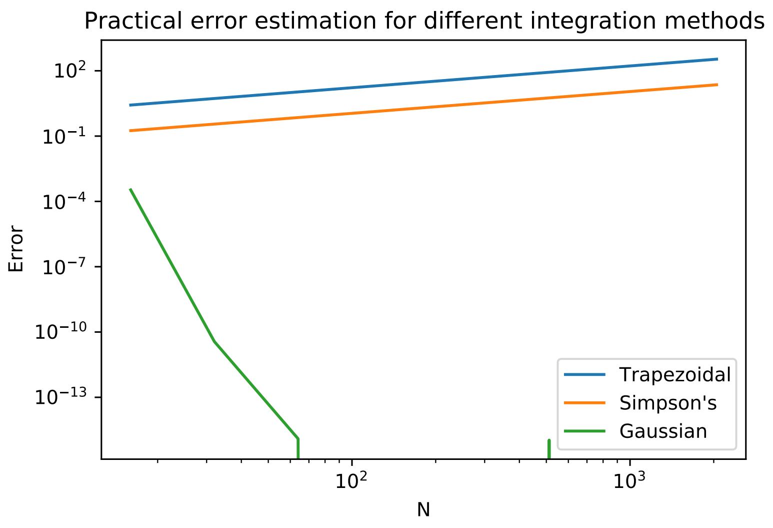

Figure 1: Plot showing the practical error estimation for all three methods in a log-log plot.

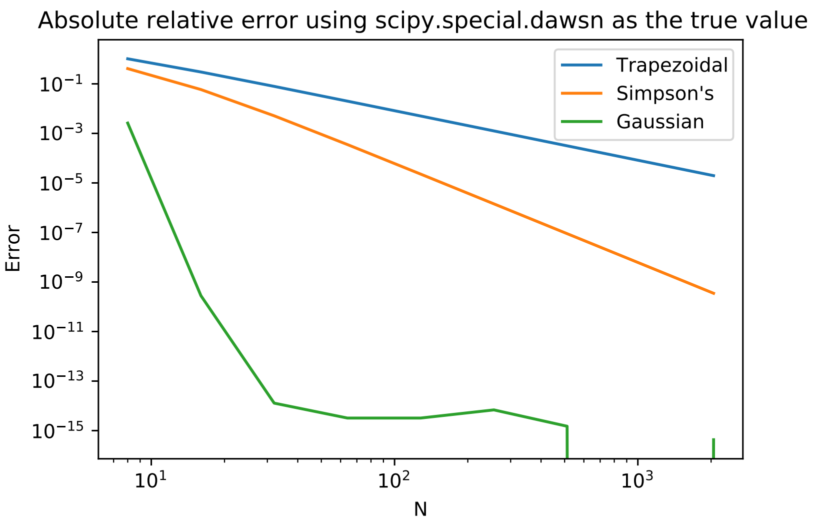

Figure 2: Plot showing the absolute relative error for all three methods in a log-log plot using `scipy.special.dawsn` as the true value.

From fig. 1 and fig. 2 we can see that the error for evaluating Dawson function using Gaussian Quadrature decreases swiftly with increasing N and the error is mostly 0 for the larger values of N. We see a similar pattern from both the practical error estimation and the relative error however, the value of N after which the error is 0 is different in these two cases. Similarly, the error for the Trapezoidal method is always the highest and the error for the Gaussian Quadrature method is always the smallest. However, the practical error estimation trend seems to be opposite from the absolute relative error for Trapezoidal and Simpson's rule. This might be due to the numerical error in estimating the error itself.

(b) **Here we use the Gaussian quadrature with $N = 100$ to calculate the probability of "blowing snow", P, (eqn. (2) in the question sheet) for $u_{10} = (6, 8, 10)ms^{-1}$ and $t_h = (24, 48, 72)$ hours. Then we plot P as a function of T_a.** The plot for probability vs. temperature in $°C$ (T_a) is produced for T_a ranging from $-60°C$ to $40°C$. Similarly, the curves for the same value of u_{10} is grouped using the same colour while the curves for the same value of t_h are grouped using the same line style.

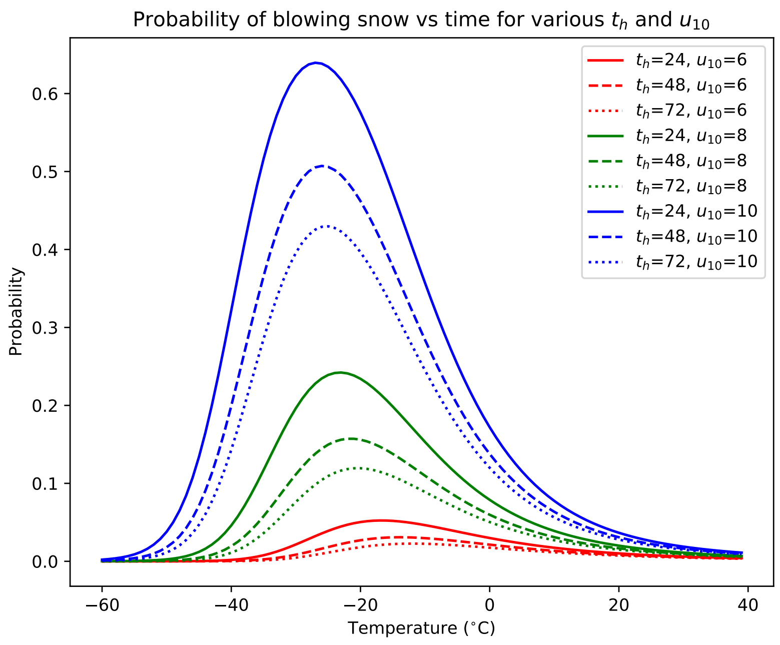

Figure 3: Plot showing the probability of blowing snow for various values of the snow surface age in hours (t_h) and the average hourly windspeed at a height of 10 m (u_10).

Figure 6 suggests that the probability of blowing snow is the highest for temperature around $-20°C$ and it decreases for the temperature higher and lower than that point. Similarly, the probability of blowing snow is higher for the lower value of t_h which means the fresh snow is more likely to create the blowing snow. Likewise, the probability is higher for higher value of u_10 which means the there is more likely to be blowing snow if the higher average hourly wind-speed. Both of these dependence is what we would expect to see in real life. The temperature at which blowing snow is most likely to occur decreases slightly with the increase in the wind-speed i.e. they seem to be inversely proportional to each other. For ex, the peak probability for $u_{10} = 10$ is around $-24°C$ whereas for $u_{10} = 6$ is around $-18°C$.

Question 2

(a) **Here we are to used a user defined H(x) to compute the wave function as specified in the lab handout. We are to then try multiple values of n to see if our plotter is working correctly.** We first defined the Hermite polynomials manually for values in which $n = 1$ and $n = 2$. Additional edge cases were also added, such as $n = 0$ and $n > 1$ for code stability. We then defined a recursive call in the final part of our function definition to recurse when n is greater than those values specified above. This was then applied into the wave function method. We checked if this was correct using `scip.special.eval_hermit()` which gave the correct analytic solution. The wave functions of this equation were then plotted as requested in the lab handout.

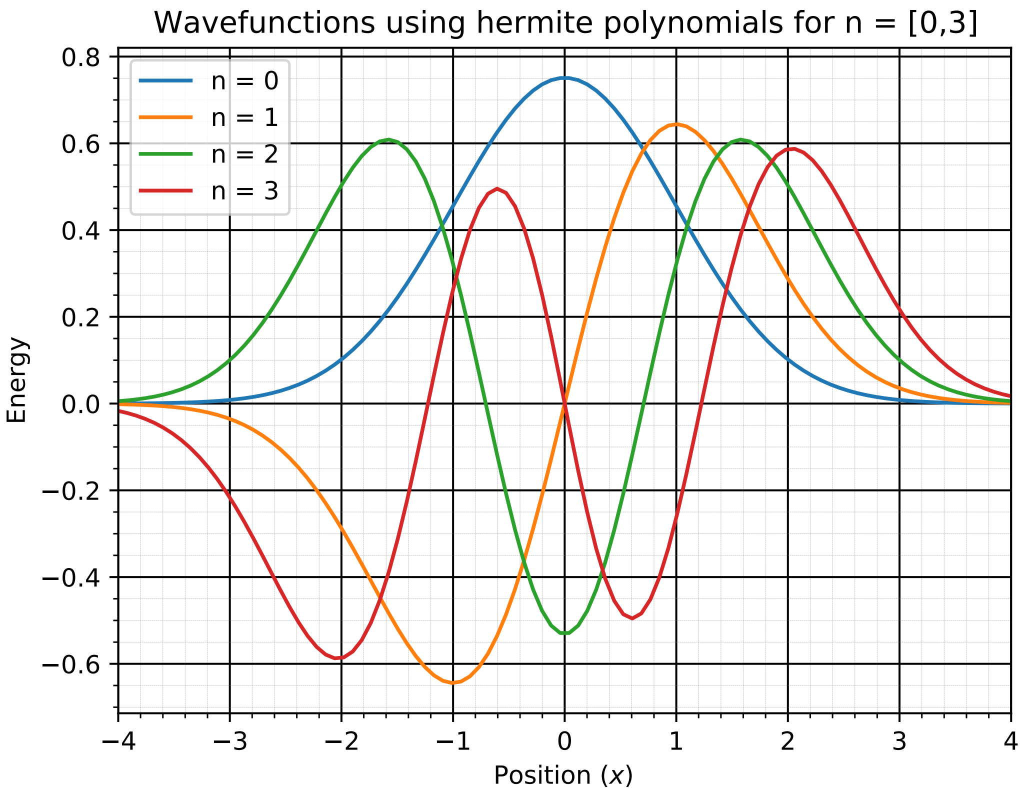

Figure 4: Here we have the plotted wave-functions which include our user defined Hermite polynomials between $n = 0$ and $n = 3$. We notice that all the even numbers result in even functions while all the odd numbers result in odd functions. This matches our quantum mechanics intuition. As n is increased we see additional nodes which linearly scale with n.

(b) This question requires us to make a separate plot for the wave-function at $n = 30$. When we initially tried to plot this we found that the run time was very lengthy. We fixed this by converting our function calculation in to a numpy array function calculation which calls much faster than its previous counterpart. Again we increased the `np.linspace` of x but we also increased the number of points to compute.

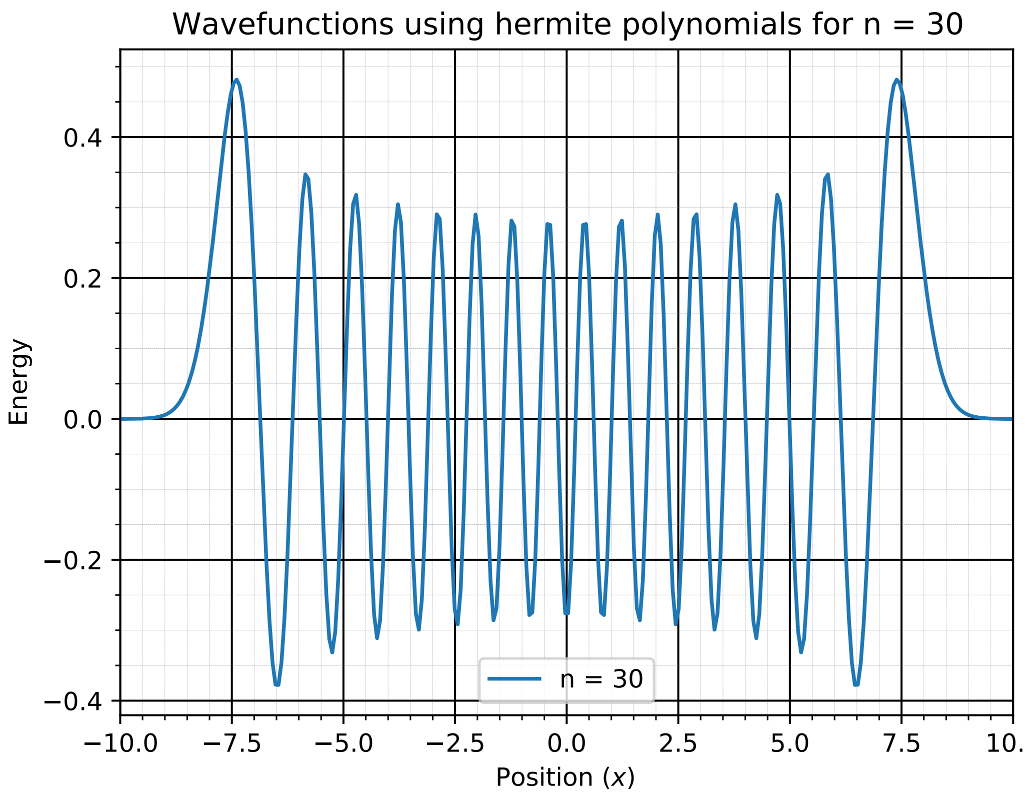

Figure 5: Here we see the wave-function constructed by the Hermite polynomials when we have set $n = 30$, as expected we observe many more nodes than our previous cases.

(c) Here we are asked to integrate using Gaussian quadrature and print the output of the functions. We are then asked to implement this method for all values of n between 0 and 15 First the `for` loop was initialized with an empty list in which the values of our integrals for various values of n would be placed. We then defined our integrand function which would be used in the Gaussian quadrature. We applied the $\pm\infty$ method of integration which was stated in pg 180 of the class textbook (Newman Computational Physics). Having set $N = 100$ as requested we carried out the integration and appended the position values. Similarly we set a new function to be the derivative of the wave-function required and carried out a similar integration using the same method. The square root of all these values were taken to formulate the root mean squared uncertainty and the half sum of both were taken in order to get the energies.

CODE OUTPUT:

```
[STATUS] Initializing
[STATUS] Running A...
[STATUS] A complete running B...
[STATUS] B complete running C...

position squared expectations (n=0,..,15):
 [ 0.5         1.5         2.5         3.5         4.49999998 5.49999997
  6.50000084 7.50000449 8.49998619 9.49984641 10.49989756 11.50222753
 12.50569488 13.4864421 14.42821377 15.49406764]

momentum squared expectations (n=0,..,15):
 [0.1767767 0.97227182 1.21533978 1.44735919 1.65037618 1.83198661
 1.99754213 2.15058172 2.29353844 2.42815749 2.55573702 2.67727203
 2.79354429 2.90518123 3.01269529 3.11651006]

energy values calculated (n=0,..,15):
 [0.33838835 1.23613591 1.85766989 2.4736796 3.07518808 3.66599329
 4.24877149 4.82529311 5.39676231 5.96400195 6.52781729 7.08974978
 7.64961958 8.19581167 8.72045453 9.30528885]

root mean squared position uncertainty:
 [0.70710678 1.22474487 1.58113883 1.87082869 2.12132034 2.34520787
 2.54950992 2.73861361 2.91547358 3.08218209 3.24035454 3.39149341
 3.53633919 3.67238915 3.79844886 3.93625045]

root mean squared momentum uncertainty:
 [0.42044821 0.98603845 1.1024245 1.20306242 1.28466968 1.353509
 1.41334431 1.46648618 1.51444328 1.55825463 1.59866726 1.63623715
 1.67138993 1.70445922 1.73571175 1.765364 ]


 Position uncertainty at n = 5, 2.34521

####################
[STATUS] Finished
####################
```

Question 3

(a) **Here we use the data from http://dds.cr.usgs.gov/srtm/version2_1/SRTM3/ to create the plots w and I using the equation in pg.212 from the textbook.** The pseudocode is given below:

```
"Pseudocode for Q3"
# Download the file "N46E006.hgt" from
    https://dds.cr.usgs.gov/srtm/version2_1/SRTM3/Eurasia/
# create an empty array for w
# open file for reading
# loop over the columns (i.e. run 1201 times)
   # initialize a temp array to store the contents of the row
   # loop over the rows (i.e. run 1201 times)
      # read 2 bytes
      # unpack the read struct and append to temp
   # append temp to the array w
# use imshow to plot w
# add labels and title
# calculate dwdx:
   # set h = 420
   # initialize an empty array dwdx
   # loop over the length of w
      # create a temp array to store the gradient for each row
      # loop over the length of w[0]
         # if it is the left-most pixel use forward difference (adapt code from the
             code snippet in the question sheet pg.3) to estimate the gradient then
             store the gradient in temp
         # if it is the right-most pixel use backward difference (adapt code from
             the code snippet in the question sheet pg.3) to estimate the gradient
             then store the gradient in temp
         # otherwise use central difference (adapt code from the code snippet in
             the question sheet pg.3) to estimate the gradient then store the
             gradient in temp
      # append temp to dwdx
# calculate dwdy:
   # initialize an empty array dwdy
   # loop over the length of w
      # create a temp array to store the gradient for each column
      # loop over the length of w[0]
         # if it is the top-most pixel use forward difference (adapt code from the
             code snippet in the question sheet pg.3) to estimate the gradient then
             store the gradient in temp
         # if it is the bottom-most pixel use backward difference (adapt code from
             the code snippet in the question sheet pg.3) to estimate the gradient
             then store the gradient in temp
         # otherwise use central difference (adapt code from the code snippet in
             the question sheet pg.3) to estimate the gradient then store the
             gradient in temp
      # append temp to dwdy
# set phi to the value given in the question
# calulate intensity by using the formula given in pg. 212 of the textbook:
    # calculate sq = sqrt((dwdx)^2 + (dwdy)^2 + 1)
    # then use sq to get I = - ((cos(theta) * dwdx + sin(theta) * dwdy) / (sq))
# plot I using imshow and use extent to change the axis numbering
# add labels and title
# save the figure
```

(b) The above code was implemented and the produced plots for w and I are given below:

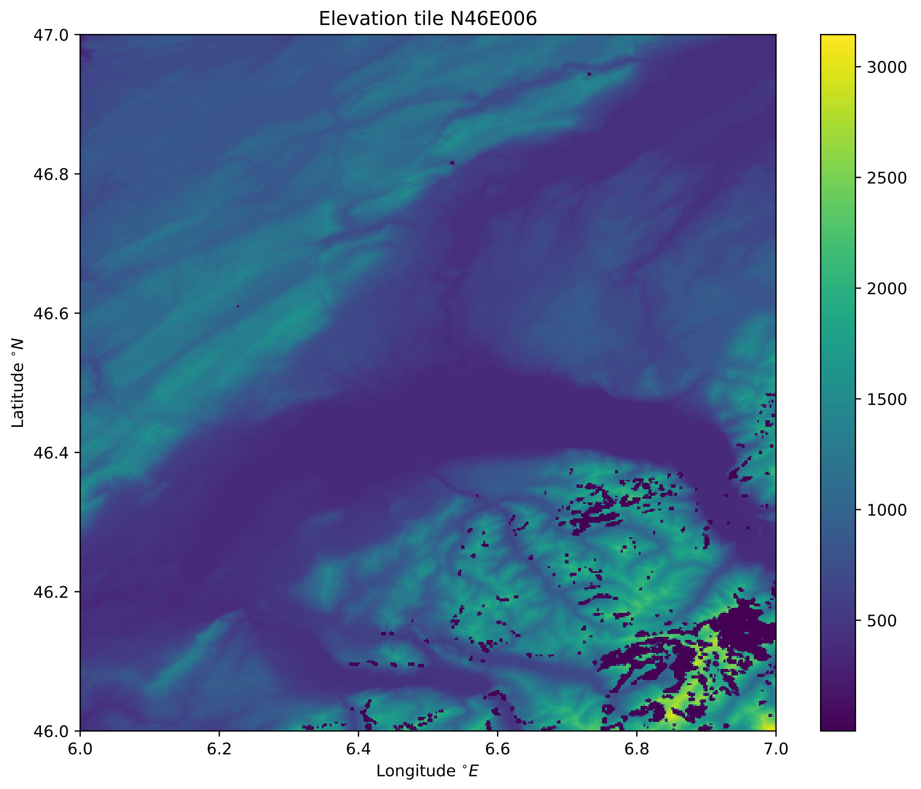

Figure 6: Plot showing the tile corresponding to Lake Geneva, at the border between France and Switzerland (i.e. N46E006.hgt).

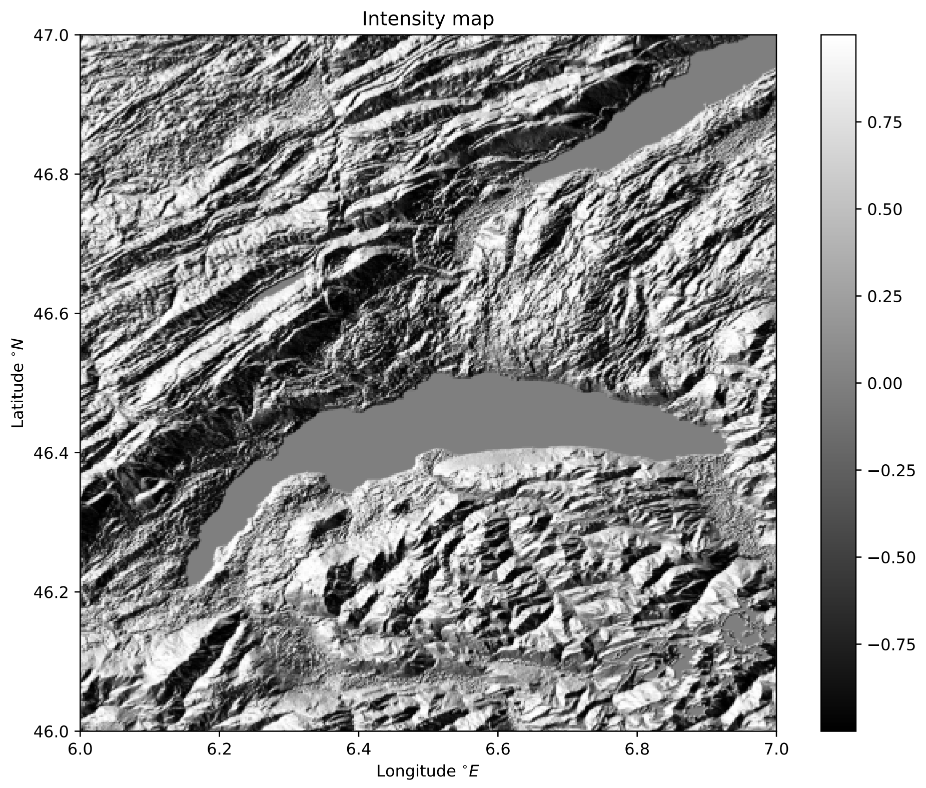

Figure 7: Plot showing the intensity map for N46E006.hgt.

The main difference between W map (fig. 6) and I map (fig. 7) is that the elevation is more clear in the I map and some structures like the lake stands out in the I map. The next plot shows some of the well known features of Geneva.

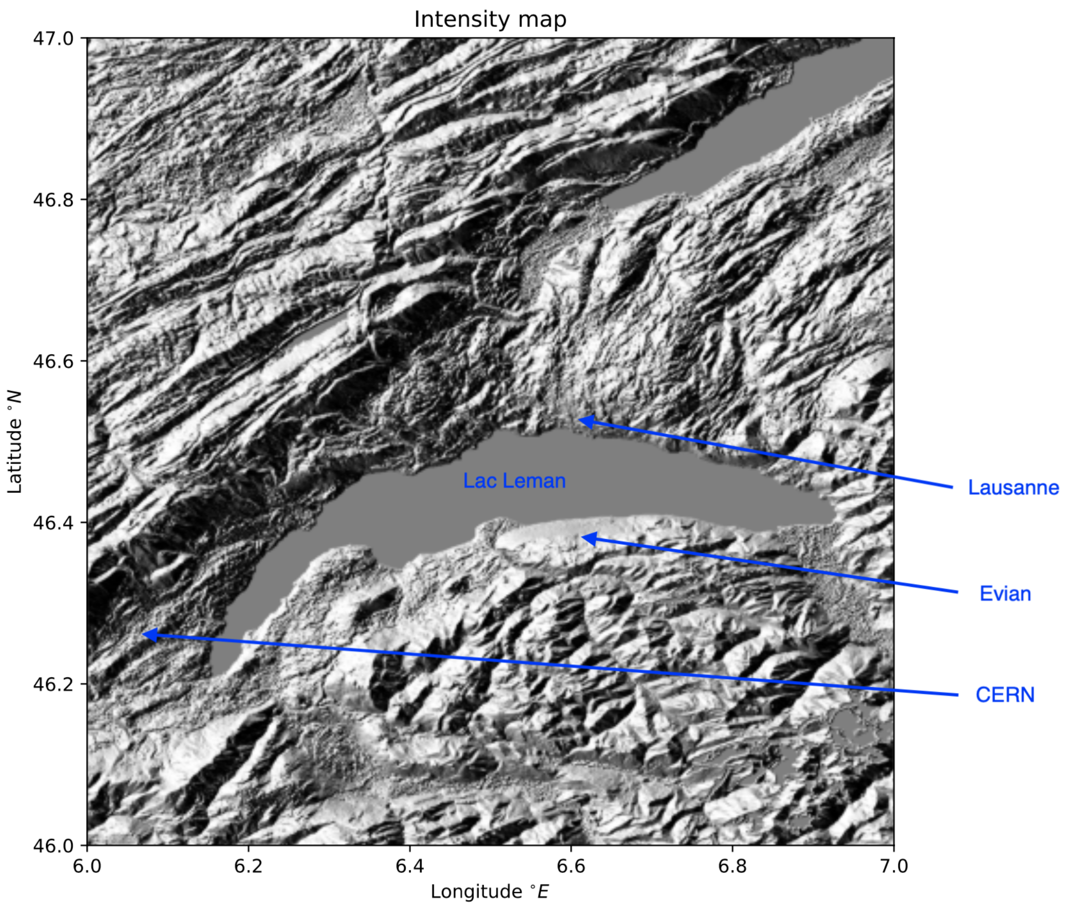

Figure 8: Plot showing some of the well known features of Geneva.