Parallel computing of geopotential and its functionals

Baroni Marco

Contents

Chapter 1

Introduction

Following I present the mathematical method, the computational method and the result of the two programs I written to calculate the global geopotential and the trajectory point-by-point of an Earth's satellite moving in a heterogeneous potential.

To write the program I used Cython language on the Jupyter Notebook. I choose to use Cython instead of Python because I needed the possibility of code parallellization to do the work.

To do the parallelization, in the cell where I wrote down all the functions, I put an header , this permits to Cython to directly use the OpenMP on the VSC. In case that a computer utilizes other compiler instead of VSC, other headers are needed, for example in GNU the header will be -fopenmp.

The mathematical approach is taken from (Fantino Casotto, 2008), in the program I adapted everything as from the paper, especially the calculation of the nALFs, for which I paid attention to calculate them row-wise, thus stabilizing the calculation.

On the other hand the calculation for the orbit was made taking advantage of a 4th order Runge-Kutta algorithm.

Chapter 2

Mathematical Approach

The disturbing potential is calculated from:

$$V = \frac{GM}{r} \sum_{n=2}^{N} \left(\frac{a}{r}\right)^n \sum_{m=0}^{n} P_n^m(\phi)\Big[C_{nm}\cos(m\lambda) + S_{nm}\sin(m\lambda)\Big] \tag{2.1}$$

In the satellite ground track application we used the full potential V*, given by:

$$V^* = \frac{GM}{r} + V = \frac{GM}{r}\left\{1 + \sum_{n=2}^{N} \left(\frac{a}{r}\right)^n \sum_{m=0}^{n} P_n^m(\phi)\Big[C_{nm}\cos(m\lambda) + S_{nm}\sin(m\lambda)\Big]\right\} \tag{2.2}$$

Where $P_n^m(\phi)$ are the normalized Associated Legendre Function (nALF), $a = 6378137$ m is the equatorial radius of Earth, C_{nm} and S_{nm} are the normalized Stokes coefficients for Earth, $GM = 3986004.415{\cdot}10^8\ m^3/s^2$ is the gravitational parameter of Earth, r is the distance taken into account in meters and λ and ϕ are the geographic coordinates in radiants.

To calculate the nALF we firstly define the following coefficients:

$$f_n = \sqrt{\frac{(1+\delta_{1n})(2n+1)}{2n}} \tag{2.3}$$

$$g_{nm} = \sqrt{\frac{4n^2-1}{n^2-m^2}} \tag{2.4}$$

$$h_{nm} = \sqrt{\frac{(2n+1)[(n-1)^2-m^2]}{(2n-3)(n^2-m^2)}} \tag{2.5}$$

$$k_{nm} = \sqrt{\frac{(2-\delta_{0m})(n-m)(n+m+1)}{2}} \tag{2.6}$$

Then, the nALF are calculated by recursive means, initializing the (0,0), (0,1) and (1,1) values

and then using the last two equation recursively to fill up the matrix:

$$P_n^m(\phi) := \begin{cases} P_0^0(\phi) = 1 \\ \\ P_1^1(\phi) = \sqrt{3}\cos(\phi) \\ \\ P_1^0(\phi) = \sqrt{3}\sin(\phi) \\ \\ P_n^n(\phi) = P_{n-1}^{n-1}(\phi)\cos(\phi) f_n \\ \\ P_n^m(\phi) = g_{nm} P_{n-1}^m(\phi)\sin(\phi) - h_{nm} P_{n-2}^m(\phi) \end{cases} \tag{2.7}$$

The derivatives of the nALF here found are then defined as:

$$\frac{dP_n^m(\phi)}{d\phi} = k_{nm} P_n^{m+1}(\phi) - m\tan(\phi) P_n^m(\phi) \tag{2.8}$$

$$\frac{d^2 P_n^m(\phi)}{d\phi^2} = \left[\frac{m^2}{\cos^2(\phi)} - n(n+1)\right] P_n^m(\phi) + \tan(\phi)\frac{dP_n^m(\phi)}{d\phi} \tag{2.9}$$

To calculate the disturbing potential in a more time-efficient way, we inverted the n and m summation, thus defining the so called *lumped coefficients* A_m, B_m and their derivatives:

$$\begin{aligned}
\begin{pmatrix} A_m \\ B_m \end{pmatrix} &= \sum_{n=m}^{N} \left(\frac{a}{r}\right)^n \begin{pmatrix} C_{nm} \\ S_{nm} \end{pmatrix} P_n^m(\phi) \\
\begin{pmatrix} \partial_r A_m \\ \partial_r B_m \end{pmatrix} &= \sum_{n=m}^{N} \left(\frac{a}{r}\right)^n (n+1) \begin{pmatrix} C_{nm} \\ S_{nm} \end{pmatrix} P_n^m(\phi) \\
\begin{pmatrix} \partial_\phi A_m \\ \partial_\phi B_m \end{pmatrix} &= \sum_{n=m}^{N} \left(\frac{a}{r}\right)^n \begin{pmatrix} C_{nm} \\ S_{nm} \end{pmatrix} \frac{dP_n^m(\phi)}{d\phi} \\
\begin{pmatrix} \partial^2_{rr} A_m \\ \partial^2_{rr} B_m \end{pmatrix} &= \sum_{n=m}^{N} \left(\frac{a}{r}\right)^n (n^2+3n+2) \begin{pmatrix} C_{nm} \\ S_{nm} \end{pmatrix} P_n^m(\phi) \\
\begin{pmatrix} \partial^2_{\phi r} A_m \\ \partial^2_{\phi r} B_m \end{pmatrix} &= \sum_{n=m}^{N} \left(\frac{a}{r}\right)^n (n+1) \begin{pmatrix} C_{nm} \\ S_{nm} \end{pmatrix} \frac{dP_n^m(\phi)}{d\phi} \\
\begin{pmatrix} \partial^2_{\phi\phi} A_m \\ \partial^2_{\phi\phi} B_m \end{pmatrix} &= \sum_{n=m}^{N} \left(\frac{a}{r}\right)^n \begin{pmatrix} C_{nm} \\ S_{nm} \end{pmatrix} \frac{d^2 P_n^m(\phi)}{d\phi^2}
\end{aligned} \tag{2.10}$$

At this point the disturbing potential is then:

$$V = \frac{GM}{r} \sum_{m=0}^{N} \left[A_m \cos(m\lambda) + B_m \sin(m\lambda)\right] \tag{2.11}$$

Its gradient will have components:

$$\begin{aligned}
\partial_r V &= -\frac{GM}{r^2}\sum_{m=0}^{N}\left[\partial_r A_m \cos(m\lambda) + \partial_r B_m \sin(m\lambda)\right] \\
\partial_\phi V &= \frac{GM}{r}\sum_{m=0}^{N}\left[\partial_\phi A_m \cos(m\lambda) + \partial_\phi B_m \sin(m\lambda)\right] \\
\partial_\lambda V &= -\frac{GM}{r}\sum_{m=0}^{N} m\left[A_m \sin(m\lambda) - B_m \cos(m\lambda)\right]
\end{aligned} \tag{2.12}$$

And its Hessian will have components:

$$\begin{aligned}
\partial^2_{rr} V &= \frac{GM}{r^3}\sum_{m=0}^{N}\left[\partial^2_{rr} A_m \cos(m\lambda) + \partial^2_{rr} B_m \sin(m\lambda)\right] \\
\partial^2_{\phi\phi} V &= \frac{GM}{r}\sum_{m=0}^{N}\left[\partial^2_{\phi\phi} A_m \cos(m\lambda) + \partial^2_{\phi\phi} B_m \sin(m\lambda)\right] \\
\partial^2_{\lambda\lambda} V &= -\frac{GM}{r}\sum_{m=0}^{N} m^2\left[A_m \cos(m\lambda) + B_m \sin(m\lambda)\right] \\
\partial^2_{r\phi} V &= -\frac{GM}{r^2}\sum_{m=0}^{N}\left[\partial^2_{r\phi} A_m \cos(m\lambda) + \partial^2_{r\phi} B_m \sin(m\lambda)\right] \\
\partial^2_{r\lambda} V &= \frac{GM}{r^2}\sum_{m=0}^{N} m\left[\partial_r A_m \sin(m\lambda) - \partial_r B_m \cos(m\lambda)\right] \\
\partial^2_{\phi\lambda} V &= \frac{GM}{r}\sum_{m=0}^{N} m\left[\partial_\phi A_m \sin(m\lambda) - \partial_\phi B_m \cos(m\lambda)\right]
\end{aligned} \tag{2.13}$$

Then, to calculate the ground track of the simulated satellite, we transformed the gradient from its polar form to its cartesian form:

$$\nabla_{cart} V = \begin{pmatrix} \partial_x V \\ \partial_y V \\ \partial_z V \end{pmatrix} = \begin{pmatrix} -\frac{\sin(\lambda)}{r\cos(\phi)}\partial_\lambda V - \frac{\sin(\phi)\cos(\lambda)}{r}\partial_\phi V + \cos(\phi)\cos(\lambda)\partial_r V \\ -\frac{\cos(\lambda)}{r\cos(\phi)}\partial_\lambda V - \frac{\sin(\phi)\sin(\lambda)}{r}\partial_\phi V + \cos(\phi)\sin(\lambda)\partial_r V \\ \frac{\cos(\phi)}{r}\partial_\phi V + \sin(\phi)\partial_r V \end{pmatrix} \tag{2.14}$$

We than used this gradient to perform the calculation of the acceleration vector acting on the satellite from the following:

$$\vec{a} = -\frac{GM}{|r|^3}\vec{r} + \vec{\nabla}_{cart} V \tag{2.15}$$

Where $\vec{r}$ is the position vector.

Then, step by step, the geographic coordinates were calculated from:

$$\phi = \arctan\left(\frac{z}{\sqrt{x^2+y^2}}\right) \tag{2.16}$$

$$\lambda = \arctan\left(\frac{y}{x}\right) - \omega_{\oplus}\Delta T - \alpha_{GW0} + \Omega \tag{2.17}$$

Where $\omega_{\oplus} = 0.004165403\frac{deg}{s}$ is the mean rotational velocity of Earth, and $\alpha_{GW0} \approx 100^{\circ}$ is the Greenwich sidereal time at the first of January and ΔT is the time step taken in seconds.
The *arctan* function we used in the Cython program was np.arctan2(), this to ensure that the calculation was made quadrant-wisely.

To obtain the initial conditions of the satellite we began from its Keplerian elements (a,i,ω,Ω,e,f) transforming them into initial cartesian position and velocity. To do so we firstly calculated the radius from the center of Earth at which the satellite is given the Keperian elements:

$$r = \frac{a(1-e^2)}{1+e\cos(f)} \tag{2.18}$$

And we calculated its mean motion:

$$n = \sqrt{\frac{GM_{\oplus}}{a^3}} \tag{2.19}$$

Having this, the cartesian set of initial condition is:

$$\begin{aligned}
x &= r\Big[\cos(\Omega)\cos(\omega+f) - \sin(\Omega)\sin(\omega+f)\cos(i)\Big] \\
y &= r\Big[\sin(\Omega)\cos(\omega+f) + \cos(\Omega)\sin(\omega+f)\cos(i)\Big] \\
z &= r\sin(\omega+f)\sin(i) \\
v_x &= -\frac{na}{\sqrt{1-e^2}}\Big\{\big[\sin(\omega+f)+e\sin(\omega)\big]\cos(\Omega) + \big[\cos(\omega+f)+e\cos(\omega)\big]\sin(\Omega)\cos(i)\Big\} \\
v_y &= -\frac{na}{\sqrt{1-e^2}}\Big\{\big[\sin(\omega+f)+e\sin(\omega)\big]\sin(\Omega) - \big[\cos(\omega+f)+e\cos(\omega)\big]\cos(\Omega)\cos(i)\Big\} \\
v_z &= \frac{na}{\sqrt{1-e^2}}\Big[\cos(\omega+f)+e\cos(\omega)\Big]\sin(i)
\end{aligned} \tag{2.20}$$

Having these initial conditions, we then integrated the motion of the simulated satellite using a 4th order Runge-Kutta method, firstly defining a function like the following for the dynamical state of the system:

$$f(t,Y) = \begin{pmatrix} Y_{0,1,2} \\ Y_{3,4,5} \end{pmatrix} = \begin{pmatrix} v_{x,y,z} \\ a_{x,y,z} \end{pmatrix} \tag{2.21}$$

And then using this function to calculate the Runge-Kutta coefficients (calling Y_i the "position-and-velocity" vector at the given time):

$$\begin{cases} k_1 = f(t,Y) \\ k_2 = f(t+\frac{h}{2}, Y+\frac{h}{2}k_1) \\ k_3 = f(t+\frac{h}{2}, Y+\frac{h}{2}k_2) \\ k_4 = f(t+h, Y+hk_3) \end{cases} \tag{2.22}$$

And then the next vector Y_{i+1} will be:

$$Y_{i+1} = Y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.23}$$

With i ∈ [0 , t_{max}-h]. As a time step h we chose h = 1 s, especially because it is the first stable value for which the orbit integration is correct.

Chapter 3

Computational Method

To do the project we used a Python extension called **Cython**. This package gives C-like performance with code that is written mostly in Python with optional additional C-inspired syntax.

The biggest advantage of Cython over pure Python is that Cython is a **compiled** language instead of being a **interpreted** one such as Python, while its syntax is almost entirely Python. Indeed this results in the fact that in general a Cython code will be faster to execute than a Python code.

The editor we used to do everything was the **Jupyter Notebook**, for which is possible to import Cython really simply, just by running the so called "Magic Cell" %**load_ext Cython**, this permits to the Notebook to bypass directly the Python interpreter and to directly let the C compiler to do the work.

The use of Cython, specifically its module **Cython.parallel**, permitted the use of the **prange** function instead of a common range.

To use all of this we firstly have to release the **GIL** (Global Interpreter Lock) to tell that we want to use the C compiler instead of the Python interpreter.

The prange function is native parallelize calling the openmp header and has 4 possible ways to parallelize:

- **static**: the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is assigned to each thread in advance;
- **dynamic**: the iterations are distributed to threads as they request them, with a default chunk size of 1;
- **guided**: the iterations are distributed to threads as they request them, but with decreasing chunk size. The size of each chunk is proportional to the number of unassigned iterations divided by the number of participating threads, decreasing to 1;
- **runtime**: the schedule and chunk size are taken from the runtime scheduling variable.

For simplicity, and for not-messing up with threads, we choose to use a *static*-type prange in each for we had.
Moreover, the Cython extension is compatible with the mathematical module **Numpy** of Python, thus permitting a good co-working between Python arrays and Cython.

In the following paragraphers we report the functions we developed and used to do the appropriate task. In advance to this we defined, in each case, a set of four functions for the coefficients (see 2.3, 2.4, 2.5 and 2.6), and a function to better vectorize (for our use) the Stokes Coefficients dataset that takes as inputs the first two columns of the Stokes coefficients file (respectively the order n and the degree m), then it defines an array called INDEX that has the same length as the other two arrays given as inputs and it is filled with zeros. Then, if t is the length of the first column of the file and N and M are the arrays containing the n and m values, remembering that the for cycle is parallelized with the prange:

Algorithm 3.1

```
INDEX ←1-dim array of zeroes
for k ∈ t do
    i ← N[k]
    j ← M[k]
    if i ← j then
        INDEX[k] ← k
    end if
end for
```

The function is then called to obtain an array with the index values that corresponds to when the indexes of the Stokes coefficients n and m are equal in the given file, so that via a simple formula we can easily call every value of the coefficients, effectively calling **C[k - n + m]**, where k is the index at which n is equal to m. This way we can access each value of the given Stokes coefficients.

Those functions are not counted in the total computational time, that is because they are previously calculated and then just remain into memory from where their data can be extracted and used. Moreover, in an ideal real use of the two programs, those function can be run in advance and then just stored into memory.
In any case their calculation often takes under 1 second if the EGM1996 is used to obtain the Stokes Coefficients, if this is not the case and someone wants to use the EGM2008 to do the calculations the time will exceed the 20 seconds, but not because of our functions, but because the computer takes more time to upload the txt file.

Lastly, we always defined an optional Boolean parameter *save*, that if set to **True** saves all the images instead of just plotting them.

3.1 Global Geopotential

To do the calculation for the global geopotential we divided the problem into 6 functions that calculate the following, then the functions was put to run and the resulting potential, potential gradient components, hessian gradient components, gravity anomaly and gravity disturbance were plotted. To see the execution time of each function see tables 4.1, 4.2 and 4.3. Those table contains the resulting times of calculation for the same code written in Python, Cython and parallelized Cython respectively.
Following we present the used functions with a pseudocode following them to better present their functioning:

- **Sine and cosine of mλ**: the function takes as inputs the λ value array and the chosen value of n_{max}.
  We firstly created a 3-dim array with n_{max} rows, n_λ columns (where n_λ is the number of longitude values), and 2 "profundity level", one for the cosine and the other for the sine. This array is firstly filled with zeros and is created with the numpy command *np.zeros(*n_{rows} *,* $n_{columns}$ *,* n_{layers}*)*.
  Then we did two nested loops to fill the array, if LAM is the array with the longitude values, g is its number of values (length) and remembering that each *for-loop* is done with the prange functions so that it is native parallelized using the *static* schedule. The sine and cosine functions are in this case taken from the mathematical library of the C language called *libc.math*:

Algorithm 3.1.1

```
1: CS ←3-dim array of zeroes
2: for i ∈ g do
3:    l ← LAM[i]
4:    for m ∈ n_max do
5:       CS[m, i, 0] ← cos(m ∗ l)
6:       CS[m, i, 1] ← sin(m ∗ l)
7:    end for
8: end for
```

- **The matrix of Normalized Associated Legendre Functions and its derivatives**, see 2.7, 2.8 and 2.9.
  To calculate the nALFs for the geopotential map we defined a for parallel loop on the latitude values and then did a while loop inside of it.
  This did not messed up with the threads because the parallel for loop was set to static.

  The function takes as inputs the chosen value of n_{max}, the array with the values of latitude, and the arrays with the values of the coefficients defined in equations 2.3, 2.4, 2.5, 2.6.

  The returned array is a 4-dim array in this case, always filled with zeros using the numpy command *np.zeros* In this case this huge array has dimension, if g is the number of values in the array of latitudes (length), $n_{max} \times n_{max} \times g \times 3$.

This definition of the array effectively says that we have 3 "matrix cubes" in one single array structure, one for P_n^m, another one for $\frac{dP_n^m}{d\phi}$ and the last one for $\frac{d^2P_n^m}{d\phi^2}$.

From the mathematical description given into equations 2.7, 2.8 and 2.9, if F, G, H and K are the arrays of coefficients and remembering that each *for-loop* is done with the prange functions so that it is native parallelized using the *static* schedule. The sine, cosine and tangent functions are in this case taken from the mathematical library of the C language called *libc.math*:

Algorithm 3.1.2

1: $P \leftarrow$4-dim array of zeroes
2: $P[0,0,:,0] \leftarrow 1$
3: **for** $i \in g$ **do**
4: $c \leftarrow cos(\phi[i])$
5: $s \leftarrow sin(\phi[i])$
6: $t \leftarrow tan(\phi[i])$
7: $P[1,0,i,0] \leftarrow \sqrt{3} * s$
8: $P[1,1,i,0] \leftarrow \sqrt{3} * c$
9: $m \leftarrow 0$
10: $n \leftarrow 2$
11: **while** $n < n_{max}$ **do**
12: $P[n,m,i,0] \leftarrow G[n,m] * P[n-1,m,i,0] * s - H[n,m] * P[n-2,m,i,0]$
13: $P[n,m+1,i,0] \leftarrow G[n,m] * P[n-1,m+1,i,0] * s - H[n,m] * P[n-2,m+1,i,0]$
14: $P[n,m,i,1] \leftarrow K[n,m] * P[n,m+1,i,0] - m * t * P[n,m,i,0]$
15: $P[n,m,i,2] \leftarrow P[n,m,i,0] * ((m/(c*c)) * (m - s*s) - n*(n+1)) + t*K[n,m]*P[n,m+1,i,0]$
16: $m \leftarrow m+1$
17: **if** $m \leftarrow n$ **then**
18: $P[n,n,i,0] \leftarrow F[n] * c * P[n-1,n-1,i,0]$
19: $P[n,n,i,1] \leftarrow -n * t * P[n,n,i,0]$
20: $P[n,n,i,2] \leftarrow n * P[n,n,i,0] * ((n - s*s)/(c*c) - n - 1)$
21: $n \leftarrow n+1$
22: $m \leftarrow 0$
23: **end if**
24: **end while**
25: **end for**

- **The lumped coefficients**, see equation 2.10.
  To calculate the lumped coefficients and their derivatives for the global geopotential we defined a function that features a triple for-loop, one over the latitude values and the other for each value of m (order of the nALF) up to the chosen n_{max} value, and the last one on the value n over the values from M to n_{max}, where M is equal to 2 or 0 depending on the will to use the full-disturbing potential including the C_{20} or to exclude the centrifugal Stokes coefficient.

  This function takes as inputs the chosen value of the radius, the array of latitude values, the chosen value of n_{max}, the two arrays of the Stokes coefficients C_{nm} and S_{nm}, the nALF and its derivatives, the array of indexes ST given by algorithm 3.1 and a Boolean parameter called C_{20}, that permits to choose if to use the full disturbing potential by setting the value of C_{20} (this Stokes coefficient is related to the centrifugal potential) to its actual value, or to zero, if the parameter's value has to be discarded, like in our case, to better visualize the map of the disturbing potential.

  The returned array is, again, a 4 dimensional array filled with zeros using *np.zeros* defined as following $2 \times g \times n_{max} \times 6$. In this case the first dimension differentiates from A_m to B_m, the second term is the dimension given by the number of latitude values, the third is the dimension given by the chosen value of n_{max} and the 4th dimension differentiate between the lumped coefficient, its two first derivative (with respect to r and to ϕ) and its three second derivative.

  From the mathematical description given into the group of equations 2.1 and remembering that each *for-loop* is done with the prange functions so that it is native parallelized using the *static* schedule. Calling $Y = \frac{r_{Earth}}{r}$ and g the length of the array of the altitude values:

Algorithm 3.1.3

if $C20 \leftarrow 0$ **then**
  $C[0] \leftarrow 0$
  $M \leftarrow 2$
else
  $M \leftarrow 0$
end if
$Lump \leftarrow$ 4-dim array of zeroes
for $i \in g$ **do**
  for $m \in n_{max}$ **do**
    $A, Ar, Ap, Arr, App, Arp \leftarrow 0, 0, 0, 0, 0, 0$
    $B, Br, Bp, Brr, Bpp, Brp \leftarrow 0, 0, 0, 0, 0, 0$
    for $n \in [M, ..., n_{max}]$ **do**
      $index \leftarrow int(ST[n-2])$
      $A \leftarrow A + P[n, m, i] * C[index - n + m] * (Y^n)$
      $B \leftarrow B + P[n, m, i] * S[index - n + m] * (Y^n)$
      $Ar \leftarrow Ar + P[n, m, i] * C[index - n + m] * (Y^n) * n$
      $Br \leftarrow Br + P[n, m, i] * S[index - n + m] * (Y^n) * n$
      $Ap \leftarrow Ap + dP[n, m, i] * C[index - n + m] * (Y^n)$
      $Bp \leftarrow Bp + dP[n, m, i] * S[index - n + m] * (Y^n)$
      $Arr \leftarrow Arr + P[n, m, i] * C[index - n + m] * (Y^n) * n * (n + 1)$
      $Brr \leftarrow Brr + P[n, m, i] * S[index - n + m] * (Y^n) * n * (n + 1)$
      $Arp \leftarrow Arp + dP[n, m, i] * C[index - n + m] * (Y^n) * n$
      $Brp \leftarrow Brp + dP[n, m, i] * S[index - n + m] * (Y^n) * n$
      $App \leftarrow App + ddP[n, m, i] * C[index - n + m] * (Y^n)$
      $Bpp \leftarrow Bpp + ddP[n, m, i] * S[index - n + m] * (Y^n)$
    end for
    $Lump[0, i, m, 0] \leftarrow A$
    $Lump[0, i, m, 1] \leftarrow Ar$
    $Lump[0, i, m, 2] \leftarrow Ap$
    $Lump[0, i, m, 3] \leftarrow Arr$
    $Lump[0, i, m, 4] \leftarrow Arp$
    $Lump[0, i, m, 5] \leftarrow App$
    $Lump[0, i, m, 1] \leftarrow B$
    $Lump[1, i, m, 1] \leftarrow Br$
    $Lump[1, i, m, 2] \leftarrow Bp$
    $Lump[1, i, m, 3] \leftarrow Brr$
    $Lump[1, i, m, 4] \leftarrow Brp$
    $Lump[1, i, m, 5] \leftarrow Bpp$
  end for
end for

- **The potential**, see equation 2.11.
  Finally, to calculate the geopotential value at each (λ,ϕ) coupled values, we have written a function that takes as inputs the chosen r value, the two arrays of longitude and latitude values, the two arrays that contains the cos(mλ) and the sin(mλ) and the two arrays that contains the lumped coefficients A_m and B_m.

  As always, the potential is expressed in the likewise of a 2-dimensional array, with number of rows equal to the number of latitude values and number of columns equal to the number of longitude value, and it was firstly expressed as a matrix of zeros that is then filled with the apposite values in each place.

  Now, calling J the term $\frac{GM}{r}$, g1 the length of the longitude array, g2 the length of the altitude array, COS and SIN the arrays of the values of cos(mλ) and sin(mλ) and A and B the arrays of the lumped coefficients A_m and B_m, remembering that each for loop is done using the prange function so that it is natively parallelized using the static schedule:

Algorithm 3.1.4

```
Pot ←2-dim array of zeros
for i ∈ g1 do
  for j ∈ g2 do
    Cor ← 0
    for m ∈ n_max do
      Cor = Cor + A[i,m] * COS[m,j] + B[i,m] * SIN[m,j]
    end for
    Pot[i,j] ← J * Cor
  end for
end for
```

- **The gradient of the potential**, see equation 2.12.
  To calculate the gradient values of the geopotential at each (λ,ϕ) coupled values, we have written a function that takes as inputs the chosen r value, the two arrays of longitude and latitude values, the two arrays that contains the cos(mλ) and the sin(mλ), the two arrays that contains the lumped coefficients A_m and B_m and the four first derivative of the lumped coefficients $\partial_r A_m$, $\partial_r B_m$, $\partial_\phi B_m$ and $\partial_\phi B_m$.

  The gradient of the potential is expressed in the likewise of a 3-dimensional array, with number of rows equal to the number of latitude values, number of columns equal to the number of longitude values and number of layers equal to 3 (the three derivatives ∂_r, ∂_ϕ and ∂_λ), and it was firstly expressed as a matrix of zeros that is then filled with the apposite values in each place.

  Now, calling J the term $\frac{GM}{r}$, g1 the length of the longitude array, g2 the length of the altitude array, COS and SIN the arrays of the values of cos(mλ) and sin(mλ) and A and B the arrays of the lumped coefficients A_m and B_m, remembering that each for loop is done using the prange function so that it is natively parallelized using the static schedule:

Algorithm 3.1.5

$dPot \leftarrow$3-dim array of zeros
for $i \in g1$ **do**
  for $j \in g2$ **do**
    $rCor \leftarrow 0$
    $pCor \leftarrow 0$
    $lCor \leftarrow 0$
    for $m \in n_{max}$ **do**
      $a \leftarrow COS[m, j]$
      $bSIN[m, j]$
      $rCor = rCor + a * (Ar[i, m] - (A[i, m]/r)) + b * (Br[i, m] - (B[i, m]/r))$
      $phiCor = pCor + a * Ap[i, m] + b * Bp[i, m]$
      $lambdaCor = lCor + m * (-b * A[i, m] + a * B[i, m])$
    end for
    $Pot[i, j, 0] \leftarrow J * rCor$
    $Pot[i, j, 1] \leftarrow J * pCor$
    $Pot[i, j, 2] \leftarrow J * lCor$
  end for
end for

- **The hessian of the potential**, see equation 2.13.

  To calculate the hessian values of the geopotential at each (λ,ϕ) coupled values, we have written a function that takes as inputs the chosen r value, the two arrays of longitude and latitude values, the two arrays that contains the cos(mλ) and the sin(mλ), the two arrays that contains the lumped coefficients A_m and B_m, the four first derivative of the lumped coefficients $\partial_r A_m$, $\partial_r B_m$, $\partial_\phi B_m$ and $\partial_\phi B_m$ and the six second derivatives of the lumped coefficients $\partial^2_{rr} A_m$, $\partial^2_{rr} B_m$, $\partial^2_{\phi\phi} A_m$, $\partial^2_{\phi\phi} B_m$, $\partial^2_{r\phi} A_m$ and $\partial^2_{r\phi} B_m$.

  The Hessian of the potential is expressed in the likewise of a 3-dimensional array, with number of rows equal to the number of latitude values, number of columns equal to the number of longitude values and number of layers equal to 6 (the six derivatives ∂^2_{rr}, $\partial^2_{\phi\phi}$, $\partial^2_{\lambda\lambda}$, $\partial^2_{r\phi}$, $\partial^2_{r\lambda}$ and $\partial^2_{\phi\lambda}$), and it was firstly expressed as a matrix of zeros that is then filled with the apposite values in each place.

  Now, calling J the term $\frac{GM}{r}$, g1 the length of the longitude array, g2 the length of the altitude array, COS and SIN the arrays of the values of cos(mλ) and sin(mλ) and A and B the arrays of the lumped coefficients A_m and B_m, remembering that each for loop is done using the prange function so that it is natively parallelized using the static schedule:

Algorithm 3.1.6

$ddPot \leftarrow$3-dim array of zeros
for $i \in g1$ **do**
  for $j \in g2$ **do**
    $rrCor \leftarrow 0$
    $ppCor \leftarrow 0$
    $llCor \leftarrow 0$
    $rpCor \leftarrow 0$
    $rlCor \leftarrow 0$
    $plCor \leftarrow 0$
    for $m \in n_{max}$ **do**
      $a \leftarrow COS[m, j]$
      $b \leftarrow SIN[m, j]$
      $rrCor \leftarrow rrCor + a * (Arr[i, m] - 2 * Ar[i, m] + 2 * (A[i, m]/r)) + b * (Brr[i, m] - 2 * Br[i, m] + 2 * (B[i, m]/r))$
      $ppCor \leftarrow ppCor + a * App[i, m] + b * Bpp[i, m]$
      $llCor \leftarrow llCor - m * m * (a * A[i, m] + b * B[i, m])$
      $rpCor \leftarrow rpCor + a * (Arp[i, m] - (Ap[i, m]/r)) + b * (Brp[i, m] - (Bp[i, m]/r))$
      $rlCor \leftarrow rlCor + m * (-b * (Ar[i, m] - (A[i, m]/r)) + a * (Br[i, m] - (B[i, m]/r)))$
      $plCor \leftarrow plCor + m * (-b * Ap[i, m] + alpha * Bp[i, m])$
    end for
    $ddPot[i, j, 0] \leftarrow J * rrCor/r$
    $ddPot[i, j, 1] \leftarrow J * ppCor$
    $ddPot[i, j, 2] \leftarrow J * llCor$
    $ddPot[i, j, 3] \leftarrow J * rpCor$
    $ddPot[i, j, 4] \leftarrow J * rlCor$
    $ddPot[i, j, 5] \leftarrow J * plCor$
  end for
end for

- **Gravity anomaly and gravity disturbance**: as a bonus, we calculated also the gravity anomaly (Δg) and the gravity disturbance (δg) defined as following:

$$\begin{cases} \Delta g = -\frac{2V}{r} - \frac{\partial V}{\partial r} \\ \\ \delta g = -\frac{\partial V}{\partial r} \end{cases} \tag{3.1}$$

The gravity disturbance is the difference between measured gravity at a point and the normal gravity at the same point, while the gravity anomaly is the difference between the observed gravity at a point, and the normal gravity on the geoid, the point where the normal to the spheroid at that point intersects the geoid.

To calculate these two quantities we designed the following function that takes as input the chosen radius r, the array of latitude values phi, the array of longitude values lam, the 2-dimensional array of the potential obtained through algorithm 3.1.5 Pot, and the 2-dimensional array of the radial derivative of the potential rPot. We calculate the quantities in the unit of measure mGal, this is why in the following algorithm there is a multiplication by a factor of 100.
Now, if g1 and g2 are, respectively, the lengths of the arrays of latitude and longitude:

Algorithm 3.1.7

$dg \leftarrow$3-dim array of zeros
for $i \leftarrow g1$ **do**
  for $j \in g2$ **do**
    $dg[i, j, 0] \leftarrow -100 * (2 * Pot[i, j]/r - rPot[i, j])$
    $dg[i, j, 1] \leftarrow -100 * rPot[i, j]$
  end for
end for

3.2 Orbit Integration

To do the calculation for the orbital integration we divided the problem into 11 functions that calculate:

- **Sine and cosine of mλ**.
  To calculate the sine and cosine of mλ at each projected point of the orbit we developed a function that takes as inputs the λ value (called l in the function) and the chosen value of n_{max}.

  The sine and cosine value are shaped into a 2 dimensional array. This array is firstly filled with zeros and is created with the numpy command *np.zeros((n_{rows} , $n_{columns}$))*. The array is then filled with the appropriate values into the parallelized for loop:

Algorithm 3.2.1

$CS \leftarrow$2-dim array of zeros
for $m \in n_{max}$ **do**
  $h \leftarrow m * l$
  $CS[m, 0] \leftarrow cos(h)$
  $CS[m, 1] \leftarrow sin(h)$
end for

- **The matrix of Normalized Associated Legendre Functions**. To calculate the nALFs for the orbital integration we defined a for parallel loop on the m indexes and then did another parallel for loop on the n indexes inside of it. This was done to avoid numerical instability (row-wise computation).

  The function we designed takes as inputs the chosen n_{max}, the latitude value ϕ (called p in the function)and the arrays F, G and H that contains the coefficients f, g and h.
  Like before, the nALF are expressed in a 2-dimensional array created with the np.zeros function, in this case with dimension $n_{max} \times n_{max}$:

Algorithm 3.2.2

$P \leftarrow$2-dim array of zeros
c $\leftarrow cos(p) s \leftarrow sin(p)$
$P[0, 0] = 1$
$P[1, 0] = \sqrt{3} * s$
$P[1, 1] = \sqrt{3} * c$
for $m \in n_{max}$ **do**
  for $n \in [m + 1, ..., n_{max}]$ **do**
    $P[n, n] \leftarrow F[n] * c * P[n - 1, n - 1]$
    $P[n, m] \leftarrow G[n, m] * s * P[n - 1, m] - H[n, m] * P[n - 2, m]$
  end for
end for

- **The matrix of the prime derivative of the Normalized Associated Legendre Functions**. This was calculated with the same method as the previous. The derivative of the

matrix of the nALF obviously has the same dimension of the matrix of the nALF, and it is initialized in the same way.

The function we designed to calculate it takes as input the chosen value of n_{max}, the value of latitude ϕ (called p in the following pseudocode), the matrix of the nALF calculated using the algorithm 3.2.4, called P and the matrix containing the k coefficient (eq. 2.6). As always the for loops present in the pseudocode are parallel for loop parallelized using the prange function in the static schedule:

Algorithm 3.2.3

$dP \leftarrow$2-dim array of zeros
$t \leftarrow tan(phi)$
for $m \in n_{max}$ **do**
  for $n \in m + 1$ **do**
    $P[n, n] \leftarrow -n * t * P[n, n]$
    $P[n, m] \leftarrow K[n, m] * P[n, m + 1] - m * t * P[n, m]$
  end for
end for

We divided the calculation of the nALF and its derivative because, using the double for loop instead of the method used in algorithm 3.1.2 revealed in some errors, due to the parallelization, in the calcualtion of the derivative of nALFs.

- **The lumped coefficients**, see equations 2.10.
  To calculate the lumped coefficient we defined a function that takes as input the chosen value of radius and n_{max}, the arrays with the Stokes coefficients C and S, the matrix of the nALF and the array of indexes ST given by algorithm 3.1:

Algorithm 3.2.4

$Lump \leftarrow$ 2-dim array of zeros
for $m \in n_{max}$ **do**
  $A \leftarrow 0$
  $B \leftarrow 0$
  for $n \in m + 1$ **do**
    $index \leftarrow int(N[n] - 2)$
    $A \leftarrow A + P[n, m] * C[index - n + m] * (Y^n)$
    $B \leftarrow B + P[n, m] * S[index - n + m] * (Y^n)$
  end for
  $Lump[0, m] \leftarrow A$
  $Lump[1, m] \leftarrow B$
end for

- **The prime derivatives of the lumped coefficients**, see equations 2.10.
  To calculate the first derivatives of the lumped coefficient we defined a function that takes as input the chosen value of radius and n_{max}, the arrays with the Stokes coefficients C and

S, the matrix of the nALF, the matrix of the first derivatives of the nALF and the array of indexes ST given by algorithm 3.1:

Algorithm 3.2.5

$dLump \leftarrow$ 2-dim array of zeros
for $m \in n_{max}$ **do**
  $Ar \leftarrow 0$
  $Br \leftarrow 0$
  $Ap \leftarrow 0$
  $Bp \leftarrow 0$
  for $n \in m + 1$ **do**
    $index \leftarrow int(N[n] - 2)$
    $Ar \leftarrow Ar + P[n, m] * C[index - n + m] * (Y^n) * (n + 1)$
    $Br \leftarrow Br + P[n, m] * S[index - n + m] * (Y^n) * (n + 1)$
    $Ap \leftarrow Ap + dP[n, m] * C[index - n + m] * (Y^n)$
    $Bp \leftarrow Bp + dP[n, m] * S[index - n + m] * (Y^n)$
  end for
  $Lump[0, m] \leftarrow A$
  $Lump[1, m] \leftarrow B$
  $Lump[2, m] \leftarrow A$
  $Lump[3, m] \leftarrow B$
end for

- **The potential**, see equation 2.11.
  For completeness, we wrote the function for the potential, in this case the function takes as input the r value at which the satellilte is from the center of Earth, the chosen value of n_max, the arrays that are output of algorithm 3.2.1 taht contains the sine and cosine of $m\lambda$ and the two lumped coefficient that are output of algorithm 3.2.4:

Algorithm 3.2.6

$Cor \leftarrow 0$
for $m \in n_{max}$ **do**
  $Cor \leftarrow Cor + A[m] * COS[m] + B[m] * SIN[m]$
end for
$Pot \leftarrow J * Cor$

- **The gradient of the potential**, see equation 2.12.
  Similarly to algorithm 3.2.6, we wrote the function for the calculation of the gradient of the potential in spherical coordinates.
  We firstly defined the gradient as a 1-dimensional array filled with zeros (using np.zeros as always) that is then filled with the appropriate values in a single and parallelized for loop. the function we designed takes as inputs the same objects as the funciton in algorithm 3.2.6 but with the adjoint of 4 inputs that are the derivatives of the lumped coefficients calculated using algorithm 3.2.5:

Algorithm 3.2.7

$dPot \leftarrow$ 1-dim array of zeros
$rCor \leftarrow 0$
$pCor \leftarrow 0$
$lCor \leftarrow 0$
for $m \in n_{max}$ **do**
  $a \leftarrow COS[m]$
  $b \leftarrow SIN[m]$
  $rCor \leftarrow rCor + a * Ar[m] + b * Br[m]$
  $pCor \leftarrow pCor + a * Ap[m] + b * Bp[m]$
  $lCor \leftarrow lCor + m * (b * A[m] - a * B[m]$
end for
$dPot[0] \leftarrow -J * rCor/r$
$dPot[1] \leftarrow J * pCor$
$dPot[2] \leftarrow -J * lCor$

- **The gradient of potential in Cartesian coordinates** see equation 2.14.
  To change the coordinates of the gradient to plot it like a ground track, we defined a simple function that takes as inputs the gradient in spherical coordinates and the values of the coordinates r, ϕ and λ:

Algorithm 3.2.8

$dPot_cart \leftarrow$ 1-dim array of zeros
$dPot_cart[0] \leftarrow -(sin(l)/(r*cos(p)))*dPot[2]-sin(p)*cos(l)*dPot[1]/r+cos(p)*cos(l)*dPot[0]$
$dPot_cart[1] \leftarrow cos(l) * dPot[2]/(r * cos(p)) - sin(p) * sin(l) * dPot[1]/r + cos(p) * sin(l) * dPot[0]$
$dPot_cart[2] \leftarrow cos(p) * dPot[1]/r + sin(p) * dPot[0]$

- **Coordinate transformation between Cartesian and latitude-longitude**, see equations 2.16 and 2.17. We designed this function that takes as inputs the three cartesian coordinates (x,y,z), the chosen time step h in seconds, the longitude of ascending node Ω of the satellite at t = t_0, the time t, in seconds, from t_0, and the Greenwhich sidereal mean time called alphaGW0 that is almost 100 degrees:

Algorithm 3.2.9

$LL \leftarrow$ 1-dim array of zeros
$LL[0] \leftarrow arctan2(z, \sqrt{x^2 + y^2})$
$LL[1] \leftarrow arctan2(y, x) - 0.004165403 * t - alphaGW0 + Omega$
if $LL[1] < -180$ **then**
  $LL[1] \leftarrow LL[1] + 360$
end if
if $LL[1] > 180$ **then**
  $LL[1] \leftarrow LL[1] - 360$
end if

- **A function for the dynamical evolution of the system** see equation 2.15 for the acceleration component.
  At last, before numerical integration, we designed the function for the dynamical evolution that takes as input the time t, the vector of initial conditions IC and the gradient of the potential calculated at time t.
  This function was designed in Python and not in Cython to be able to use the output as an object and not a pure array:

Algorithm 3.2.10

$r \leftarrow \sqrt{IC[0] * IC[0] + IC[1] * IC[1] + IC[2] * IC[2]}$
$f \leftarrow$ 1-dim array of zeros
$f[0] \leftarrow IC[3]$
$f[1] \leftarrow IC[4]$
$f[2] \leftarrow IC[5]$
$f[3] \leftarrow nablaPot[0] - (GM/(r * r * r)) * IC[0]$
$f[4] \leftarrow nablaPot[1] - (GM/(r * r * r)) * IC[1]$
$f[5] \leftarrow nablaPot[2] - (GM/(r * r * r)) * IC[2]$

- **A 4th order Runge-Kutta method for integration**, this function takes into it all the previous functions (except for the calculation of the potential) to perform the calculation step by step.
  See equation 2.22 and 2.23.
  The function takes as inputs the time step in seconds h, the initial time t0, the final time tfin, the set of initial condition y0, the chosen value of n_{max}, the rrays of the coefficients F, G, H and K, the array os fthe Stokes coefficients C and S, the array of the indexes given by algorithm 3.1 and the value of the longitude of the ascending node Omega.

  We did not write this in Cython because we wanted to have multiple outputs and, to avoid shutdowns of the kernel due to the tto high request of memory, we wrote it in Python, and the outputs are the collection of cartesian coordinate per unit of time, called y, the collection of the λ values per unit of time of the satellite called LAMdeg, because they are expressed in degrees, the collection of the ϕ values per unit of time of the satellite called PHIdeg, always because of their expression in degrees, and at last the array of time, from t0 to tfin with step h:

Algorithm 3.2.11

$t \leftarrow$ 1-dim array of times, from t0 to tfin with step h
$y \leftarrow$ 2-dim array of positions
$y[0] \leftarrow y0$
$LAM \leftarrow$ 1-dim array of zeros
$PHI \leftarrow$ 1-dim array of zeros
$r \leftarrow \sqrt{y[0,0] * y[0,0] + y[0,1] * y[0,1] + y[0,2] * y[0,2]}$
$T \leftarrow$length of t minus one
for $i \in T$ **do**
$CS \leftarrow Algorithm3.2.1(l, n_{max})$
$P \leftarrow Algorithm3.2.2(n_{max}, p, F, G, H)$
$dP \leftarrow Algorithm3.2.3(n_{max}, p, P, K)$
$Lcoeffs \leftarrow Algorithm3.2.4(r, n_{max}, C, S, N, P)$
$dLcoeffs \leftarrow Algorithm3.2.5(r, n_{max}, C, S, N, P, dP)$
$DPot \leftarrow Algorithm3.2.7(r, n_max, CS[:,0], CS[:,1], Lcoeffs[0,:], Lcoeffs[1,:], dLcoeffs[0,:]$
$], dLcoeffs[1,:], dLcoeffs[2,:], dLcoeffs[3,:])$
$DPotcart \leftarrow Algorithm3.2.8(DPot, l, p, r)$
$T, Y \leftarrow t[i], y[i]$
$k1 \leftarrow Algorithm3.2.10(T, Y, DPotcart)$
$k2 \leftarrow Algorithm3.2.10(T + 0.5 * h, Y + 0.5 * h * k1, DPotcart)$
$k3 \leftarrow Algorithm3.2.10(T + 0.5 * h, Y + 0.5 * h * k2, DPotcart)$
$k4 \leftarrow Algorithm3.2.10(T + h, Y + h * k3, DPotcart)$
$y[i+1] \leftarrow Y + (k1 + 2 * k2 + 2 * k3 + k4)/6$
$r \leftarrow \sqrt{y[i+1,0] * y[i+1,0] + y[i+1,1] * y[i+1,1] + y[i+1,2] * y[i+1,2]}$
$LLnew \leftarrow Algorithm3.2.9(y[i+1,0], y[i+1,1], y[i+1,2], h, Omega, T, 100)$
$p \leftarrow LLnew[0]$
$l \leftarrow LLnew[1]$
$LAM[i+1] \leftarrow l$
$PHI[i+1] \leftarrow p$
end for

The execution time are visible in the tables 4.4 and 4.5 that contains, respectively the execution time for the pure Cython code and the parallelized Cython code. The Python code was not included in this case because of its enormous computational time or this problem.
To obtain a precise result with our program it is better to use a time step h of *1 s* or less.

Chapter 4

Final Results

Following we present the visual results of the two codes we wrote and the execution times for different maximum chosen order n_{max}.
The choice to use Cython to parallelize Python code was rewarded with exceptional time gains. For the orbit integration, Python was neither taken into consideration, having exceeded an execution time of more than a day in some cases.

The tables 4.1, 4.2 and 4.3 are relative to the global geopotential, while the tables 4.4 and 4.5 are relative to the orbital integration.

From figures 4.3 and 4.4 is easy to see that the option to parallelize the orbit integration code is not a time gaining option at low values of n_{max}, especially regarding the calculation of the nALF and its derivative. On the other hand we can see that for the most time requiring fucntions, the lumped coefficients and their derivatives, the parallelization seems to give a good time gain (see figures 4.5 and 4.6).

We did not put other plots for the time gain because, as one can see from tables 4.4 and 4.5, the other functions have a random-kind of time gain level, maybe due to the fact that for these function (i.e. algorithm 3.2.1) we neither used a double for loop or a nested for-while loop, thus effectivley looping on only one variable, shortening time by semplicity. In every case, from figure 4.2 it is easy to see that the parallelization is, globally, a really good option to speed up the code.

For the global geopotential we can easily see from figure 4.1 that the Python code is always worse in time gain than the Cython one, as one can expect, but the Cython code is not always slower than the parallelized Cython code, indeed it seems that for $n_{max} < 200$, it is less computationally costly to do the global geopotential map not parallelizing the code. But, in every case, at high enough values of n_{max} the parallelization is always a better time gaining option.

The resulting plots are shown in the appendix.

Table 4.1: Execution times given in seconds for the Parallelized Cython code for each function defined to calculate the global geopotential. The precision for the latitude and longitude is of 1 degree squared.

$\mathbf{n_{max}}$	100	200	360	500	1000	1500	2000
cos(mλ) and sin(mλ)	0.0014	0.0013	0.0024	0.0029	0.0038	0.0084	0.0076
nALFs	0.0174	0.0635	0.1752	0.3501	1.4138	3.5917	7.7508
Lumped Coeffs.	0.0190	0.1691	0.5696	1.0830	4.3646	11.7753	604.5792
V	0.0121	0.0140	0.0297	0.0382	0.0792	0.1464	0.3466
∇(V)	0.2481	0.2573	0.2576	0.2278	0.2634	0.4614	0.8296
H(V)	0.2557	0.2550	0.2212	0.2894	0.5229	0.9441	1.5030
Δg and δg	0.0012	0.0014	0.0008	0.0014	0.0014	0.0012	0.0127
Total Time	0.5549	0.7616	1.2565	1.9928	6.6491	16.9285	615.0295

Table 4.2: Execution times given in seconds for the pure Cython code for each function defined to calculate the global geopotential. The precision for the latitude and longitude is of 1 degree squared.

$\mathbf{n_{max}}$	100	200	360	500	1000	1500	2000
cos(mλ) and sin(mλ)	0.0007	0.0015	0.0025	0.0042	0.0062	0.0093	0.0222
nALFs	0.0292	0.0982	0.3257	0.6279	2.5974	8.5287	20.6744
Lumped Coeffs.	0.0598	0.4452	1.8118	3.6305	15.0997	36.8380	7543.394
V	0.0205	0.0397	0.0684	0.1157	0.2284	0.3700	0.7719
∇(V)	0.0395	0.0896	0.1641	0.2378	0.4634	0.7753	1.3490
H(V)	0.0887	0.1925	0.3345	0.4931	1.0217	1.6860	2.6807
Δg and δg	0.0006	0.0007	0.0006	0.0007	0.0007	0.0009	0.0009
Total Time	0.2390	0.8674	2.7076	5.1099	19.4175	48.2082	7568.8931

Table 4.3: Execution times given in seconds for the pure Python code for each function defined to calculate the global geopotential. The precision for the latitude and longitude is of 1 degree squared.

$\mathbf{n_{max}}$	100	200	360	500	1000	1500	2000
cos(mλ) and sin(mλ)	0.2980	0.2168	0.3877	0.5312	1.0346	1.4847	1.8019
nALFs	6.3636	25.2822	84.0452	153.8971	611.1911	1406.4911	2166.7249
Lumped Coeffs.	22.1491	85.4376	288.3869	524.6560	2167.0900	4578.5502	15567.4344
V	6.6765	13.6546	26.0904	34.3549	65.8921	94.0601	125.1426
∇(V)	24.4430	48.7042	89.0543	118.8121	231.8200	314.9503	421.0932
H(V)	63.4916	125.2731	230.9612	310.8630	588.5861	832.3681	1089.1536
Δg and δg	0.1654	0.1606	0.1592	0.1528	0.1503	0.1315	0.1316
Total Time	123.5872	298.7291	719.0785	1143.2671	3665.7642	7228.0360	19371.4822

Table 4.4: Execution times given in seconds for the parallelized Cython code for each function defined to calculate the orbital integration. The last row is the time for the total integration taking the time step h = 1 s. For the sake of readability, the total times are reported with two decimal numbers.

$\mathbf{n_{max}}$	70	100	120	240	360	500	720	1000	1500	2000
cos(mλ) and sin(mλ)	0.0002	0.0003	0.0003	0.0003	0.0002	0.0003	0.0004	0.0004	0.0019	0.0004
nALFs	0.0002	0.0003	0.0003	0.0006	0.0011	0.0022	0.0032	0.0075	0.0159	0.0249
deriv. nALF	0.0005	0.0002	0.0004	0.0005	0.0009	0.0018	0.0031	0.0067	0.0161	0.0252
Lumped Coeffs.	0.0002	0.0003	0.0003	0.0009	0.0027	0.0065	0.0147	0.0187	0.0487	0.0704
deriv. Lump Coeffs.	0.0001	0.0003	0.0003	0.0009	0.0025	0.0071	0.0095	0.0191	0.0352	0.0701
∇(V)	0.0002	0.0002	0.0001	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002
∇(Vcart)	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
Total Time	34.67	50.63	62.35	133.60	394.42	945.40	2223.66	4561.52	12064.31	19694.18

Table 4.5: Execution times given in seconds for the pure Cython code for each function defined to calculate the orbital integration. The last row is the time for the total integration taking the time step h = 1 s. For the sake of readability, the total times are reported with two decimal numbers.

$\mathbf{n_{max}}$	70	100	120	240	360	500	720	1000	1500	2000
cos(mλ) and sin(mλ)	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
nALFs	0.0001	0.0003	0.0003	0.0005	0.0025	0.0024	0.0044	0.0096	0.0247	0.0403
deriv. nALF	0.0001	0.0001	0.0005	0.0005	0.0013	0.0021	0.0085	0.0111	0.0348	0.0429
Lumped Coeffs.	0.0002	0.0005	0.0007	0.0017	0.0064	0.0134	0.0381	0.0526	0.1137	0.1993
deriv. Lump Coeffs.	0.0002	0.0004	0.0007	0.0018	0.0070	0.0116	0.0250	0.0532	0.1221	0.2307
∇(V)	0.0001	0.0001	0.0001	0.0001	0.0003	0.0001	0.0001	0.0001	0.0001	0.0001
∇(Vcart)	0.0002	0.0000	0.0001	0.0001	0.0001	0.0000	0.0001	0.0001	0.0001	0.0001
Total Time	41.77	74.11	100.43	385.01	1056.09	2443.00	5331.60	11176.91	26580.77	47051.35

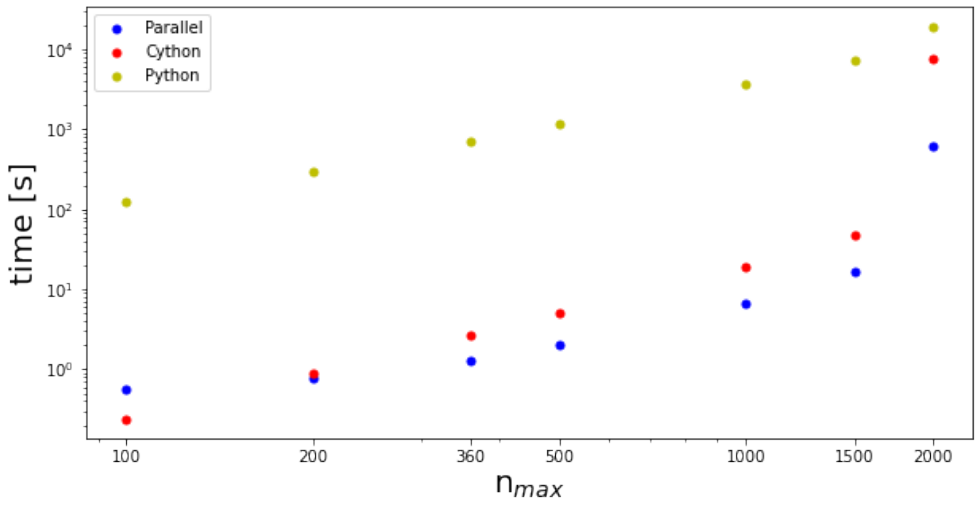

Figure 4.1: Plot of the total computational time for the global geopotential. The two axis are logarithmic.

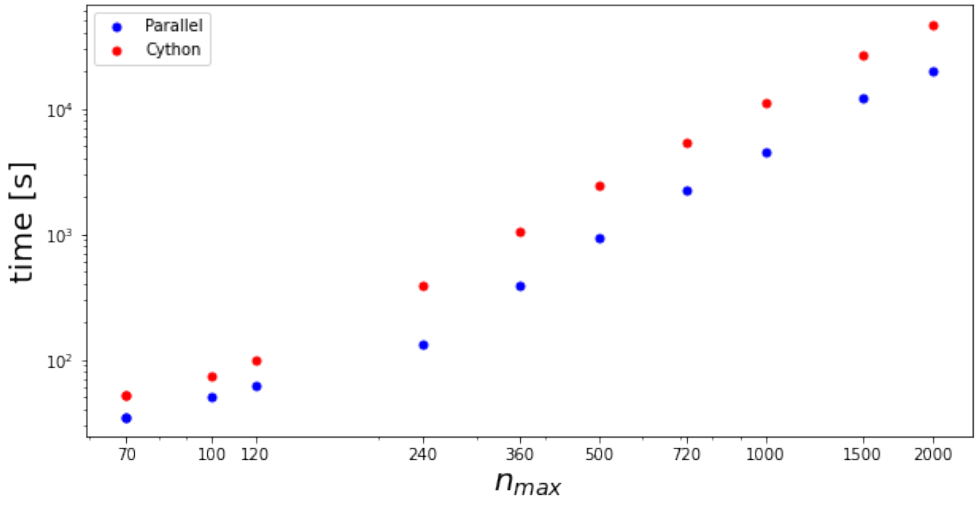

Figure 4.2: Plot of the total computational time for the ground track of the satellite. The two axis are logarithmic.

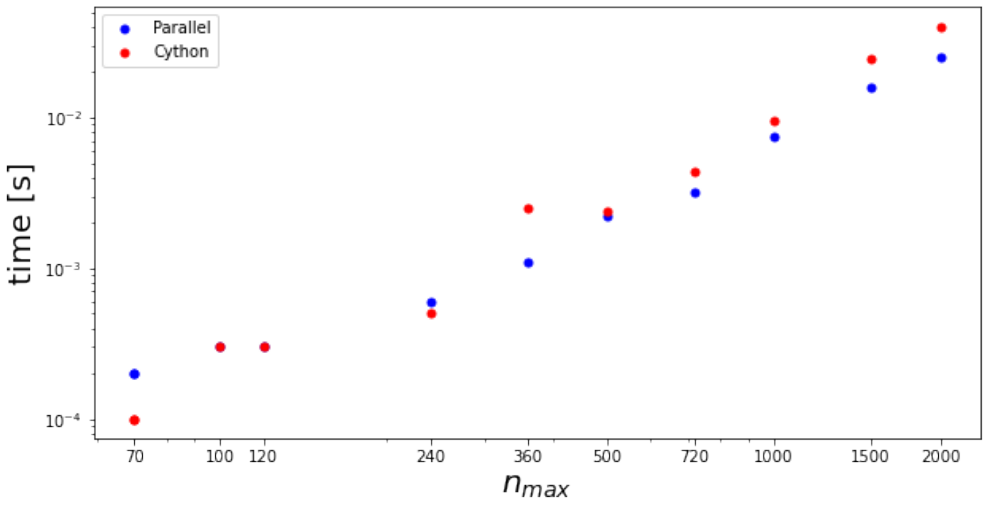

Figure 4.3: Detail of the calculation of the nALF for the ground track. The two axis are logarithmic. The resolution taken into exam is of 1 degree squared (a grid of 360×180.

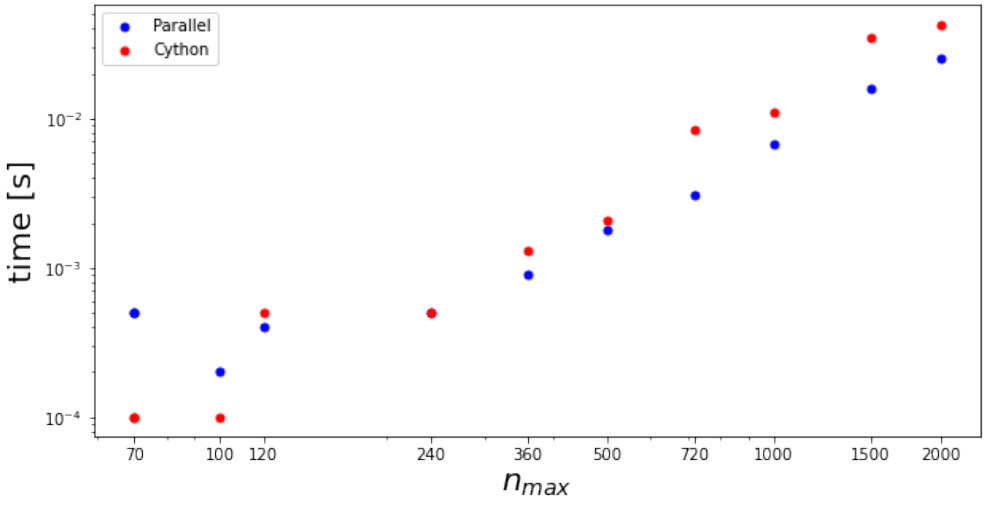

Figure 4.4: Detail of the calculation of the derivative of the nALF for the ground track. The two axis are logarithmic.

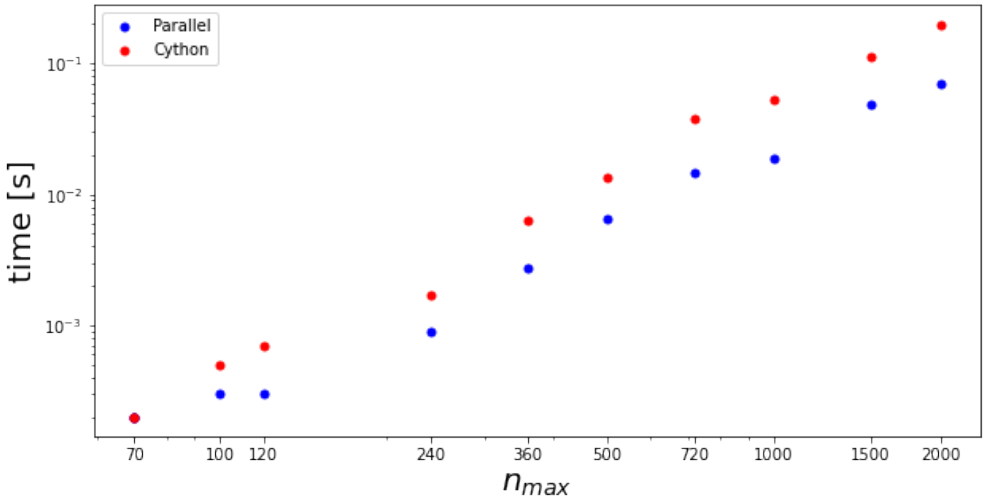

Figure 4.5: Detail of the calculation of the lumped coefficients for the ground track. The two axis are logarithmic.

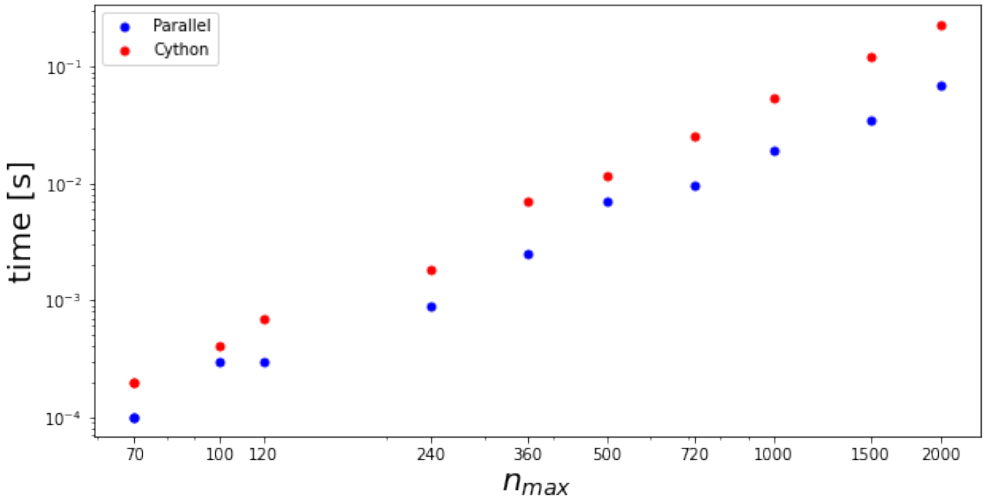

Figure 4.6: Detail of the calculation of the derivative of the lumped coefficients for the ground track. The two axis are logarithmic.

Appendix A

Global geopotential maps

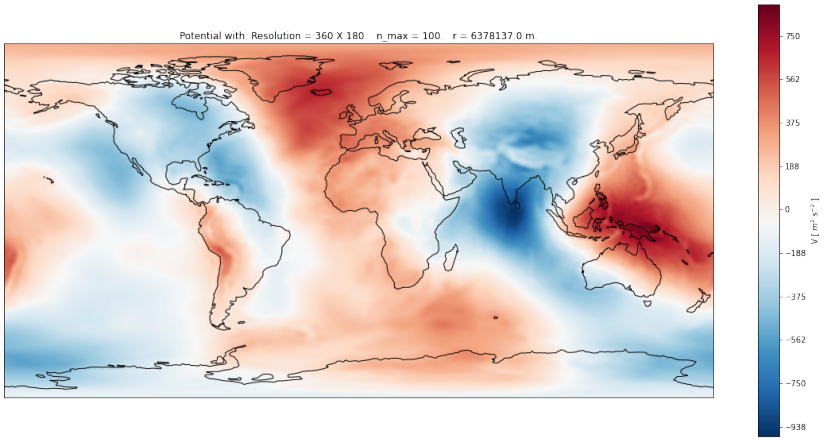

Figure A.1: Color map of the geopotential with $n_{max} = 100$ and with resolution $360^\circ \times 180^\circ$.

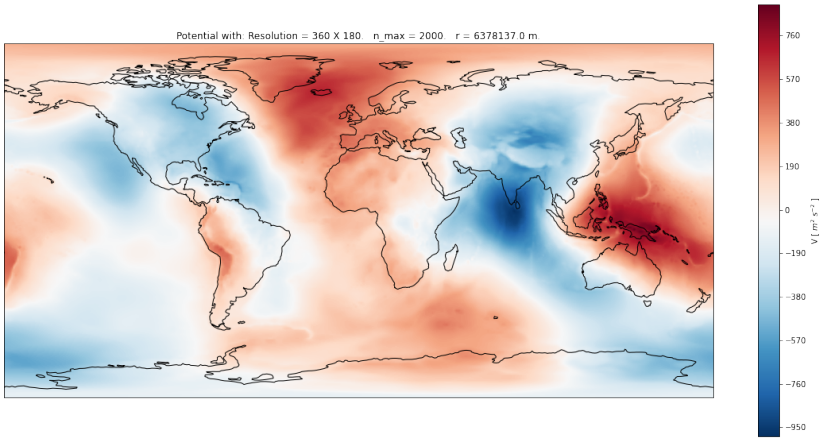

Figure A.2: Color map of the geopotential with $n_{max} = 2000$ and with resolution $360^\circ \times 180^\circ$.

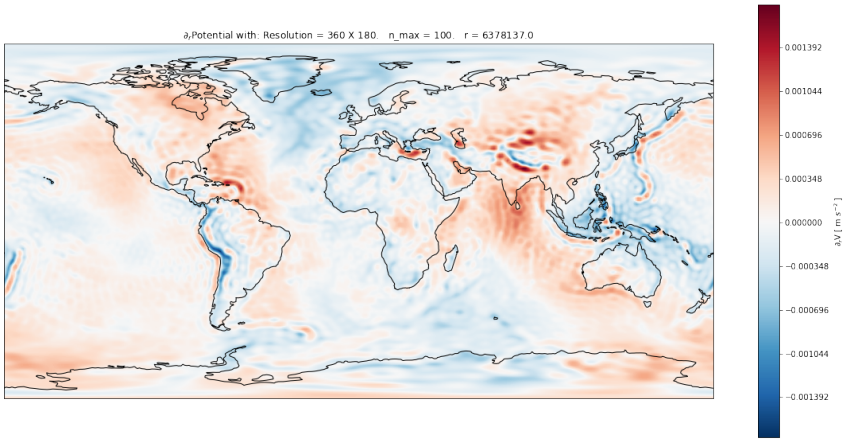

Figure A.3: Color map of the radial derivative geopotential with $n_{max} = 100$ and with resolution $360° \times 180°$.

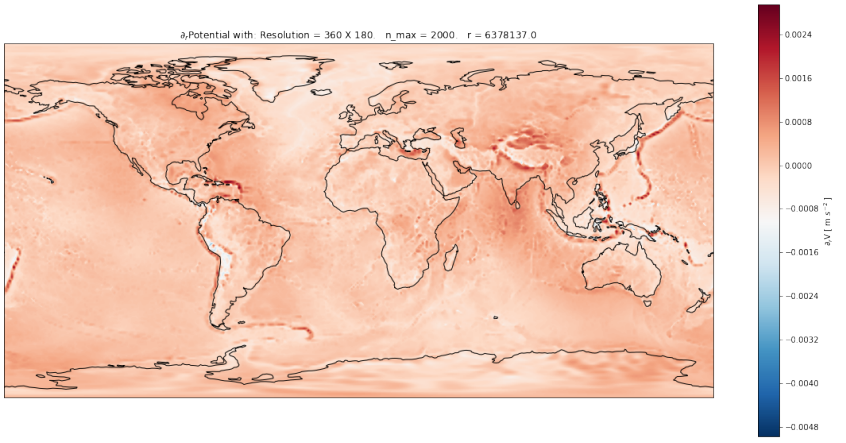

Figure A.4: Color map of the radial derivative geopotential with $n_{max} = 2000$ and with resolution $360° \times 180°$.

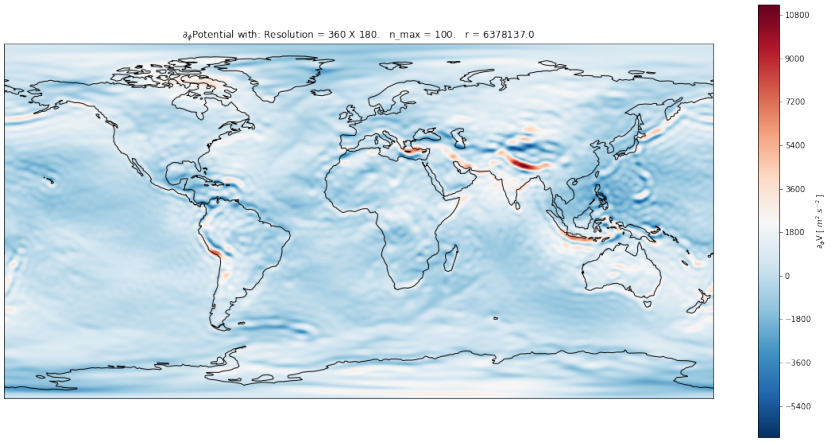

Figure A.5: Color map of the latitudinal derivative geopotential with $n_{max} = 100$ and with resolution $360° \times 180°$.

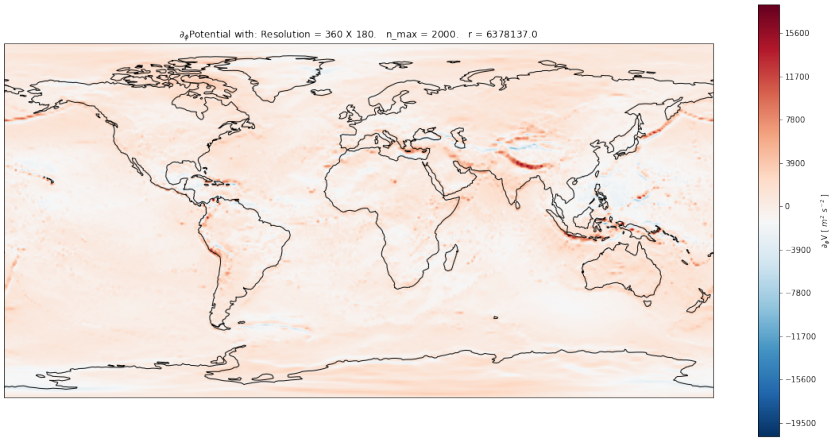

Figure A.6: Color map of the latitudinal derivative geopotential with $n_{max} = 2000$ and with resolution $360° \times 180°$.

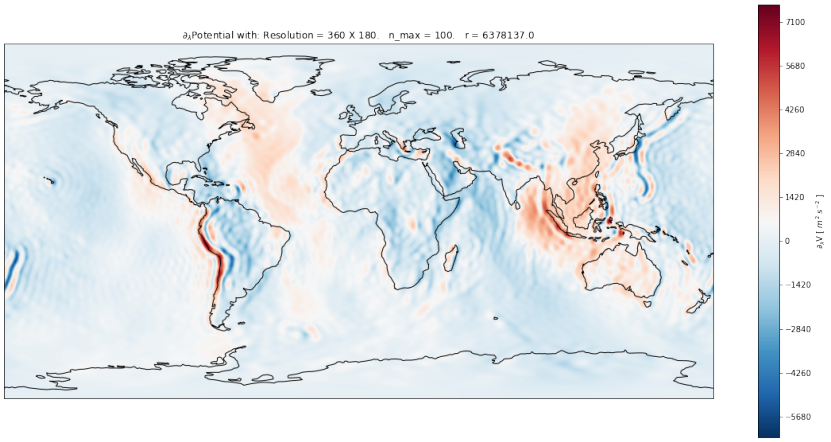

Figure A.7: Color map of the longitudinal derivative geopotential with $n_{max} = 100$ and with resolution $360° \times 180°$.

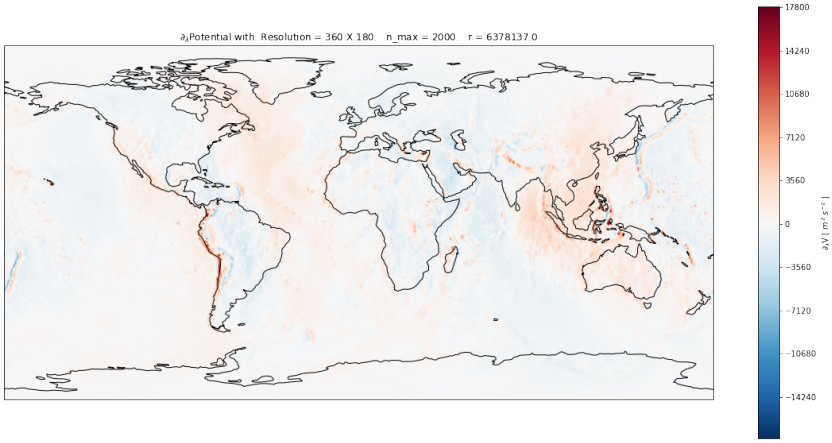

Figure A.8: Color map of the longitudinal derivative geopotential with $n_{max} = 2000$ and with resolution $360° \times 180°$.

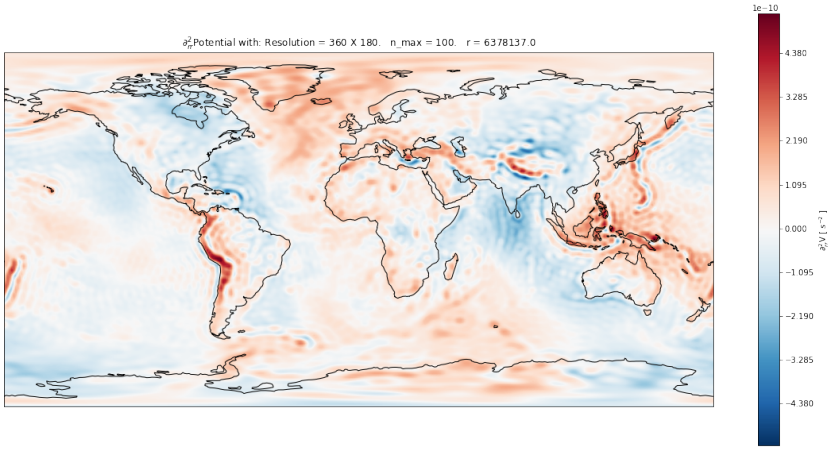

Figure A.9: Color map of the ∂_{rr}^2 of the geopotential $n_{max} = 100$ and with resolution $360° \times 180°$.

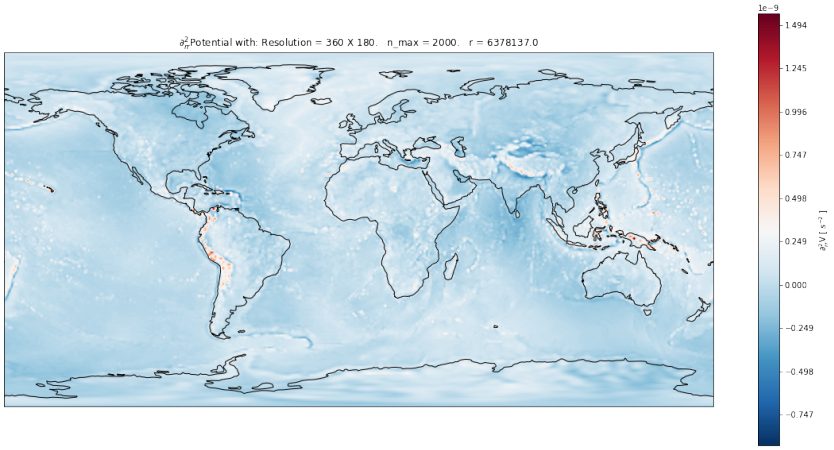

Figure A.10: Color map of the ∂_{rr}^2 of the geopotential $n_{max} = 2000$ and with resolution $360° \times 180°$.

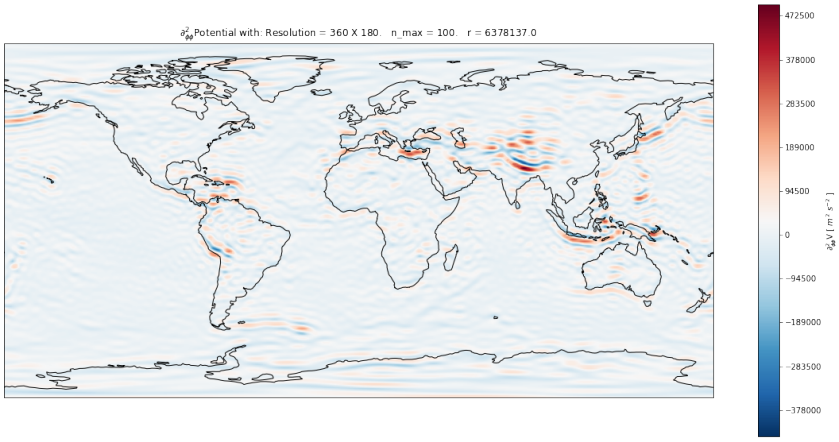

Figure A.11: Color map of the $\partial^2_{\phi\phi}$ of the geopotential $\mathrm{n}_{max} = 100$ and with resolution $360° \times 180°$.

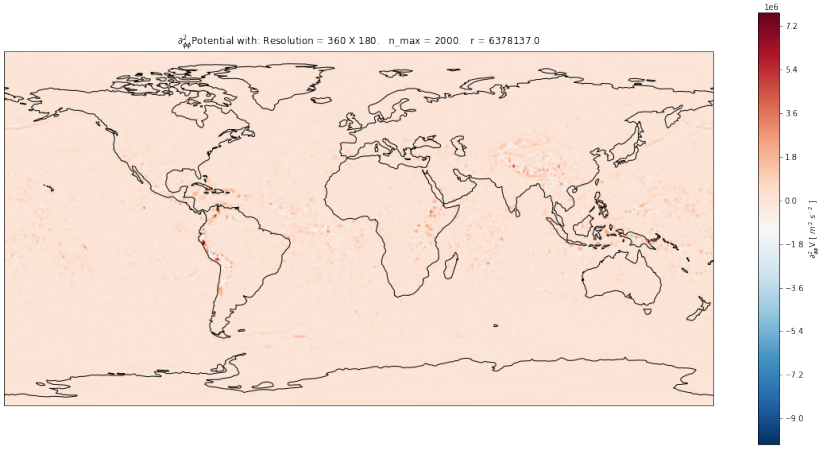

Figure A.12: Color map of the $\partial^2_{\phi\phi}$ of the geopotential $\mathrm{n}_{max} = 2000$ and with resolution $360° \times 180°$.

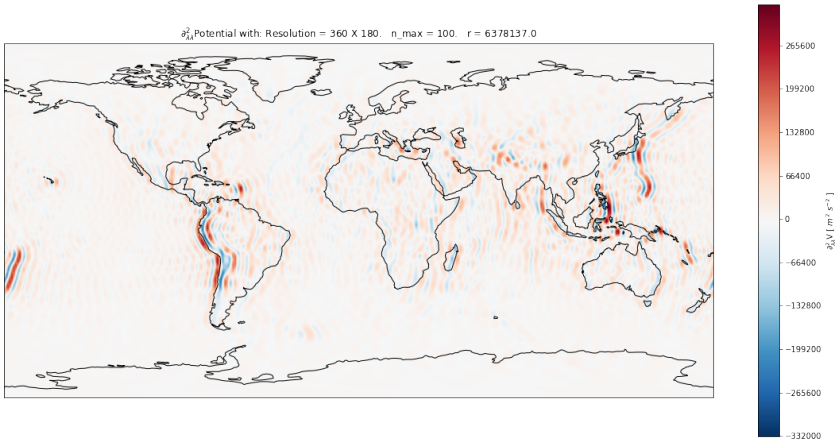

Figure A.13: Color map of the $\partial^2_{\lambda\lambda}$ of the geopotential $\mathrm{n}_{max} = 100$ and with resolution $360° \times 180°$.

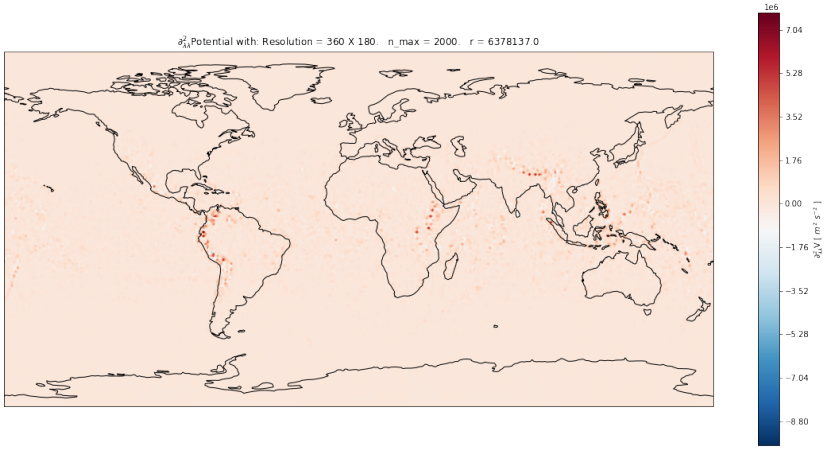

Figure A.14: Color map of the $\partial^2_{\lambda\lambda}$ of the geopotential $\mathrm{n}_{max} = 2000$ and with resolution $360° \times 180°$.

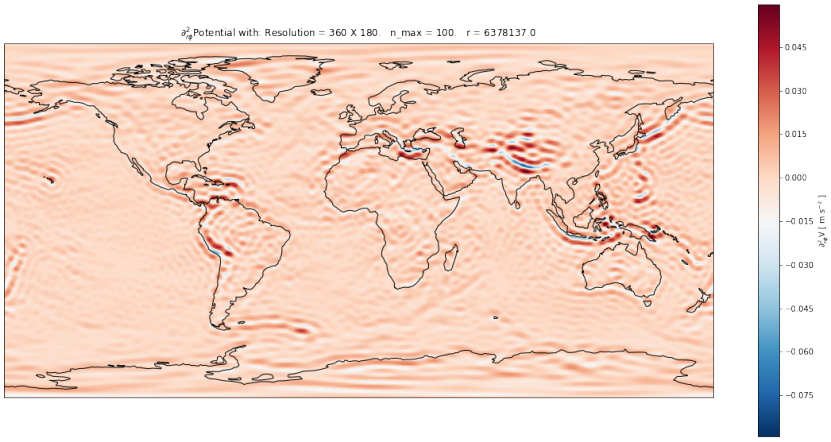

Figure A.15: Color map of the $\partial^2_{r\phi}$ of the geopotential $\mathrm{n}_{max} = 100$ and with resolution $360° \times 180°$.

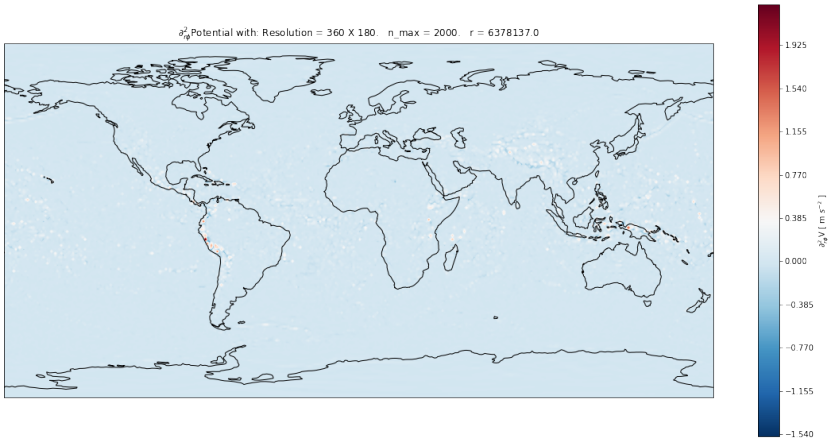

Figure A.16: Color map of the $\partial^2_{r\phi}$ of the geopotential $\mathrm{n}_{max} = 2000$ and with resolution $360° \times 180°$.

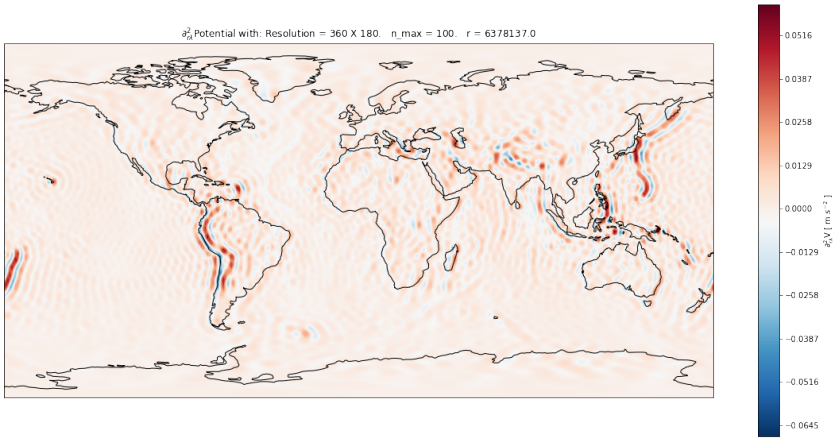

Figure A.17: Color map of the $\partial^2_{r\lambda}$ of the geopotential $\mathrm{n}_{max} = 100$ and with resolution $360° \times 180°$.

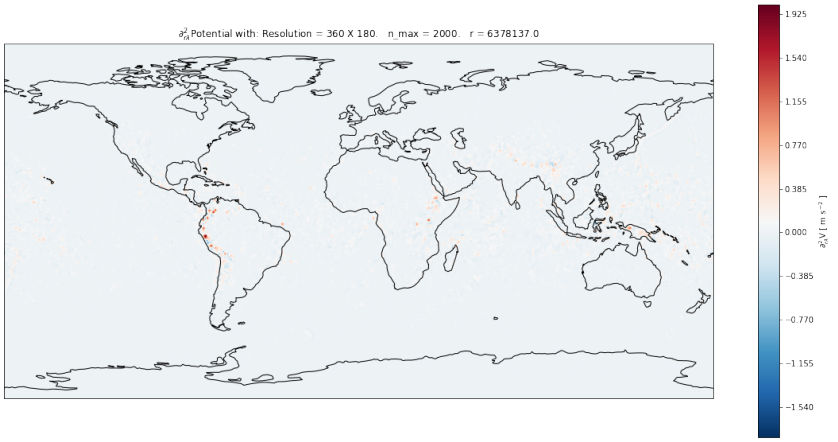

Figure A.18: Color map of the $\partial^2_{r\lambda}$ of the geopotential $\mathrm{n}_{max} = 2000$ and with resolution $360° \times 180°$.

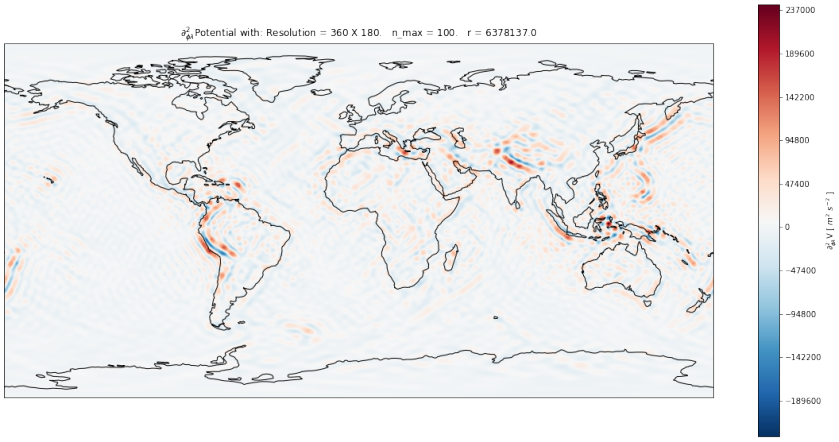

Figure A.19: Color map of the $\partial^2_{\phi\lambda}$ of the geopotential $\mathrm{n}_{max} = 100$ and with resolution $360° \times 180°$.

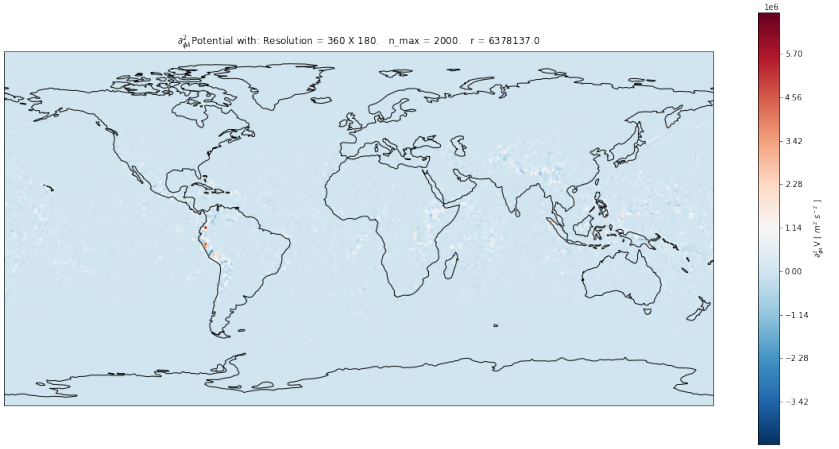

Figure A.20: Color map of the $\partial^2_{\phi\lambda}$ of the geopotential $\mathrm{n}_{max} = 2000$ and with resolution $360° \times 180°$.

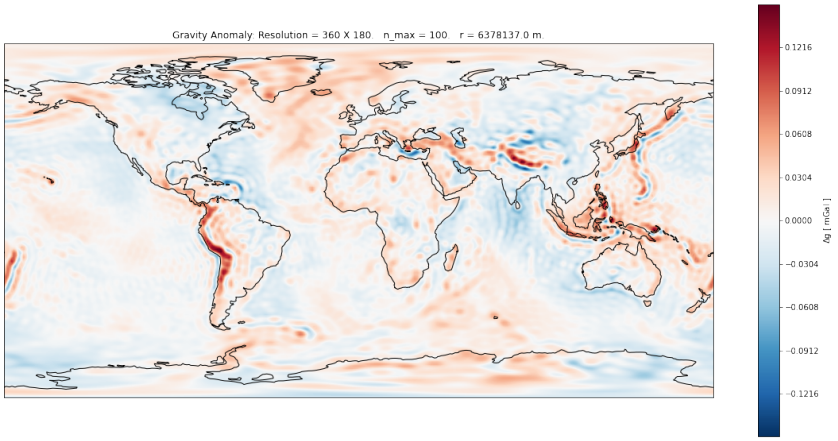

Figure A.21: Color map of the gravity anomaly with $n_{max} = 100$ and with resolution $360° \times 180°$.

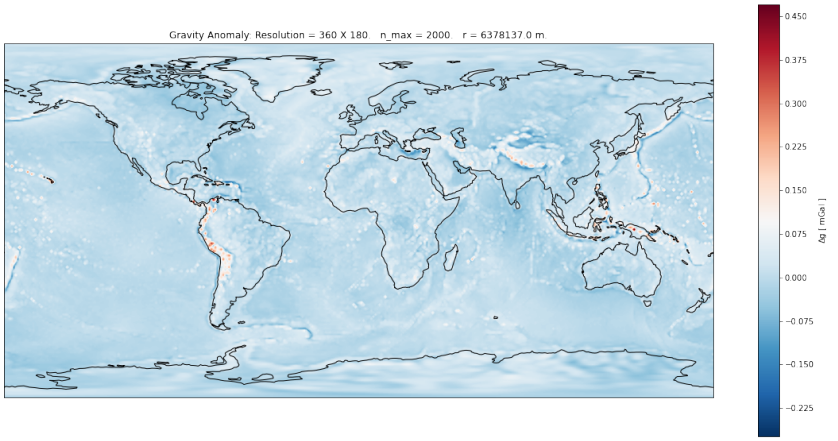

Figure A.22: Color map of the gravity anomaly with $n_{max} = 2000$ and with resolution $360° \times 180°$.

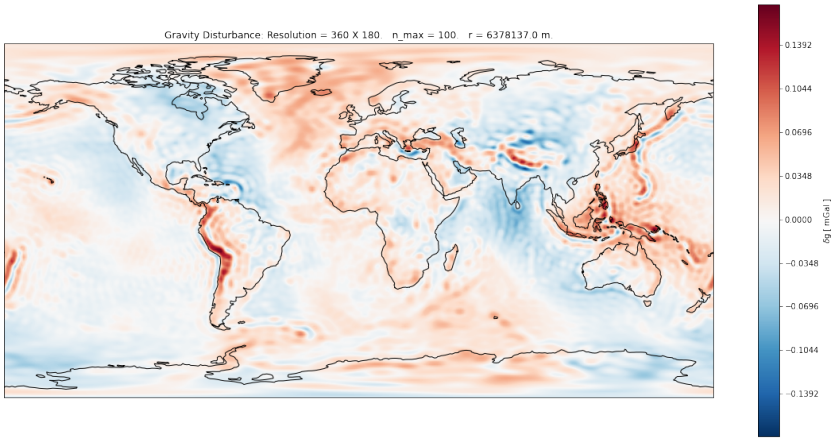

Figure A.23: Color map of the gravity disturbance with $n_{max} = 100$ and with resolution $360° \times 180°$.

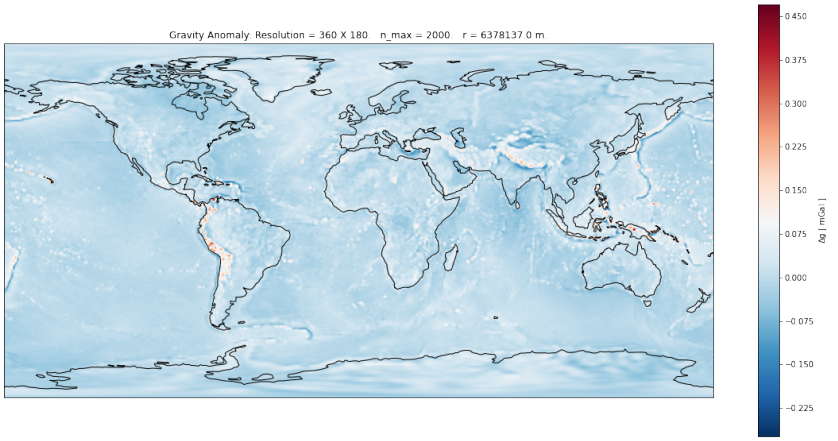

Figure A.24: Color map of the gravity disturbance with $n_{max} = 2000$ and with resolution $360° \times 180°$.

Appendix B

Ground track maps

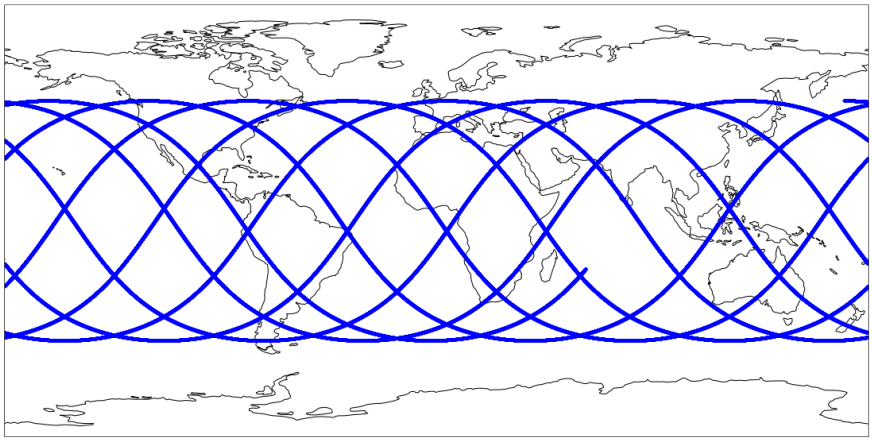

Figure B.1: Neary Circular Orbit plot with initial conditions: $\mathbf{i = 50°}$, $\mathbf{e = 0.02}$, $\mathbf{a = 10^7 m}$, $\mathbf{\Omega = 60°}$, $\boldsymbol{\omega} \mathbf{= 90°}$, $\mathbf{f_0 = 0°}$.

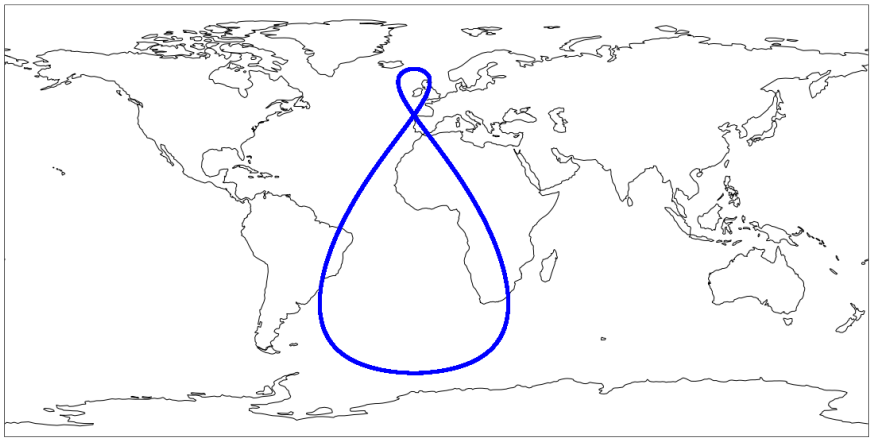

Figure B.2: Tundra Orbit plot with initial conditions: $\mathbf{i = 63.4°}$, $\mathbf{e = 0.3}$, $\mathbf{a = 4.2164{\cdot}10^7 m}$, $\mathbf{\Omega = 60°}$, $\boldsymbol{\omega} \mathbf{= 270°}$, $\mathbf{f_0 = 0°}$.

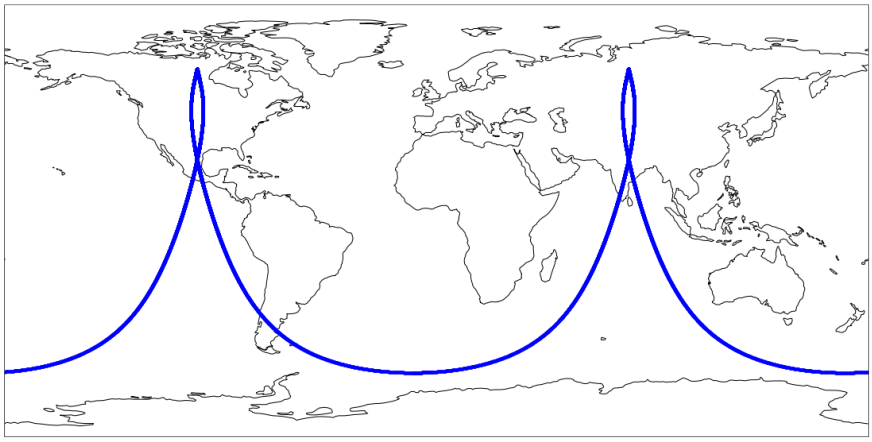

Figure B.3: Molniya Orbit plot with initial conditions: $\mathbf{i = 63.4°}$, $\mathbf{e = 0.74}$, $\mathbf{a = 2.66{\cdot}10^7 m}$, $\mathbf{\Omega = 60°}$, $\boldsymbol{\omega} \mathbf{= 270°}$, $\mathbf{f_0 = 0°}$.