SL$_{C0de}$ Documentation

Release 0.4.0

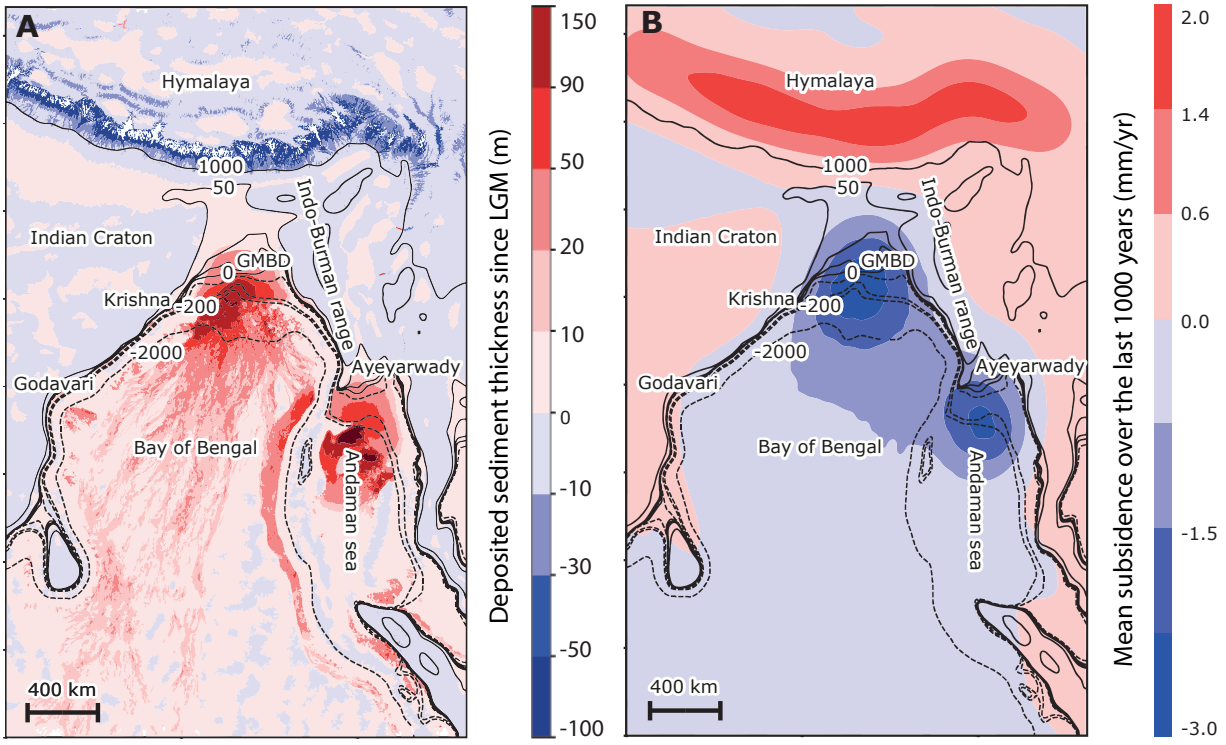
Adrien Henry

Nov 15, 2023

Contents:

CHAPTER 1

SL_{C0de}



Note: This project is under active development.

SL_{C0de} is a python library based on the theory described in [Dalca *et al.*, 2013]. This module provides tools to resolve the Sea Level Equation (SLE). This theory incorporates the governing equations, shoreline migration due to local sea level variation and changes in the geometry of grounded/marine based ice. This theory is based on Love numbers theory, which include gravitational, deformational and rotational effects of the sediment redistribution.

1.1 Usage

1.1.1 INSTALLATION

Note: The code is developped on windows for the moment. We want to export it on other OS but for the moment we focus on the packages.

Download

To install the library, run in command line :

```
(.venv) $ conda create -n SL_C0de
(.venv) $ conda activate SL_C0de
(.venv) $ conda install pip
(.venv) $ pip install --index-url https://test.pypi.org/simple/ --no-deps slcode
```

To install the dependencies, download the requirements.txt on https://github.com/AHenryLarroze/Py_SL_C0de and run inside the file :

```
(.venv) $ pip install -r requirements
```

You also need the stripy package. This package has unstable deployment. Try to install this package with pip and conda.

Note: You need to be in the activated environment to install the packages locally

For testing the module. You can download the script file in https://github.com/AHenryLarroze/Py_SL_C0de. You will also need cartopy to install it.

```
(.venv) $ conda install -c conda-forge cartopy
```

1.1.2 SL_{C0de} minimal exemple

Here we propose to run the code over 26 kyr with a time resolution of 500 years. The spatial resolution is deduced from the maximum spherical harmonic degree (see section xx). Here 64 degree correspond to a spatial resolution of 170 km at the equator.

```
import numpy as np

stop=26 #stop the computation at 26 kyr
step=0.5 # run the model at a time resolution of 0.5
time_step=np.arange(start=stop,stop=-step,step=-step)
maxdeg=64 #Define the maximum degrre of spherical harmonics
```

We preset the ice, sediment and topographic time grid.

```
from SL_C0de.grid import ICE_TIME_GRID
from SL_C0de.grid import SEDIMENT_TIME_GRID
from SL_C0de.grid import TOPOGRAPHIC_TIME_GRID
```

(continues on next page)

(continued from previous page)

```
ice_time_grid=ICE_TIME_GRID(time_step,maxdeg,grid_name='ice')
sed_time_grid=SEDIMENT_TIME_GRID(time_step,maxdeg,grid_name='sed')
topo_time_grid=TOPOGRAPHIC_TIME_GRID(time_step,maxdeg,grid_name='topo')
```

For this example we will set no ice load, with the method zeros_time. topography have a general depth of 3000 m. We create a disk centered at the equator with a thickness of 100 m, a radius of 300 km and deposited after 500 years.

```
sed_time_grid.disk_time(self,1,0,180,1.5,100):
ice_time_grid.zeros_time(self,np.arange(0,ice_time_grid.time_step_number,1))
topo_time_grid.zeros_time(self,np.arange(0,topo_time_grid.time_step_number,1))
topo_time_grid.height_time_grid-=3000
topo_time_grid.topo_pres=topo_time_grid.height_time_grid[-1,:,:]
topo_time_grid.topo_initial=topo_time_grid.height_time_grid[0,:,:]
```

Now all entries are set up, we can resolve the sea level equation. We set as entry love numbers based on VM5 [Peltier *et al.*, 2015] with a litosphere of 60 km, a upper mantle viscosity of $10^{20.5}$ Pa.s and a lower mantle viscosity of $10^{22.699}$ Pa.s. We use a forward modeling version of the SLE resolution. We set a convergence criterion of the SLE to 10^{-10}

```
from SL_C0de.SOLVER import SLE_forward_modeling

love_file="VM5a.l60.um20.5.lm22.699"
love_way="input_data/love/"
conv_lim=10^(-10)

ocean_time_grid,ice_time_grid_model,topo_time_grid_model=SLE_forward_modeling(ice_time_
↪grid,sed_time_grid,topo_time_grid,love_way,love_file,conv_lim)
```

From here, we can post process these data to compute subsidence linked to the different loads. First you must save your result into a file.

```
import os
import sys

Output_way="outputs/"

if not(os.path.exists(Output_way+love_file)):
    os.mkdir(Output_way+love_file)
ocean_time_grid.timecoefftotimegrd()
ocean_time_grid.save(Output_way+love_file)
ice_time_grid_model.save(Output_way+love_file)
topo_time_grid_model.save(Output_way+love_file)
sediment_time_grid.save(Output_way+love_file)
```

Now we apply the post process

```
from SL_C0de.SOLVER import Post_process

Input_way_sed=Output_way+love_file
Input_way_model_output=Output_way
Post_process(Input_way_sed,Input_way_model_output,love_way):
```

Post process create new files in a folder called LOAD in the output folder. These data can be plotted using different

functions.

```
import numpy as np

from SL_C0de.grid import LOAD_TIME_GRID
import os

sediment_color=(0.4,0.7,0.5)
sediment_color_dark=(0.2,0.4,0.25)
ice_color=(0.1,0.8,0.8)
ocean_color=(0.2,0.2,0.6)

def extract_local_map(way,shot,res,time):
    load=LOAD_TIME_GRID(from_file=(True,way+"/LOAD/SEDIMENT_LOAD_122_512"))
    load.height_time_coeff=load.viscuous_deformation+load.elastic_deformation
    t_it=np.where(load.time_step==time)[0][0]
    load.coeff=(load.coeff_from_step(t_it-2).coeff-load.coeff_from_step(t_it-3).coeff)/
↪(load.time_step[t_it-1]-load.time_step[t_it])
    grid,lon_hd,lat_hd=load.coefftogrdhd(res)
    lon_lim_min=np.abs(lon_hd-shot[0]).argmin()
    lon_lim_max=np.abs(lon_hd-shot[1]).argmin()
    lat_lim_min=np.abs(lat_hd-shot[2]).argmin()
    lat_lim_max=np.abs(lat_hd-shot[3]).argmin()
    lon=lon_hd[lon_lim_min:lon_lim_max+1]
    lat=lat_hd[lat_lim_min:lat_lim_max+1]
    grid_sediment=grid[lat_lim_min:lat_lim_max+1,lon_lim_min:lon_lim_max+1]


    #import ocean_load
    load=LOAD_TIME_GRID(from_file=(True,way+"/LOAD/OCEAN_LOAD_122_512"))
    load.height_time_coeff=load.viscuous_deformation+load.elastic_deformation
    load.coeff=(load.coeff_from_step(t_it-2).coeff-load.coeff_from_step(t_it-3).coeff)/
↪(load.time_step[t_it-1]-load.time_step[t_it])
    grid,lon_hd,lat_hd=load.coefftogrdhd(res)
    grid_ocean=grid[lat_lim_min:lat_lim_max+1,lon_lim_min:lon_lim_max+1]

    #import ice_load
    load=LOAD_TIME_GRID(from_file=(True,way+"/LOAD/ICE_LOAD_122_512"))
    load.height_time_coeff=load.viscuous_deformation+load.elastic_deformation
    load.coeff=(load.coeff_from_step(t_it-2).coeff-load.coeff_from_step(t_it-3).coeff)/
↪(load.time_step[t_it-1]-load.time_step[t_it])
    grid,lon_hd,lat_hd=load.coefftogrdhd(res)
    grid_ice=grid[lat_lim_min:lat_lim_max+1,lon_lim_min:lon_lim_max+1]

    #import oceanic_sediment_load
    load=LOAD_TIME_GRID(from_file=(True,way+"/LOAD/OCEANIC_SEDIMENT_LOAD_122_512"))
    load.height_time_coeff=load.viscuous_deformation+load.elastic_deformation
    load.coeff=(load.coeff_from_step(t_it-2).coeff-load.coeff_from_step(t_it-3).coeff)/
↪(load.time_step[t_it-1]-load.time_step[t_it])
    grid,lon_hd,lat_hd=load.coefftogrdhd(res)
    grid_oceanic_sediment=grid[lat_lim_min:lat_lim_max+1,lon_lim_min:lon_lim_max+1]

    return (grid_sediment, grid_ocean, grid_ice, grid_oceanic_sediment), lon,lat
```

```
import os
Input_way="output/"
earth_model_name_list=os.listdir(Input_way)
area =(150,210,-30,30)
L=[]
res=1024
time=0
for earth_model_name in earth_model_name_list:
    print(f'preparing map for earth model : {earth_model_name}')
    out=extract_local_map(Input_way+earth_model_name,area,res,time)
    L.append(out[0])
    lat=out[1]
    lon=out[2]
```

```
import cartopy
import cartopy.crs as ccrs
import matplotlib
from matplotlib import cm
import matplotlib.colors as colors
import numpy as np
import matplotlib.pyplot as plt


font = {'family' : 'Times New Roman',
        'weight' : 'normal',
        'size'   : 12}

matplotlib.rc('font', **font)

def mapplot(lon,lat,grid,area,name,sup_name,save_way=None,add_contour=None,vmin=None,
↪vmax=None,contour=None,color=None):
    alpha_ocean=0
    coast_line_width=0.5
    cmap=cm.get_cmap('bwr', 100)
    if not(color is None):
        cmap=None

    fig = plt.figure(figsize=(4,4), facecolor="none")

    norm = colors.TwoSlopeNorm(vmin=vmin,vmax=vmax,vcenter=0)
    ax1  = plt.subplot(111, projection=ccrs.PlateCarree())
    ax1.set_extent(area)
    m2 = ax1.contourf(lat,lon,grid,levels=contour,origin='lower', transform=ccrs.
↪PlateCarree(),extent=area, zorder=0, cmap=cmap,colors=color, interpolation="gaussian",
↪norm=norm)
    if not(add_contour is None):
        CS=ax1.contour(lat,lon,add_contour,5,colors='k',linewidths=0.5,linestyles='solid
↪')
        ax1.clabel(CS, CS.levels, inline=True, fontsize=10)

    cbar2=plt.colorbar(mappable=m2, orientation="horizontal", shrink=0.5)
    cbar2.set_label(f'Total_subsidence (m) fro {name}')
```

(continues on next page)

(continued from previous page)

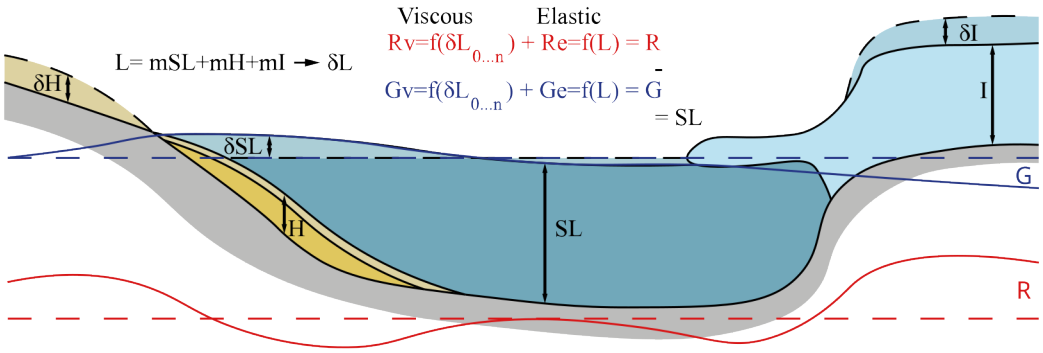
```
    if not(contour is None):
        cbar2.set_ticks(contour)
    elif not(vmax is None) and not(vmin is None):
        cbar2.set_ticks([vmin, 0, vmax])
    elif not(vmin is None):
        cbar2.set_ticks([vmin, 0, grid.max()])
    elif not(vmax is None):
        cbar2.set_ticks([grid.min(), 0, vmax])
    else:
        cbar2.set_ticks([grid.min(), 0, grid.max()])
    cartopy.mpl.geoaxes.GeoAxes.gridlines(ax1,crs=ccrs.PlateCarree(),draw_labels=True)
    ax1.add_feature(cartopy.feature.OCEAN, alpha=alpha_ocean, zorder=99, facecolor="
↪#BBBBBB")
    ax1.coastlines(resolution="50m", zorder=100, linewidth=coast_line_width)

    if not(save_way is None) :
        fig.savefig(f"{save_way}/{sup_name}_subsidence_map_at_{time}_model_{name}.pdf")
```

Based on the output we can calculate the total subsidence induced by the cumulated load of ocean, and sediment (no ice here)

```
grid=np.array(L)
grid=grid[0]+grid[1]+grid[2]
#grid=grid[1]+grid[2]+grid[3]
color_for_final=[[0,0,1],[0.5,0.5,1],[0.7,0.7,1],[0.5,0.7,0.7],[0.5,0.7,0.5],[0.6,0.7,0.
↪2],[0.8,0.6,0.6],[1,0,0]]
#contours=[-3,-2,-1,-0.5,-0.2,-0.1,0,0.1,0.2] # Ayeyarwady
contours=[-1,-0.5,-0.4,-0.3,-0.2,-0.1,0,1,3] # Gange
mapplot(lon,lat,grid,area,'l32_AYS1','min_max_Total',vmin=-3,vmax=0.2,save_way=Input_way,
↪contour=contours,color=color_for_final)
plt.show()
```

1.2 Mathematical Theory

Most of our code is based on the mathematic and computational theory from [Dalca *et al.*, 2013].



1.2.1 The SLE

The Relative sea level (ΔSL) variations is the result of the interaction between the vertical mouvement of geoïd, $\Delta Gtot$ (surface of the ocean) and the vertical mouvement of the ground, $\Delta Rtot$ (called subsidence). We can then express the equation as :

$$\Delta SL(\Theta, \Psi, t) = \Delta Gtot(\Theta, \Psi, t) - \Delta Rtot(\Theta, \Psi, t) - \Delta H - DeltaI$$

The geoïd variation include both, variation of the geoïd $\Delta \mathcal{G}tot$ surface and variation of the ocean volume. The variation of the ocean volume following a conservation of the mass, denoted $\frac{\Delta \Phi}{g}$.

$$\Delta Gtot(\Theta, \Psi, t) = \Delta \mathcal{G}tot(\Theta, \Psi, t) - \frac{\Delta \Phi}{g}$$

Both geoïd and ground variations ($\Delta Xtot$) can be decomposed into variations (ΔX) due to mass redistribution and varaitions induced by earth rotation (ΔX^T).

$$\Delta Xtot(\Theta, \Psi, t) = \Delta X^T(\Theta, \Psi, t) + \Delta X(\Theta, \Psi, t)$$

The resulting SLE is :

$$\begin{aligned}\Delta SL(\Theta, \Psi, t) = \Delta \mathcal{G}^T(\Theta, \Psi, t) + \Delta \mathcal{G}(\Theta, \Psi, t) - \frac{\Delta \Phi}{g} \\ -\Delta R^T(\Theta, \Psi, t) - \Delta R(\Theta, \Psi, t) - \Delta H - \Delta I\end{aligned}$$

This equation shows that variations in relative sea level are the result of the interaction of geoid and ground variations induced by mass variations, plus the effect of the earth's rotation, plus respectively the redistribution of water masses between ice, sediment and ocean and variations in the earth's surface due to sedimentary input and ice.

1.2.2 Conservation of mass

The term $\frac{\Delta \Phi}{g}$ follows a conservation of mass equation based on the variation of ice (ΔI) and ocean volume (ΔS).

$$\iint_\Omega \Delta I \, \mathrm{d}\Omega = -\frac{\rho_\mathrm{W}}{\rho_\mathrm{I}} \iint_\Omega \Delta S \, \mathrm{d}\Omega.$$

ΔS include three variations, the variations of the sea level, the variation of ocean volume due to ice ocean interaction and the variation of the ocean surface. These three variations are expressed as follows :

$$\Delta S = \Delta \mathcal{SL} \cdot C + \frac{\Delta \Phi}{g} C - T_0 \left[C - C_0\right]$$

Where T_0 is the initial ocean volume and C is the ocean function (1 in the ocean and 0 on the continent) :

$$C = \begin{cases} 1 & \text{if } Z > 0 \\ 0 & \text{if } Z \leq 0 \end{cases}$$

Injecting this expression to the conservation of mass we obtain :

$$\begin{aligned}\frac{\Delta \Phi}{g} = &-\frac{1}{\mathcal{A}} \frac{\rho_\mathrm{I}}{\rho_\mathrm{w}} \iint_\Omega \Delta I \, \mathrm{d}\Omega - \frac{1}{\mathcal{A}} \iint_\Omega \Delta \mathcal{SL} C \, \mathrm{d}\Omega \\ &+ \frac{1}{\mathcal{A}} \iint_\Omega T_0 \left[C - C_0\right] \mathrm{d}\Omega,\end{aligned}$$

with $\mathcal{A} \equiv \iint_\Omega C \, \mathrm{d}\Omega$

Behind the ocean function the variation of topography include ice and sediment thickness. The conservation term $\frac{\Delta \Phi}{g}$ include then the replacement of ocean by sediment.

1.2.3 Development of ΔG and ΔR

To determine both ΔG and ΔR, denoted from here $\Delta\chi$, [Peltier, 1974] and [Mitrovica and Peltier, 1989] introduce the Green's functions that describe the response of a radial symetric self gravitating sphere. The relation include a spatial and temporal convolution between the Green functions and the Load ΔM.

$$\Delta\chi(\Theta,\Psi,t) = \int_{-\infty}^{t}\iint_{\Omega}\Delta M\left(\Theta',\Psi',t'\right)\cdot GF\left(\gamma,t-t'\right)\mathrm{d}\Omega'\mathrm{d}t'$$

Where γ is $cos(\gamma) = cos(\theta)cos(\theta') + sin(\theta)sin(\theta')cos(\psi-\psi')$. GF here denote the Green function.

Case of a non-rotational Earth

The GFs follows the love numbers theory [Love, 1892]. Our code differs from the work of [Dalca *et al.*, 2013] by using decay love numbers where the normal mode of love numbers was used (see section xx for details about the love numbers used in this code). We use the h and k love numbers and derive for both the elastic (x_ℓ^E) and viscous (decay, $x_\ell^V(t)$) part.

Here we are working on two GF, for the geoïd ($\phi(\gamma,t)$) and the ground ($\Gamma(\gamma,t)$) vertical motion.

$$\phi(\gamma,t) = \frac{ag}{M_\mathrm{e}}\sum_{\ell=0}^{\infty}\left[\delta(t)+k_\ell^E\delta(t)+k_\ell^V(t)\right]P_\ell(\cos\gamma)$$

$$\Gamma(\gamma,t) = \frac{ag}{M_\mathrm{e}}\sum_{\ell=0}^{\infty}\left[h_\ell^E\delta(t)+h_\ell^V(t)\right]P_\ell(\cos\gamma)$$

Where a is the Earth radius, M_e the Earth mass, g the gravitational constant of earth and $\delta(t)$ is the Dirac function. For the non-rotational part, in the convolution, GFs are applied to the Load ($\Delta L(\Theta,\Psi,t)$) a pure variation of masses.

$$\Delta\chi(\Theta,\Psi,t) = \int_{-\infty}^{t}\iint_{\Omega}\Delta L\left(\Theta',\Psi',t'\right)\cdot GF\left(\gamma,t-t'\right)\mathrm{d}\Omega'\mathrm{d}t'$$

Case of a rotational Earth

The effect of rotation on sea level is expressed by the perturbation of Earth's rotational vector solved by using tidal love numbers k^T and h^T [Milne and Mitrovica, 1998] in the GFs, for both elastic $x_\ell^{T,E}$ and viscous $x_\ell^{T,V}(t)$.

$$\phi^T(\gamma,t) = \frac{ag}{M_\mathrm{e}}\sum_{\ell=0}^{\infty}\left[\delta(t)+k_\ell^{T,E}\delta(t)+k_\ell^{T,V}(t)\right]P_\ell(\cos\gamma)$$

$$\Gamma^T(\gamma,t) = \frac{ag}{M_\mathrm{e}}\sum_{\ell=0}^{\infty}\left[h_\ell^{T,E}\delta(t)+h_\ell^{T,V}(t)\right]P_\ell(\cos\gamma)$$

Where

For the rotational Earth convolution a rotational potential is defined as $\Lambda(\Theta,\Psi,t_j)$. The equations behind are described in [Milne and Mitrovica, 1998] and are not developed here.

1.2.4 Resolution of temporal and spatial convolution

Spatial convolution

The spatial convolution is resolved using the spherical harmonic transformation. For a function $\chi(\Theta, \Psi, t)$, we can define spherical harmonics coefficients $\chi_{lm}(t)$, where l is the degree and m is the order of the associated Legendre polynomial ($Y_{lm}(\Theta, \Psi)$) :

$$\mathcal{X}(\Theta, \Psi, t) = \sum_{lm} \mathcal{X}_{lm}(t) Y_{lm}(\Theta, \Psi)$$

with $\sum_{lm} = \sum_{l=0}^{\infty} \sum_{m=-l}^{m=l}$, for a degree l there is 2l+1 order.

In the spectral domain the convolution can be solved :

$$\iint_{\Omega} \sum_{l=0}^{infty} \mathcal{X}(\Theta', \Psi', t) P_l(cos\gamma')\, d\Omega = T_l \sum_{lm} \mathcal{X}_{lm}(t) Y_{lm}(\Theta, \Psi)$$

With $T_l = \frac{4\pi a^2}{2l+1}$

Temporal convolution

The resolution of temporal convolution is performed by a Heaviside distribution of the load $\mathcal{H}(t)$.

$$\mathcal{H}(t) = \begin{cases} 1 & \text{si t>0} \\ \varnothing & \text{si t=0} \\ 0 & \text{si t<0} \end{cases}$$

The Heavyside distributed load is :

$$\Delta L(\Theta, \Psi, t) = \sum_{n=0}^{N} \delta L(\Theta, \Psi, t_n) \mathcal{H}(t - t_n)$$

Resolution of the convolutions

Applying the temporal convolution resolution :

$$\Delta\chi = \iint_{\Omega} \sum_{n=0}^{N} \delta M(\Theta, \Psi, t_n) \int_{-\infty}^{t} \mathcal{H}(t - t_n) \cdot GF\,(\gamma, t - t')\, \mathrm{d}\Omega' \mathrm{d}t'$$

and :

$$\int_{-\infty}^{t} \mathcal{H}(t - t_n) \cdot GF\,(\gamma, t - t')\, \mathrm{d}t' = IGF\,(\gamma, t - t_n)$$

with IGF the time integration of the GF.

We have then :

$$\Delta\chi(\Theta, \Psi, t) = \iint_{\Omega} \sum_{n=0}^{j-1} \delta M(\Theta, \Psi, t_n) \cdot IGF\,(\gamma, t_j - t_n)$$

By application of the spatial convolution solution :

$$\Delta\chi(\Theta,\Psi,t) = \sum_{lm} T_l \sum_{n=0}^{j-1} \delta M_{lm}(t_n) Y_{lm}(\Theta,\Psi) \cdot IGF\,(\gamma, t-t_n)$$

The respective IGF are :

$$I\phi(\gamma,t) = \frac{ag}{M_e} \sum_{\ell=0}^{\infty} \left[1 + k_\ell^E + K_\ell^V(t)\right]$$

$$I\Gamma(\gamma,t) = \frac{ag}{M_e} \sum_{\ell=0}^{\infty} \left[h_\ell^E + H_\ell^V(t)\right]$$

$$I\phi^T(\gamma,t) = \frac{ag}{M_e} \sum_{\ell=0}^{\infty} \left[1 + k_\ell^{T,E} + K_\ell^{T,V}(t)\right]$$

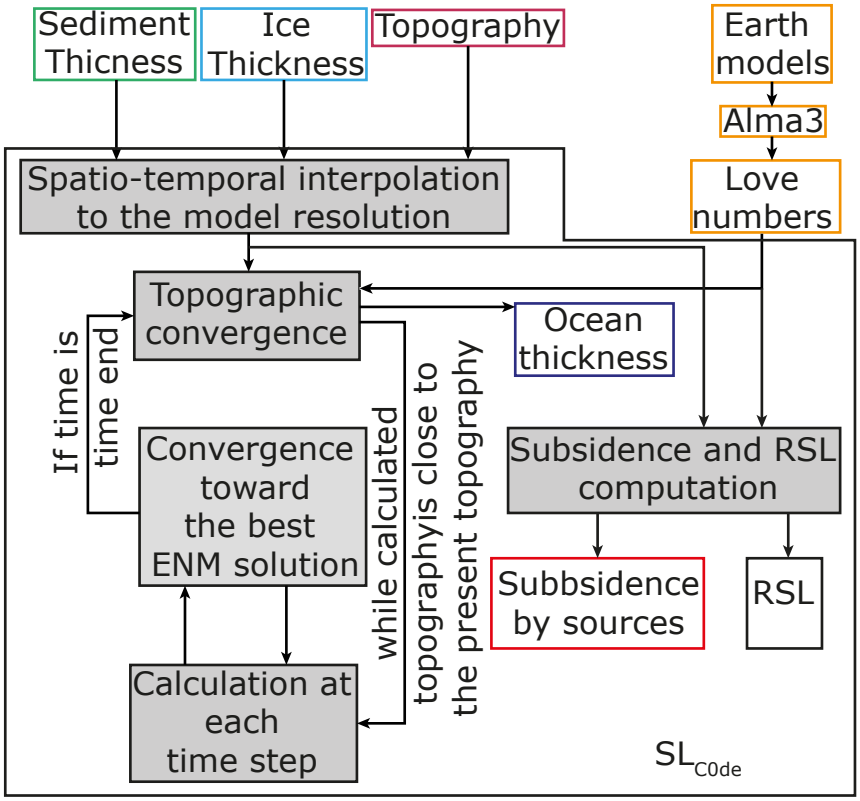
$$I\Gamma^T(\gamma,t) = \frac{ag}{M_e} \sum_{\ell=0}^{\infty} \left[h_\ell^{T,E} + H_\ell^{T,V}(t)\right]$$

Where K and H are the integrated love numbers between 0 and the considered time t.

The resulting SLE is :

$$\Delta\mathcal{SL}(\Theta,\Psi,t) = \int_{-\infty}^{t_j} \iint_{\Omega} \Delta L(\Theta',\Psi',t') \cdot [\frac{\Phi(\gamma,t-t')}{g} - \Gamma(\gamma,t-t')]\mathrm{d}\Omega'\mathrm{d}t' - \Delta H(\Theta,\Psi,t) - \Delta I(\Theta,\Psi,t)$$

1.3 Numerical implementation



The numerical implementation is using the derived impletation of [Dalca *et al.*, 2013] from [Kendall *et al.*, 2005]. This implementation is using two iteration counter, i and j. j is associated to the time iteration and at each iteration the implementation iterate a convergence

to find the best solution of the SLE where the counter is i. The i counter will be called the inner iteration. From $\delta S_j^{i=1}$ to $\delta S_j^{i=\infty}$ to will calculate the best solution of SLE. The numerical implementation require a third counter, k called outer iteration, where the loop over the whole considered time is covered to improve first guess of initial topography, $T_0^{k=1}$ until the convergence ($T_0^{k=\infty}$).

The SLE is rewrite to :

$$\Delta\mathcal{SL}_j = \int_{-\infty}^{t_j} \iint_\Omega \Delta L(\Theta', \Psi', t') \cdot [\frac{\Phi(\gamma, t_j - t')}{g} - \Gamma(\gamma, t_j - t')]\mathrm{d}\Omega'\mathrm{d}t' - \Delta H_j - \Delta I_j$$

1.3.1 Sea Level equation resolution

Sea level equation Resolution implementation

We introduce the variation of ocean thickness :

$$\delta S_j^{i,k} = -\Delta S_{j-1}^{i=\infty,k} + \Delta\mathcal{SL}_j^{i-1,k} C_j^{k-1} + \frac{\Delta\Phi(t_j)^{i-1,k}}{g} C_j^{k-1} - T_0^{k-1}[C_j^{k-1} - C_0^{k-1}]$$

If the last step of the outer iteration was completed, $k-1$. The topography is updated :

$$T_j^{k-1} = T_p + \Delta SL_p^{i=\infty,k-1} - \Delta SL_j^{i=\infty,k-1}$$

Where T_p is the present day topography, we substract to the present day topography the earth movement. The resulting ocean function deduced from the reformulation of the topography is :

$$C_j^{k-1} = \begin{cases} 1 & \text{if } T_j^{k-1} < 0 \\ 0 & \text{if } T_j^{k-1} \geq 0 \end{cases}$$

The sea level can be then estimated using a new estimation of the sea level change for the kth iteration.

$$\Delta\mathcal{SL}_j^{i-1,k} = \Delta\mathcal{G}_j^{i-1,k} - (\Delta R_j^{i-1,k} + \Delta H_j + \Delta I_j^{k-1})$$

for the first inner iteration $i = 1$, the initial variation of ocean thickness is predefine. The spatially invariant component is resulting from the variable ocean surface.

$$\frac{\Delta\Phi_j^{i-1,k}}{g} = \frac{-1}{\mathcal{A}_j^{k-1}} \frac{\rho_I}{\rho_w} \iint_\Omega \Delta I_j^{k-1} d\Omega - \frac{-1}{\mathcal{A}_j^{k-1}} \iint_\Omega \Delta\mathcal{SL}_j^{i-1,k} C_j^{k-1} d\Omega + \iint_\Omega T_0^{k-1}[C_j^{k-1} - C_0^{k-1}]d\Omega$$

With

$$\mathcal{A}_j^{k-1} = \iint_\Omega C_j^{k-1} d\Omega$$

Resolution of SLE including the deconvolution

The implementation in iteration result in a modification of the geoid and ground displacement :

$$\Delta\chi_j = \sum_{lm} T_l \sum_{n=0}^{j-1} \delta M_{lm}(t_n) Y_{lm}(\Theta, \Psi) \cdot IGF\,(\gamma, t_j - t_n)$$

This applied to the SLE equation, by linearity of the IGFs:

$$[\Delta\mathcal{SL}_{lm,j}]^{i-1,k} = T_l E_l \Delta M_{lm,j}^{k,i} + T_l \sum_{n=0}^{j-1} \beta(l, t_n, t_j) \delta M_{lm,n}^{k,i} + \frac{1}{g} E_l^T([\Delta\Lambda_{lm,j-1}]^{i=\infty,k} + [\delta\Lambda_{lm,j}]^{i-1,k}) + \frac{1}{g} \sum_{n=0}^{j-1} \beta^T(l, t_n, t_j)[\delta\Lambda_{lm,j}$$

where $E_l = 1 + k_l^E - h_l^E$, $\beta(l, t_n, t_j) = k_l^V(t_j - t_n) - h_l^V(t_j - t_n)$ and $T_l = \frac{4\pi a^2}{2l+1}$

In SL_C0de, $T_l \sum_{n=0}^{j-1} \beta(l, t_n, t_j) \delta M_{lm,n}^{k,i}$ is resolved in matrix produce. This result in a strong allocation of RAM as the $\beta(l, t_n, t_j)$ are stored in a matrix of size (time,time,(maximum degree + 1)(maximum degree +2)/2). The resulting time gain is very important.

The conservation formula become :

$$\frac{\Delta\Phi_j}{g} = \frac{1}{C_{00,j}}(-\frac{\rho_i}{rho_w}\Delta I_{00,j} - RO_{00,j} + TO_{00,0})$$

Where $RO_j = \Delta\mathcal{SL}_j C_j$ and $TO_j = T_0[C_j - C_0]$.

Convergence parameter

Inner convergence on SLE

We define a convergence criterion :

$$\xi_j^{i,k} = |\frac{\sum_{l,m} |[\delta S_{lm}(t_j)]^{i,k}| - \sum_{l,m} |[\delta S_{lm}(t_j)]^{i-1,k}|}{\sum_{l,m} |[\delta S_{lm}(t_j)]^{i-1,k}|}|$$

Convergence for the SLE is limited by the convergence criterion : $\xi_j^{i,k}$. We suppose that $\xi_j^{i,k} < \epsilon_1$ when

$$[\delta S_{lm}(t_j)]^{i,k} = [\delta S_{lm}(t_j)]^{i=\infty,k}$$

Outer convergence criterion

Grounded ice correction

The marine grounded ice is dependent of RSL variations. The ice is grounded if it satisfies :

$$I_j > (SL_j + I_j)\frac{\rho_w}{\rho_I}$$

At each topographic iteration (k) we update the grounded ice.

$$I_j^k = \begin{cases} Ice\ Height & SL_j^{k-1} + Ice\ Height < 0 \\ Ice\ Height & SL_j^{k-1} + Ice\ Height > 0 \\ & and\ Ice\ Height > SL_j^{k-1}, \frac{\rho_w}{\rho_I - \rho_w} \\ 0 & elsewhere \end{cases}$$

1.3.2 Computation of ground and geoid subsidence from different load source

A functionality developed in SL_{C0de} is the computation of the different component of the SLE separately, by type of Load and by viscous or elastic component. The development of this functionality was motivated by the necessity of exploring the different source of the RSL variation in a more and more complex modelization.

Elastic components of SLE :

We define 4 elastique component in the SLE, the ground displacement ΔR^E_{lm}, the geoid displacement ΔG^E_{lm}, the rotational ground displacement $\Delta R^{T,E}_{lm}$ and the rotational geoid displacement $\Delta G^{T,E}_{lm}$.

$$\Delta R^E_{lm,j} = T_l h^E_l \Delta M^{k,i}_{lm,j}$$

$$\Delta G^E_{lm,j} = T_l (1 + k^E_l) \Delta M^{k,i}_{lm,j}$$

$$\Delta R^{T,E}_{lm,j} = \frac{1}{g} h^{T,E}_l ([\Delta\Lambda_{lm,j-1}]^{i=\infty,k} + [\delta\Lambda_{lm,j}]^{i-1,k})$$

$$\Delta G^{T,E}_{lm,j} = \frac{1}{g} (1 + h^{T,E}_l)([\Delta\Lambda_{lm,j-1}]^{i=\infty,k} + [\delta\Lambda_{lm,j}]^{i-1,k})$$

Viscous components of the SLE :

We define also 4 viscous component in the SLE, the ground displacement ΔR^V_{lm}, the geoid displacement ΔG^V_{lm}, the rotational ground displacement $\Delta R^{T,V}_{lm}$ and the rotational geoid displacement $\Delta G^{T,V}_{lm}$.

$$\Delta R^V_{lm,j} = T_l \sum_{n=0}^{j-1} h^V_l (tj - tn) \delta M^{k,i}_{lm,n}$$

$$\Delta G^V_{lm,j} = T_l \sum_{n=0}^{j-1} k^V_l (tj - tn) \delta M^{k,i}_{lm,n}$$

$$\Delta R^{T,V}_{lm,j} = \frac{1}{g} \sum_{n=0}^{j-1} h^V_l (tj - tn) [\delta\Lambda_{lm,j}]^{i=\infty,k}$$

$$\Delta G^{T,V}_{lm,j} = \frac{1}{g} \sum_{n=0}^{j-1} k^V_l (tj - tn) [\delta\Lambda_{lm,j}]^{i=\infty,k}$$

True sediment subsidence

This library was originally developed to compute effect of sediment on RSL. We considered the pure effect of sediment on RSL but also a corrected effect of sediment from water replacement. The sediment, when they are deposited, replace water and then generates an uplift induced by the diminution of ocean thickness. We choose to correct the sediment from the ocean load.

$$\delta M_{lm,n} = \delta H_{lm,n} C_{lm,n} \rho_w$$

To estimate effect of sediment on RSL, you must substract the effect of the mass variation described above to the effect of sediment mass variation.

Relative sea level variations

We estimate a pure RSL where the sea level is not including variations of sediment thickness and ice thickness.

$$\Delta SL^{i-1,k}_j = \Delta \mathcal{G}^{i-1,k}_j - \Delta R^{i-1,k}_j + \frac{\Delta \Phi^{i-1,k}_j}{g}$$

The other estimation is the full RSL :

$$\Delta SL_j^{i-1,k} = \Delta \mathcal{G}_j^{i-1,k} - (\Delta R_j^{i-1,k} + \Delta H_j + \Delta I_j^{k-1}) + \frac{\Delta \Phi_j^{i-1,k}}{g}$$

The resulting estimation of RSL can be compared with the ESL (only $\frac{\Delta \Phi_j^{i-1,k}}{g}$).

1.3.3 Input data format

Mass grid format

The different mass grid can be input as height grid, converted to mass by a simple multiplication by a defined density or as mass grid directly. The grids can be irregular or regular, they are interpolated over a sphere using stripy. These data are input as the derivative variations over time.

The topography as initial parameter is the present day topography. The initialization will update the topography according to the ice and sediment thickness.

Implementation of Love numbers

The [Dalca *et al.*, 2013] theory is based on the love number theory which forces us to calculate love numbers. The love numbers exits in two forms, normal mode and decay. They can also include compressible processes. We choose for computation facilities to use the love numbers computed by ALMA3 code [Melini *et al.*, 2022]. This code is calculating incompressible decay love numbers. Benchmarking on compressible vs incompressible love numbers have demonstrated no significant difference in computed vertical displacement over 256 spherical harmonics degree. We urge you to use this code with a degree higher than 256.

The code is working with a precise file structure for love numbers :

earth_model_name
├── h_e.dat
├── h_e_T.dat
├── h_ve.dat
├── h_ve_T.dat
├── k_e.dat
├── k_e_T.dat
├── k_ve.dat
├── k_ve_T.dat
├── l_e.dat
├── l_e_T.dat
├── l_ve.dat
├── l_ve_T.dat
└── time.dat

Table 1: Corresponding love number from equation to the file names

file	Love numbers
h_e	h_ℓ^E
h_e_T	$h_\ell^{T,E}$
h_ve	$h_\ell^V(t)$
h_ve_T	$h_\ell^{T,V}(t)$
k_e	k_ℓ^E
k_e_T	$k_\ell^{T,E}$
k_ve	$k_\ell^V(t)$
k_ve_T	$k_\ell^{T,V}(t)$
l_e	l_ℓ^E
l_ve	$l_\ell^{T,E}$
l_ve	$l_\ell^V(t)$
l_ve_T	$l_\ell^{T,V}(t)$

The time.dat file contains the time at which the viscous decay love numbers are computed. An example file of configurations files for ALMA3 is provided in the code supplementary files.

Note: Add the link to the ALMA3 configuration files.

1.4 Developer guide

1.4.1 SOLVER

Note: The entire functions used here must be verified before publication of the code. They have not been tested yet.

Functions

SL_C0de.SOLVER.**Precomputation**(*ice_grid*, *sed_grid*, *topo_grid*, *Output_way*, *stop=26*, *step=0.5*, *maxdeg=512*, *irregular_time_step=None*, *backend=False*)

The Precomputation method prepare the different data to match temporal and spatial resolution of the modelisation. spatial resolution (m,n) and temporal resolution (step). The input data format is described in (*Mass grid format* p.14).

Attribute :

ice_grid
[dict(name = str, grid = np.array((n_i,m_i,t_i)), time = np.array((t_i,)), lon = np.array((n_i)), lat = np.array((m_i)))] ice thickness grid

sed_grid
[dict(name = str, grid = np.array((n_j,m_j,t_j)), time = np.array((t_j,)), lon = np.array((n_j)), lat = np.array((m_j)))] sediment thickness

topo_grid
[dict(name = str, grid = np.array((n_k,m_k)), lon = np.array((n_j)), lat = np.array((m_j)))] present topography

Output_way
[str] The filepath toward wich the different grid will be saved

stop
[float] age at wich the computation stop

step
[float] time step used for the temporal resolution of the model

maxdeg
[int] The maximum degree of spherical harmonics, that define the spatial resolution of the model. (m = 2*maxdeg, n = maxdeg)

irregular_time_step
[np.array((time_step_number,))] An irregular time array to compute the model over non regular time step

backend
[bool] If required you can ask a backend during the run of each computation. True, will generate a backend, False will not. Defaul is False

Returns :

None

The data output of the function are saved to the output way.

SL_C0de.SOLVER.**SLE_forward_modeling**(*Input_way*, *ice_name*, *sed_name*, *topo_name*, *ocean_name*, *love_way*, *love_file*, *conv_lim*)

The SLE_forward_modeling method solve the SLE with no constrain on final topography. This result in a forward modeling of the SL. It can be used for test and exploration of models. For informations on iterations see (*description of iteration* p.10).

Attribute :

Input_way
[str] File location of the input data (ice, sediment and topography)

ice_name
[str] The name of the ice data file

sed_name
[str] The name of the sediment data file

topo_name
[str] The name of the topographic data file

ocean_name
[str] The name of the output file containig the data on the ocean

love_way
[str] way of the love numbers file used to compute earth visco elastic deformation

conv_lim
[float] Limit of precision required for the convergence of the SLE resolution (10^-3)

output_way
[str] filepath for saving results of the modelisation

Returns :

None

The resulting ocean class object ((*OCEAN_TIME_GRID* p.35)) containing the ocean thickness is saved in the outputway under the name ocean.

SL_C0de.SOLVER.**SLE_solver**(*Input_way*, *ice_name*, *sed_name*, *topo_name*, *ocean_name*, *love_way*, *love_file*, *topo_lim*, *conv_lim*, *Output_way*)

The SLE_solver solve the SLE and converge toward the actual topography. The computation of the SLE resolution is describe in (*Sea level equation resolution* p.11). For informations on iterations see (*description of iteration* p.10).

Attribute :

Input_way
[str] File location of the input data (ice, sediment and topography)

ice_name
[str] The name of the ice data file

sed_name
[str] The name of the sediment data file

topo_name
[str] The name of the topographic data file

ocean_name
[str] The name of the output file containig the data on the ocean

love_way
[str] way of the love numbers file used to compute earth visco elastic deformation

conv_lim
[float] Limit of precision required for the convergence of the SLE resolution (10^-3)

topo_lim
[float] Topography convergence limit toward wich the iteration will converge (1 m)

output_way
[str] filepath for saving results of the modelisation

Returns :

None

The resulting ocean class object ((*OCEAN_TIME_GRID* p.35)), updated ice thickness ((*ICE_TIME_GRID* p.33)) and updated topography ((*TOPOGRAPHIC_TIME_GRID* p.39)) are saved to output_way.

SL_C0de.SOLVER.**Post_process**(*Input_way_sed*, *Input_way_model_output*, *love_way*)

The Post_process calculate the earth viscoelastic response resulting from the different masses caculated with the SLE_solver. This computation is made for all models available in the files.

Attribute :

input_way_sed
[str] way where the sediment precomputed are stored

input_way_model_output
[str] filepath to the different masses calculated by the SLE solver

love_way
[str] way to the love number output

Returns :

None

The resulting LOAD and GEOID are stored in a LOAD file in the same file then the input files. The structure is based on the (*LOAD_TIME_GRID* p.40) class object.

SL_C0de.SOLVER.**calculate_deformation**(*love_number*, *ice_time_grid*, *sed_time_grid*, *ocean_time_grid*, *a*, *Me*, *Output_way*, *backend=False*)

The calculate_deformation method calculate the earth viscoelastic response resulting from the different masses caculated with the (*SLE_solver* p.17). This computation follows the decomposition of the SLE described in (*Computation of ground and geoid subsidence from different load source* p.12)

Attribute :

love_number
[(*LOVE* p.45)] The loaded love number in the form of a LOVE class object

ice_time_grid
[(*grid.ICE_TIME_GRID* p.33)] ice thickness in the form of ICE_TIME_GRID class object

sed_time_grid
[(*grid.SEDIMENT_TIME_GRID* p.31)] sediment thickness in the form of SEDIMENT_TIME_GRID class object

ocean_time_grid
[(*grid.OCEAN_TIME_GRID* p.35)] ocean thickness calculated using the SLE solver in the form of OCEAN_TIME_GRID class object

a
[float] radius of earth in meter

Me
[float] mass of earth (kg)

Output_way
[str] way of the output where the load is calculated

backend
[bool] set if the function is writing its state of computation

Returns :

None

The resulting LOAD and GEOID are stored in a LOAD file in the same file then the input files. The structure is based on the (*LOAD_TIME_GRID* p.40) class object.

SL_C0de.SOLVER.**calculate_sediment_ocean_interaction**(*love_number*, *ice_time_grid*, *sed_time_grid*, *ocean_time_grid*, *a*, *Me*, *topo_time_grid*, *Output_way*, *backend=False*)

The calculate_sediment_ocean_interaction method calculate the earth viscoelastic response resulting from sediment under sea surface. This is used to retrieve the effect of ocean replacement by sediment on the sediment load. The resulting subsidence is the true subsidence induced by sediment. The problems and resolution in equation is described in (*True sediment subsidence* p.13).

Attribute :

love_number
[(*LOVE* p.45)] The loaded love number in the form of a LOVE class object

ice_time_grid
[(*grid.ICE_TIME_GRID* p.33)] ice thickness in the form of ICE_TIME_GRID class object

sed_time_grid
[(*grid.SEDIMENT_TIME_GRID* p.31)] sediment thickness in the form of SED_TIME_GRID class object

ocean_time_grid
[(*grid.OCEAN_TIME_GRID* p.35)] ocean thickness calculated using the SLE solver in the form of OCEAN_TIME_GRID class object

a
[float] radius of earth in meter

Me
[float] mass of earth (kg)

Output_way
[str] way of the output where the load is calculated

backend
[bool] set if the function is writing its state of computation

Returns :

None

The resulting LOAD and GEOID are stored in a LOAD file in the same file then the input files. The structure is based on the (*LOAD_TIME_GRID* p.40) class object.

SL_C0de.SOLVER.**find_files**(*filename*, *search_path*)

1.4.2 spharm

Functions

SL_C0de.spharm.**get_coeffs**(*a_lm*, *n*)

The get_coeffs function get the spherical harmonics coefficients of the nth order from a linearly Spharm array.

Attribute :

a_lm
[np.array([maxdeg*(maxdeg+1)/2,])] An array containing the spherical coefficient in a linear form

n
[int] The order of the spherical harmonics coefficient you are trying to retrive

Returns :

a_n
[np.array([n,])] The spherical harmonic coefficient of the order n.

CLASS

class SL_C0de.spharm.**sphericalobject**(*grd=None*, *coeff=None*, *maxdeg=None*)

The sphericalobject class include any spherical object. This class is working with pyshtools (pyshtools).

Attributes

grd
[np.array[(maxdeg,maxdeg x2)]] Value of the spherical object on a Gaussian Grid.

isgrd
[Bool] A boolean to define if a Gaussian grid have been defined for this object.

coeff
[np.array[(maxdeg,maxdeg)]] Spherical harmonic coefficient array.

iscoeff
[Bool] A boolean to define if a spherical harmonic coefficient have been defined for this object.

saved
[np.array[(n, maxdeg, maxdeg)]] An array wich contain the spherical harmonic coefficient each time th save method is applied.

prev
[np.array[(maxedg, maxdeg)]] save the spherical coefficient using save_prev.

Methods

(*grdtocoeff* p.21)
[] Convert the Gaussian grid to spherical harmonic coefficient

(*coefftogrd* p.21)
[] Convert spherical harmonic coefficient to Gaussian grid

(*coefftogrdhd* p.22) :
Convert spherical harmonic coefficient to a Gaussian grid with a higher resolution then maxdeg

(*save_prev* p.22) :
Save the spherical harmonic coefficient to the attribute prev

Methods

SL_C0de.spharm.sphericalobject.**grdtocoeff**(*self*)

The grdtocoeff method convert a Gaussian grid into spherical harmonic coefficient array using a numerical method to create spherical harmonic coefficient self.coeff is updated usig these output. self.iscoeff defining if a coefficient have been created for this object. If there is no grid created (self.isgrid == 0) then it returns an error.

Attribute :

None

Returns :

None

SL_C0de.spharm.sphericalobject.**coefftogrd**(*self*)

The coefftogrd method convert spherical harmonic coefficient into a gird array using shtools. The output of pysh.SHCoeff are converted to real. self.grd is updated usig these output. self.isgrd defining if a grid have been created for this object. If there is no grid created (self.iscoeff == 0) then it returns an error. A modifier pour pouvoir modifier les entrées de la fonction.

Parameters :

See the documentation of the cited class object for more information on different parameters used in the function.

Returns :

Added fields :

SL_C0de.spharm.sphericalobject.**coefftogrdhd**(*self*, *max_calc_deg*)

The coefftogrdhd convert spherical harmonic coefficient into a gird array using shtools. The output of pysh.SHCoeff are converted to real. self.grd is updated usig these output. self.isgrd defining if a grid have been created for this object. If there is no grid created (self.iscoeff == 0) then it returns an error. A modifier pour pouvoir modifier les entrées de la fonction.

Attribute :

max_calc_deg
[int] The maximum spherical harmonic degree to calculate the grid. This function can be used to have a better rendering in output.

Returns :

None

SL_C0de.spharm.sphericalobject.**save_prev**(*self*)

The save_prev create a new field for the object to save the spherical coefficient at the moment of the applied function. This function make a clean copy of the array to avoid modification.

Attribute :

None

Returns :

None

1.4.3 Grid

The Grid module is used to manage all the data in there version grided. This module define object class forgrid management, time grid management.

Note: Je doit creuser plus en détail la relation avec le module Love pour éviter tout problème.

GRID

CLASS

class SL_C0de.grid.**GRID**

The class GRID is used to represent the Gaussian Grid.

Attributes

Note: This grid have no attribute * this might be changed in future version to be used as stand alone. The absence of attribute is the result of the use of this function in (*TIME_GRID* p.25) that use that define all parameters used in the init function of GRID.

Methods

(*interp_on* p.23)
[] method used to interpolate the grid over the model grid.

(*smooth_on* p.24)
[] method used to smooth the grid to reduce noise effect.

(*disk* p.24)
[] method used to create a disk of certain shape to test parameters.

Methods

SL_C0de.grid.GRID.**interp_on**(*self*, *grd*, *lon*, *lat*, *smoothing=False*, *grid_type='global'*, *error=False*)

The method interp_on interpolate a grid of data on the grid of the model calculated in the init function of (*GRID* p.22). This function is using stripy library to pursue the interpolation (Stripy library).

Attribute

grd
[np.array([m,n])] The grid data over space.

lon
[np.array([m,])] The longitude of the data.

lat
[np.array([n,])] The latitude of the data.

smoothing
[bool] If a smoothing is applied to the grid before the interpolation, see (*smooth_on* p.24).

grid_type
[str] grid type define if it's a grid over the whole world or over a small area. This is used to avoid long computation for the interpolation in the case of interpolating small areas over the world. There is two possible 'global' and 'local'. The global interpolation is using stripy.sTriangulaion. The local interpolation is using stripyt.Triangulation.

error
[bool] If true, the method return the error of the interpolation calculated by stripy.

Return

grd
[np.array([maxdeg*2,maxdeg])] The grid interpolated on the model grid.

SL_C0de.grid.GRID.**smooth_on**(*self*, *grd*, *lon*, *lat*)

The method smooth_on is smoothing the grid over the area whith smoothing. This function can be used to correct values before the interpolation over time and space. This way, you can have better results on topographic convergence.

Attribute

grd
[np.array([m,n])] Array containig the grid values over the space.

lat
[nparray([n,])] latitude.

lon
[nparray([m,])] longitude.

Return

grd
[np.array([m,n])] smoothed array with the same shape then the initial grid.

SL_C0de.grid.GRID.**disk**(*self*, *lat*, *lon*, *radius*, *high*, *tx=1*)

disk is a method used to create a thickness grid. This grid can be used to test different parameters.

Attribute

lat
[nparray([1,])] Array contaning the latitudinal coordinate in degree of the center of the disk.

lon
[nparray(1,[])] Array containing the longitudinal coordinate in degree of the center of the disk.

radius
[double] The radius in degree of the disk in degree.

high
[double] The thickness in meter of the disk over the considered area.

Return

grd
[np.array([maxdeg*2,maxdeg])] The grid as defined by the (*GRID* p.22) class with a disk of thikness high at lon,lat position with the size of radius.

SL_C0de.grid.GRID.**zeros**(*self*, *tx=1*)

zeros is a method used to generate a zero array with the caracteristics of the grid.

Attribute :

tx
[int] times the thickness is repeated

Return :

np.zeros((tx,self.lats.size,self.elons.size))
An array containing only zeros

SL_C0de.grid.GRID.**along_transect**(*self*, *coord=('lat_start', 'lon_start', 'lat_stop', 'lon_stop')*, *point_density=None*, *point_distance=None*)

TIME_GRID

CLASS

class SL_C0de.grid.**TIME_GRID**(*time_step=array([1, 2])*, *maxdeg=64*, *height_time_grid=None*, *mass_time_grid=None*, *mass_time_coeff=None*, *height_time_coeff=None*, *rho=0*, *grid_name='time_grid'*, *from_file=(False,)*, *superinit=False*)

The TIME_GRID class is used to manage the mass grids. These grids have a time dimenssion this way wa can manage the time variation of the mass. We can define the mass grid by it's mass directly or by coupling a height with a density. If needed you can load spherical harmonics coefficient.Tis class is inheriting the methods from (*sphericalobject* p.20) and (*GRID* p.22).

Attributes

time_step
[np.array([time_step_number,])] This array contains the time step of the data you are importing. They will be use for temporal interpolation.

maxdeg
[int] Maximum harmonic coefficient degree of the data. this define the chape of the grid and coefficient arrays

height_time_grid
[np.array([maxedg*2,maxdeg])] This array is the height grid at each time steps defined in grid_time_step

mass_time_grid
[np.array([maxedg*2,maxdeg])] This array is the mass grid at each time steps defined in grid_time_step

height_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the height spherical harmonic coefficient at each time steps defined in grid_time_step

mass_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the mass spherical harmonic coefficient at each time steps defined in grid_time_step

rho
[float] The density of the considered layer.

Note: In future development the density may vary threw space and time. We'll have to make a variable object more then a constant density.

---

grid_name
[str] The name of the grid. We recommand you to choose a specific name for each grid you create. This name is used to save the grid in an nc file with (*save* p.31).

from_file
[(bool,way)] This parameter define if the data are new or loaded from a previously saved model in a nc file. If the first element is False, the code will create a blank object, based on provided datas. If the first element is True, the method will get the data from the file way specified in the second element of this attribute.

superinit
[bool] This parameter is used to specify if the object is used as herited method in an initialisation of a child class object.

Methods

(*interp_on_time* p.27)
Interpolate a grid over the time considered in the model

(*interp_on_time_and_space* p.27) :
Interpolate the grid over time and space as in the defined Grid during the initialisation of the class

(*grid_from_step* p.28) :
Get the grid for a defined time iteration

(*coeff_from_step* p.28) :
Get the spherical harmonics coefficient for a defined time iteration

(*timegrdtotimecoeff* p.29) :
Convert the grid into spherical harmonics coefficient for all time steps

(*timecoefftotimegrd* p.29) :
Convert the spherical harmonics coefficient into grid for all time steps

(*zeros_time* p.29) :
Generate a zero grid for all time steps

(*disk_time* p.30) :
Generate a disk of a specified thickness at a specified location over all time steps

(*update_0* p.30) :
Update the 0 time step data of the grid

(*save* p.31) :
Save the grid in a specified nc file with the name of the grid

---

Note: This class is under active developement and not usable right now

---

Methods

SL_C0de.grid.TIME_GRID.**interp_on_time**(*self*, *grid_to_interp*, *grid_time_step*, *model_time_step*, *interp_type='Thickness_divide'*, *backend='False'*, *grid_type='regular'*)

The function interp_on_time is used for interpolation upon time and space it call the interpolation function of the (*GRID* p.22) parameter. This function adapt the order of time and space interpolation to reduce computation time. The temporal interpolation try to preserve the thickness of the overall time. Tis is down by cutting and merging time steps of the original grid to match the model time_step.

Attributes

grid_to_interp
[np.array([k,n,m])] The grid to be interpreted.

grid_time_step
[np.array([k,])] The time value of each time step of the grid model.

model_time_step ; np.array([time_step_number,])
The time values of the model.

interp_type
[str] No use of this parameter anymore

backend
[bool] Define if the function retur, backends. True it will return the backends, False (default value) don't give any backend.

Return :

grid_interpolated
[np.array([time_step_number,n,m])] The interpolated grid over time. Depending of the model parameters.

SL_C0de.grid.TIME_GRID.**interp_on_time_and_space**(*self*, *grid_to_interp*, *grid_time_step*, *grid_lon*, *grid_lat*, *interp_type='Thickness_divide'*, *backend=False*, *grid_type='global'*)

The interp_on_time_and_space function is used for interpolation upon time and space it call the interpolation function of the (*GRID* p.22) parameter. This function perform the temporal and spatial interpolation in different order to ameliorate the computation time. If the temporal resolution of the input grid is higher than the model time resolution the temporal resolution will be perform first. The spatial resolution is performed first in the other case.

Attributes

grid_to_interp
[np.array([k,n,m])] The grid to be interpreted.

grid_time_step
[np.array([k,])] The time value of each time step of the grid_to_interp.

grid_lon
[np.array([n])] The longitudinal coordinates of the grid_to_interp.

grid_lat
[np.array([m])] The latitudinal coordinate of the grid_to_interp.

interp_type
[str] No use of this parameter anymore

backend
[bool] Define if the function retur, backends. True it will return the backends, False (default value) don't give any backend.

Return :

grid_interpolated
[np.array([time_step_number,maxdeg*2,maxdeg])] The interpolated grid over time. Depending of the model parameters.

SL_C0de.grid.TIME_GRID.**grid_from_step**(*self*, *t_it*)

The grid_from_step method is used to get the value of the grid at the defined time step.

Attributes :

t_it
[int] This is the value of the time step iteration on wich you are trying to retreave the grid. It must inside the time_step interpolation you have used during the initialisation of the time grid.

Return :

None

SL_C0de.grid.TIME_GRID.**coeff_from_step**(*self*, *t_it*)

The coeff_from_step method is used to get the value of the coefficient at the requested time iteration.

Attributes :

t_it
[double] This is the value of the time step iteration on wich you are trying to retreave the coefficient. It must be inside the time_step interpolation you have used during the initialisation of the time grid.

Return :

None

SL_C0de.grid.TIME_GRID.**timegrdtotimecoeff**(*self*)

The timegrdtotimecoeff method transform for each time step the grid into spherical harmonics coefficient.

Attribute :

None

Result :

None

SL_C0de.grid.TIME_GRID.**timecoefftotimegrd**(*self*)

The timecoefftotimegrd method transform for each time step the spherical harmonic coefficient into a grid.

Attribute :

None

Result :

None

SL_C0de.grid.TIME_GRID.**zeros_time**(*self*, *time_step_number*)

The zeros_time method is used to define a grid over time with only 0 value. It is based on (*GRID.zeros* p.25).

Attribute :

time_step_number
[int] The number of time step on wich we apply the zeros grid.

Return :

None

SL_C0de.grid.TIME_GRID.**disk_time**(*self*, *time_step_number*, *lat*, *lon*, *radius*, *high*)

The disk_time method is used to define a grid over time with a disk defined with it's center coordinate and the height. This function is based on (*GRID.disk* p.24).

Attribute :

time_step_number
[int] The number of time step on wich the disk load will be applyed.

lat
[double] The latitude of the center of the disk (°).

lon
[double] The longitude of the center of the disk (°).

radius
[double] The radius of the disk (°).

high
[double] The high of the disk (m).

Return :

None

SL_C0de.grid.TIME_GRID.**update_0**(*self*)

The update_0 function is used to save the first time iteration of the object before it's modification to be called at any moment in the code without alteration.

Attribute :

None

Return :

None

SL_C0de.grid.TIME_GRID.**save**(*self*, *save_way=''*, *supersave=False*)

The save function is used to save the grid and all it's parameters inside a nc file. Parameters saved are, longitude, latitude, maximum degree, the time steps of the grid, the thickness of the grid, the harmonic coefficient, grid density. The harmonic coefficient, due to complexe data type management of nc, are saved separately in there complexe and real part. The created file will have the name of the grid.

Attribute :

save_way
[str] The filepath where the data will be saved. Default value is the current file

supersave
[bool] Define if the save is called as a super method from an object that inherit the function. This precise if this method has to close the nc file (False) of if the herited class will do it (True). Default value is False.

Return :

None

SEDIMENT_TIME_GRID

CLASS

class SL_C0de.grid.**SEDIMENT_TIME_GRID**(*time_step=array([1, 2])*, *maxdeg=64*, *height_time_grid=None*, *mass_time_grid=None*, *mass_time_coeff=None*, *height_time_coeff=None*, *rho=2600*, *grid_name='time_grid'*, *from_file=(False,)*)

The SEDIMENT_TIME_GRID class is used to represent sediment deposited thickness and time over time. It inherit from (*TIME_GRID* p.25) and just add a default value of sediment density at 2600 kg/m3.

Note: This class must include the developement of [Ferrier *et al.*, 2017] on sediment compaction and it's effect on water redistriution.

Attributes

time_step
[np.array([time_step_number,])] This array contains the time step of the data you are importing. They will be use for temporal interpolation.

maxdeg
[int] Maximum harmonic coefficient degree of the data. this define the chape of the grid and coefficient arrays

height_time_grid
[np.array([maxedg*2,maxdeg])] This array is the height grid at each time steps defined in grid_time_step

mass_time_grid
[np.array([maxedg*2,maxdeg])] This array is the mass grid at each time steps defined in grid_time_step

height_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the height spherical harmonic coefficient at each time steps defined in grid_time_step

mass_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the mass spherical harmonic coefficient at each time steps defined in grid_time_step

rho
[float] The density of the considered layer. Default value is 2600.

Note: In future development the density may vary threw space and time. We'll have to make a variable object more then a constant density.

---

grid_name
[str] The name of the grid. We recommand you to choose a specific name for each grid you create. This name is used to save the grid in an nc file with **`save`_**.

from_file
[(bool,way)] This parameter define if the data are new or loaded from a previously saved model in a nc file. If the first element is False, the code will create a blank object, based on provided datas. If the first element is True, the method will get the data from the file way specified in the second element of this attribute.

Method

(*save* p.33)
used to save data

METHODS

SL_C0de.grid.SEDIMENT_TIME_GRID.**save**(*self*, *save_way=''*)

The save method is used to save the grid data to a file. It call the super method (*save* p.31) method with no additional saved parameters.

Note: Because we need to include [Ferrier *et al.*, 2017] works, this save function will be modified to include data about the sediment compaction.

Attributes

rho
[float] Density value of the sediment as a constant, default value is 2600.

Return

None

ICE_TIME_GRID

CLASS

class SL_C0de.grid.**ICE_TIME_GRID**(*time_step=array([1, 2])*, *maxdeg=64*, *height_time_grid=None*, *mass_time_grid=None*, *mass_time_coeff=None*, *height_time_coeff=None*, *rho=916.7*, *grid_name='time_grid'*, *from_file=(False,)*)

The ICE_TIME_GRID class is used to represent the ice thickness evolution threw time. This class inherit of (*TIME_GRID* p.25).

...

Attributes

time_step
[np.array([time_step_number,])] This array contains the time step of the data you are importing. They will be use for temporal interpolation.

maxdeg
[int] Maximum harmonic coefficient degree of the data. this define the chape of the grid and coefficient arrays

height_time_grid
[np.array([maxedg*2,maxdeg])] This array is the height grid at each time steps defined in grid_time_step

mass_time_grid
[np.array([maxedg*2,maxdeg])] This array is the mass grid at each time steps defined in grid_time_step

height_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the height spherical harmonic coefficient at each time steps defined in grid_time_step

mass_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the mass spherical harmonic coefficient at each time steps defined in grid_time_step

rho
[float] The density of the considered layer. Default is 916.7 kg/m3.

Note: In future development the density may vary threw space and time. We'll have to make a variable object more then a constant density.

---

grid_name
[str] The name of the grid. We recommand you to choose a specific name for each grid you create. This name is used to save the grid in an nc file with **`save`_**.

from_file
[(bool,way)] This parameter define if the data are new or loaded from a previously saved model in a nc file. If the first element is False, the code will create a blank object, based on provided datas. If the first element is True, the method will get the data from the file way specified in the second element of this attribute.

Methods

(*ice_correction* p.34)
correct the grounded ice thickness from the created floating ice by ground vertical mouvement

(*save* p.35)
used to save data

METHODS

SL_C0de.grid.ICE_TIME_GRID.**ice_correction**(*self*, *topo*, *oc*)

The ice_correction method is used to correct the grounded ice thickness from the floating ice generted by vertical ground motion. This correction is done for each time step to remove the floating ice. The correction of grounded ice is based on (*Grounded ice correction* p.12).

Attributes

topo
[(*TOPOGRAPHIC_TIME_GRID* p.39) class object] A topogrphic grid object, used to check if grounded ice become floating ice. The topography is then modified to by this function.

oc
[(*OCEAN_TIME_GRID* p.35) class object] An oceanic time grid object used only to get the density of ocean set for the model.

Note: To avoid memory consumption the oc parameter should be replace simply by oc_rho the ocean density.

Return

None

SL_C0de.grid.ICE_TIME_GRID.**save**(*self*, *save_way=''*)

The save method is used to save the data of the ice grid. Because of the (*ice_correction* p.34) method that update the grid we choosed to preserve the original ice thickness data in a ice parameter that is saved inside the nc file. Otherwise this method use the super method save to save the rest of the data.

Attributes

save_way :str
The way where the nc file is saved. Default value is the current file (an empty str).

Return

None

OCEAN_TIME_GRID

CLASS

class SL_C0de.grid.**OCEAN_TIME_GRID**(*time_step=array([1, 2])*, *maxdeg=64*, *height_time_grid=None*, *mass_time_grid=None*, *mass_time_coeff=None*, *height_time_coeff=None*, *rho=1000*, *grid_name='time_grid'*, *from_file=(False,)*)

The OCEAN_TIME_GRID class is used to represent the ocean thickness variation and contains the method to resolve the sea level equation. This method inherit from (*TIME_GRID* p.25).

...

Attributes

time_step
[np.array([time_step_number,])] This array contains the time step of the data you are importing. They will be use for temporal interpolation.

maxdeg
[int] Maximum harmonic coefficient degree of the data. this define the chape of the grid and coefficient arrays

height_time_grid
[np.array([maxedg*2,maxdeg])] This array is the height grid at each time steps defined in grid_time_step

mass_time_grid
[np.array([maxedg*2,maxdeg])] This array is the mass grid at each time steps defined in grid_time_step

height_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the height spherical harmonic coefficient at each time steps defined in grid_time_step

mass_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the mass spherical harmonic coefficient at each time steps defined in grid_time_step

rho
[float] The density of the considered layer. Default is 1000 kg/m3.

Note: In future development the density may vary threw space and time. We'll have to make a variable object more then a constant density.

---

grid_name
[str] The name of the grid. We recommand you to choose a specific name for each grid you create. This name is used to save the grid in an nc file with `save`_.

from_file
[(bool,way)] This parameter define if the data are new or loaded from a previously saved model in a nc file. If the first element is False, the code will create a blank object, based on provided datas. If the first element is True, the method will get the data from the file way specified in the second element of this attribute.

Methods

(*update_0* **p.36**)
set the grd_0 from the actual grd loaded in the grid

(*evaluate_ocean* **p.37**)
evaluate the ocean function based on the Gaussian grid of the topography

METHODS

SL_C0de.grid.OCEAN_TIME_GRID.**update_0**(*self*)
The update_0 method update the grd_0 parameter of the object to the currend loaded grd.

Attributes

None

Return

None

SL_C0de.grid.OCEAN_TIME_GRID.**evaluate_ocean**(*self*, *topo*)

The evaluate_ocean method evaluate the ocean function using the topography. It create a 0-1 matrix wich is 1 where topo<0 and 0 where topo>0. The ocean function is described in (*ocean function* p.11).

Attribute

topo
[np.array(maxdeg,maxdegx2)] topographic gaussian grid.

Returns :
None

SL_C0de.grid.OCEAN_TIME_GRID.**sea_level_solver**(*self*, *load*, *ice_time_grid*, *sed_time_grid*, *love_number*, *TO*, *t_it*, *conv_it*, *conv_lim*)

The sea_level_solver method solve the sea level equation until. Beacause of the iterative type of the resolution of the SLE, this method define also a first guess of the Sea level at the first iteration and the first time step. This function is based on the convergence iteration for the estimation of the variability defined in (*Convergence parameter* p.12).

Attribute

load
[(*LOAD_TIME_GRID* p.40) class object] The load time grid as specified in the class object. This grid needs to be of the same shape (maxdeg) then the one of the current object.

ice_time_grid
[(*ICE_TIME_GRID* p.33) class object] The ice time grid as specified in the class object. This grid needs to be of the same shape (maxdeg) then the one of the current object.

sed_time_grid
[(*SEDIMENT_TIME_GRID* p.31) class object] The sediment time grid as specified in the class object. This grid needs to be of the same shape (maxdeg) then the one of the current object.

love_number
[(*LOVE* p.45) class object] The love numbers as specified in the class object. The love numbers must have been set up with the same maximm degree thne the currend object.

TO
[(*sphericalobject* p.20) class object] The ocean contours variability area computed as a sphericalobject class computed for the previous iteration. !Trouver où définir ce calcul!.

t_it
[int] The time iteration of the current computation on wich apply the resolution of the SLE.

conv_it
[int] convergence iteration set to 0 if it's for a simple resolution of the SLE on one time step. This is used when you are working on a topographic convergence. In the code, the first guess for the SLE will be if it's not the first topographic convergence iteration, the guess of the previuous one.

conv_lim
[float] To stop the convergence of the solution, the conv_lim is usually set to 10^-3. The number of required step is then between 13 and 7.

Return

None

SL_C0de.grid.OCEAN_TIME_GRID.**sea_level_equation**(*self*, *load*, *ice_time_grid*, *sed_time_grid*, *love_number*, *TO*, *t_it*)

The sea_level_equation method calculate the Sea level variation following the SLE. Tis function is resolving both the conservation of mass equation and the SL variation. This follows the method described in (*Resolution of SLE including the deconvolution* p.11).

Attribute

load
[(*LOAD_TIME_GRID* p.40) class object] The load time grid as specified in the class object. This grid needs to be of the same shape (maxdeg) then the one of the current object.

ice_time_grid
[(*ICE_TIME_GRID* p.33) class object] The ice time grid as specified in the class object. This grid needs to be of the same shape (maxdeg) then the one of the current object.

sed_time_grid
[(*SEDIMENT_TIME_GRID* p.31) class object] The sediment time grid as specified in the class object. This grid needs to be of the same shape (maxdeg) then the one of the current object.

love_number
[(*LOVE* p.45) class object] The love numbers as specified in the class object. The love numbers must have been set up with the same maximm degree thne the currend object.

TO
[(*sphericalobject* p.20) class object] The ocean contours variability area computed as a sphericalobject class computed for the previous iteration. !Trouver où définir ce calcul!.

t_it
[int] The time iteration of the current computation on wich apply the resolution of the SLE.

Return

None

TOPOGRAPHIC_TIME_GRID

CLASS

class SL_C0de.grid.**TOPOGRAPHIC_TIME_GRID**(*time_step=array([1, 2])*, *maxdeg=64*, *height_time_grid=None*, *mass_time_grid=None*, *mass_time_coeff=None*, *height_time_coeff=None*, *rho=0*, *grid_name='time_grid'*, *from_file=(False,)*)

The TOPOGRAPHIC_TIME_GRID class is used to save and include all the topographic variations. This class inherits of (*TIME_GRID* p.25). This class main difference with TIME_GRID is the presence of a parameter called topo_pres wich is the present topography. It is created using (*Precomputation* p.15).

Attributes

time_step
[np.array([time_step_number,])] This array contains the time step of the data you are importing. They will be use for temporal interpolation.

maxdeg
[int] Maximum harmonic coefficient degree of the data. this define the chape of the grid and coefficient arrays

height_time_grid
[np.array([maxedg*2,maxdeg])] This array is the height grid at each time steps defined in grid_time_step

mass_time_grid
[np.array([maxedg*2,maxdeg])] This array is the mass grid at each time steps defined in grid_time_step

height_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the height spherical harmonic coefficient at each time steps defined in grid_time_step

mass_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the mass spherical harmonic coefficient at each time steps defined in grid_time_step

rho
[float] The density of the considered layer.

Note: In future development the density may vary threw space and time. We'll have to make a variable object more then a constant density.

---

grid_name
[str] The name of the grid. We recommand you to choose a specific name for each grid you create. This name is used to save the grid in an nc file with **`save`**_.

from_file
[(bool,way)] This parameter define if the data are new or loaded from a previously saved model in a nc file. If the first element is False, the code will create a blank object, based on provided datas. If the first element is True, the method will get the data from the file way specified in the second element of this attribute.

Methods

(*save* p.40)
Method to save the topographic datas

METHODS

SL_C0de.grid.TOPOGRAPHIC_TIME_GRID.**save**(*self*, *save_way=''*)

The save method is used to save the data of the topographic grid. Particularity of the topography is the present day topography used in the code to converge toward it. The function save is. Otherwise this method use the super method save to save the rest of the data.

Attributes

save_way :str
The way where the nc file is saved. Default value is the current file (an empty str).

Return

None

LOAD_TIME_GRID

CLASS

class SL_C0de.grid.**LOAD_TIME_GRID**(*sdelL=array([], dtype=float64)*, *beta_l=array([], dtype=float64)*, *E=array([], dtype=float64)*, *a=7371000*, *Me=5000*, *time_step=array([1, 2])*, *maxdeg=64*, *height_time_grid=None*, *mass_time_grid=None*, *mass_time_coeff=None*, *height_time_coeff=None*, *rho=0*, *grid_name='time_grid'*, *from_file=(False,)*)

The LOAD_TIME_GRID class is used to save and include all the topographic variations. This class inherits of (*TIME_GRID* p.25) and LOAD.

Attributes

sdelL
[np.array([time_step_number,maxdeg,maxdegx2])] The load variation grid used to compute earth vertical motion.

betal
[np.array([time_step_number,time_step_number,maxdeg])] The beta love number as described in (*Variation of geoïd and ground Equations* p.8) section. There calculated in the (*LOVE* p.45) class.

E
[np.array([(maxdeg+1)(maxdeg+2)/2,])] The elastic component of the earth as love numbers computed form (*LOVE* p.45) class.

a
[float] The earth radius in meter, set by default to 7371000 meters.

Me
[float] The earth mass in set by default to 5000.

time_step
[np.array([time_step_number,])] This array contains the time step of the data you are importing. They will be use for temporal interpolation.

maxdeg
[int] Maximum harmonic coefficient degree of the data. this define the chape of the grid and coefficient arrays

height_time_grid
[np.array([maxedg*2,maxdeg])] This array is the height grid at each time steps defined in grid_time_step

mass_time_grid
[np.array([maxedg*2,maxdeg])] This array is the mass grid at each time steps defined in grid_time_step

height_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the height spherical harmonic coefficient at each time steps defined in grid_time_step

mass_time_coeff
[np.array([(maxdeg+1)(maxedg+2)/2,])] This array is the mass spherical harmonic coefficient at each time steps defined in grid_time_step

rho
[float] The density of the considered layer.

grid_name
[str] The name of the grid. We recommand you to choose a specific name for each grid you create. This name is used to save the grid in an nc file with **`save`**_.

from_file
[(bool,way)] This parameter define if the data are new or loaded from a previously saved model in a nc file. If the first element is False, the code will create a blank object, based on provided datas. If the first element is True, the method will get the data from the file way specified in the second element of this attribute.

Methods

(*calc_viscuous* p.42):
Compute the viscous motion of the geoïd and ground for one time step.

(*calc_viscuous_time* p.42) :
Compute the viscous ground motion of the earth on all time steps.

(*calc_elastic_time* p.42) :
Compute the elastic ground motion of the earth on all time steps.

(*save* p.43) :
Save the load data

Methods

SL_C0de.grid.LOAD_TIME_GRID.**calc_viscuous**(*self*, *sdelL*, *beta*, *t_it*)

The calc_viscuous method is used to calculate the ground and geoïd deformation based on viscuous love numbers.

Attribute

sdelL
[np.array([t_it,(maxdeg+1)(maxdeg+20/2)])] The load grid used to estimate the ground vertical mouvement. This include all previous loading history because of the viscous comportment of earth.

beta
[np.array([time_step_number,time_step_number,(maxdeg+1)(maxdeg+2)/2])] The beta love numbers used to compute the earth deformation to include the viscous part. These love numbers are particularly heavy in the memory due to the representation of the time.

t_it
[int] The time iteration at wich the computation is performed.

SL_C0de.grid.LOAD_TIME_GRID.**calc_viscuous_time**(*self*, *backend=False*)

The calc_viscuous_time method compute the vicuous vertical ground motion. This method call the LOAD method for this.

Attribute

backend
[bool] Specifie if the method give backend (True) or not (False). Default is False.

Return

None

SL_C0de.grid.LOAD_TIME_GRID.**calc_elastic_time**(*self*)

The calc_elastic_time method compute the elactic vertical ground motion. This method call the LOAD method for this.

Attribute

None

Return

None

SL_C0de.grid.LOAD_TIME_GRID.**save**(*self*, *save_way=''*)

The save method is used to save the data from the class. It is based on the inherited (*save* p.31) method. Because of the particularity of this TIME_GRID, we had to save the new parameters, and calculated data. This function save for each data the real and complex part of the data due to the nc file particularity. The saved data are, The load (load), the viscuous groud motion (viscous_deformation), the elastic ground motion (elastic_deformation), the elastic love numbers (elastic_love), earth radius (a), earth mass (Me).

Attribute

save_way
[str] file path to where the grid will be saved.

Return

None

SL_C0de.grid.LOAD_TIME_GRID.**clean_memory**(*self*)

This method is used to clean the memory to avoïd over charging RAM.

1.4.4 love

This module is used to manage the love numbers for the earth deformation calculation.

Functions

SL_C0de.love.**love_lm**(*num*, *maxdeg*)

the love_lm funtion get from love numbers the h_lm spherical coefficient.

Attribute

num
[np.array([n,])] Love number coefficient of the size of the entry file

maxdeg
[int] The maximum harmonic coefficient degree.

Returns

h_lm
[np.array([(maxdeg+1)(maxdeg+2)/2,])] Array of the love number repeated on harmonic degree orders.

SL_C0de.love.**get_tlm**(*maxdeg*, *a*, *Me*)

The get_lm function generate the T spherical harmonic coefficient as defined in (*Theory* p.9).

Attribute

maxdeg
[int] maximum degree of spherical harmonic defined in the model parameters.

a
[float] The earth radius in meters.

Me
[float] The earth mass.

Returns

T_lm
[np.array([(maxdeg+1)(maxdeg+2)/2])] The T harmonic coefficient

---

SL_C0de.love.**calc_beta_counter**(*self*, *maxdeg*)

The calc_beta_counter define the degree of the spherical harmonic for each beta coefficient

Attribute

self
[(*LOVE* p.45) class object] The LOVE class object on wich the betacounter is calculated

maxdeg
[int] maximum spherical harmonic coefficient

Returns

None

CLASS

class SL_C0de.love.**LOVE**(*maxdeg*, *way*, *time_step*, *a*, *Me*, *type='time'*)

The LOVE class is used to keep the love numbers values and prepare them for the computation of geoïd and ground vertical motion. The love number are calculted and loaded from a file as described in (*Implementation of Love numbers* p.45). This function also include the possibility to compute the love numbers from normal modes love numbers parameters.

Attributes

maxedeg
[int] The spherical harmonic maximum degree.

way
[str] The file path to the ALMA output file. This file must follow the described pattern in ...

time_step
[np.array([time_step_number,])] The time step of the model, used to prepare the viscuous love numbers.

a
[float] The earth radius in meter.

Me
[float] The earth mass.

type
[str] The type of love number in input. could be 'time' or 'normal, where 'time' is for love numbers from ALMA3 code and 'normal' is for love number in normale mode from MIT serveur. Default is 'time'.

Methods

`calc_beta_G`_ :
This method is used to calculate the beta love numbers associated to the geoïd.

`calc_beta_R`_ :
This method is used to calculate the beta love numbers associated to the ground motion.

`calc_beta`_ :
This method is used to calculate the beta love number associated to both geoïd and ground.

(*clean_memory* p.45) :
This method is used to clean the memory of your computer.

Methods

SL_C0de.love.LOVE.**clean_memory**(*self*)

The clean_memory method can be used to araise the beta_l, beta_R and beta_G to avoid memory issues.

Attribute

None

Return

None

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[1] A. V. Dalca, K. L. Ferrier, J. X. Mitrovica, J. T. Perron, G. A. Milne, and J. R. Creveling. On postglacial sea level—III. Incorporating sediment redistribution. *Geophysical Journal International*, 194(1):45–60, July 2013. doi:10.1093/gji/ggt089.

[2] Ken L. Ferrier, Jacqueline Austermann, Jerry X. Mitrovica, and Tamara Pico. Incorporating sediment compaction into a gravitationally self-consistent model for ice age sea-level change. *Geophysical Journal International*, 211(1):663–672, October 2017. doi:10.1093/gji/ggx293.

[3] Roblyn A. Kendall, Jerry X. Mitrovica, and Glenn A. Milne. On post-glacial sea level - II. Numerical formulation and comparative results on spherically symmetric models. *Geophysical Journal International*, 161(3):679–706, June 2005. doi:10.1111/j.1365-246X.2005.02553.x.

[4] Augustus Love. *A Treatise on the Mathematical Theory of Elasticity*. Volume 1. Cambridge, 1892.

[5] D Melini, C Saliby, and G Spada. On computing viscoelastic Love numbers for general planetary models: the ALMA3 code. *Geophysical Journal International*, 231(3):1502–1517, August 2022. doi:10.1093/gji/ggac263.

[6] Glenn A. Milne and Jerry X. Mitrovica. Postglacial sea-level change on a rotating Earth. *Geophysical Journal International*, 133(1):1–19, April 1998. doi:10.1046/j.1365-246X.1998.1331455.x.

[7] J. X. Mitrovica and W. R. Peltier. Pleistocene deglaciation and the global gravity field. *Journal of Geophysical Research: Solid Earth*, 94(B10):13651–13671, October 1989. doi:10.1029/JB094iB10p13651.

[8] W. R. Peltier. The impulse response of a Maxwell Earth. *Reviews of Geophysics*, 12(4):649, 1974. doi:10.1029/RG012i004p00649.

[9] W. R. Peltier, D. F. Argus, and R. Drummond. Space geodesy constrains ice age terminal deglaciation: The global ICE-6G_C (VM5a) model: Global Glacial Isostatic Adjustment. *Journal of Geophysical Research: Solid Earth*, 120(1):450–487, January 2015. doi:10.1002/2014JB011176.

Index