Master agreement 14026

# Functions definition and specification of the Geophysical/Environmental Correction and Orbitography Module

| | |
|---|---|
| Reference: | CLS-GECO-1-09 |
| Nomenclature: | / |
| Issue: | 10.2 |
| Date: | 2024,Apr.23 |


**CLS** headquarters
11 rue Hermès
Parc technologique du Canal
31520 Ramonville Saint-Agne
FRANCE

Tel. +33 (0)5 61 39 47 00
Fax +33 (0)5 61 75 10 14
Mail: info@cls.fr
www.cls.fr

CLS Brest Le Ponant
Zone du technopôle de Brest Iroise
Avenue La Pérouse
29280 Plouzané
FRANCE

Tel. 33 (0)2.98.05.76.80
Fax 33 (0)2.98.05.76.90

# Chronology Issues:

| Issue: | Date: | Reason for change: | Author |
|---|---|---|---|
| 0.1 | 1 March 2018 | Initial version (preliminary) | G. Bracher |
| 0.4 | 23 April 2018 | Updates for PKPV | G. Bracher |
| 0.5 | 30 April 2018 | Addition of grid error algorithm spec | G. Bracher |
| 1.0 | 22 May 2018 | Updates for Orbital Altitude Rate | G. Bracher |
| 1.1 | 31 May 2018 | Improve the datetime tutorial<br>CNES comments taken into account (clarification) | G. Bracher<br>V. Claverie |
| 2.0 | 18 Jul. 2018 | Initial version for GECO v0.2.0 | G. Bracher |
| 2.1 | 11 Sept. 2018 | Taking into account PKPV v0.2.0 remarks. | G. Bracher |
| 2.2 | 21 Sept. 2018 | Quality comments taken into account | G. Bracher |
| 2.3 | 02 Oct. 2018 | Update the ocean tide algorithm's interface | G. Bracher |
| 2.4 | 22 Oct. 2018 | Updates the document with CNES comments during the v0.2.0 acceptance phase. | G. Bracher |
| 2.5 | 9 Nov. 2018 | Update SSB documentation about outliers values. | G. Bracher |
| 3.0 | 13 Feb. 2019 | Initial version for GECO v0.3.0 | G. Bracher |
| 3.1 | 29 Mar. 2019 | CNES comments at PKPV v0.3.0 taken into account | G. Bracher |
| 4.0 | 11 Dec. 2019 | Release 0.5.0: all interfaces.<br>Orbitography functions have been set apart in a new document (CLS-GECO-2-09). | G. Bracher |
| 6.0 | 24 Mar. 2020 | Updates for degrees interpolation in MFWAM and ice concentration interpolation. | G. Bracher |
| 6.1 | 2020 June 08 | Correction of release and date in header. | V. Claverie |
| 7.0 | 14 Oct. 2020 | Updates for v0.7.0:<br>- Mathematical statement updated<br>- Rain flag<br>- Time management section<br>- Box adds 2 degrees to bounds | G. Bracher |
| 7.1 | 8 Dec. 2020 | Mathematical statement of rain flag was missing.<br>Updates ocean atmospheric attenuation algorithm (Lillibridge).<br>Updates dry tropospheric (ocean) interface.<br>Adds pseudo-code of grid TEC interpolator. | G. Bracher |
| 7.2 | 4 Mar. 2021 | Time management section refers to 09b document | G. Bracher |
| 8.0 | July 23, 2021 | Updates for geco v0.8.0: sad and dad used by algorithm added<br>New interfaces for dry/wet 3D. | G. Bracher |

| Issue: | Date: | Reason for change: | Author |
|---|---|---|---|
| | | Special treatment over land/ocean added for ocean tide. | |
| 9.0 | 2022,Jan.14 | S1S2 was missing in auxiliary data requirement for dry/wet hydro and sig0.<br>XCal: Baseline is divided into left and right. New algorithm to compute flags.<br>Inverted barometer: Sea surface pressure must be corrected from s1s2 ECMWF climatology (but s1s2 air tide model should not be added). Floating ice surfaces should be considered in the compute of mean sea pressure. | G. Bracher |
| 9.1 | 16 Sep. 22 | SWOT-FT-699 : GECO v1.0.1 | G. Bracher |
| 9.2 | 9 May. 2023 | Timing left and right are in wrong order in xcal interface.<br>Patch 1.0.3:<br>- FT-738 “Loading of TEC map in geco”: TEC map of Day+1 is needed to interpolate between 23h and 00h00 of Day D. | G. Bracher |
| 10.0 | May 22, 2023 | FT-1021 (geco v2.0.0) : time performance improvement:<br>Meteo reader has changed (name and header to include)<br>Only one function to compute dry, wet and atmospheric attenuation | G. Bracher |
| 10.1 | August 2, 2023 | FT-1115: New pole tide method added: compute components.<br>MDT computed by bilinear interpolation.<br>Oceanic and internal tides computed by pyfes library.<br>Ice flag algorithm changes. | G. Bracher |
| 10.2 | April 23, 2024 | FT-1179: GECO v2.2.0:<br>- XCal reader can take list of files in input.<br>- XCal LR algorithm can take a time varying cross-track vector.<br>- S1s2 climatology reader can take a folder in input.<br>- New TEC reader available, non-sensitive to the name. | G. Bracher |

## People involved in this issue:

| | | |
|---|---|---|
| Written by: | Bracher Geoffroy | 2024,Apr.23 Geoffroy Bracher |
| Checked by: | V. Claverie | 2024,Apr.23 [Checker] Vincent Claverie |
| Approved by: | L. Amarouche | 2024,Apr.23 [Approver] L. Amarouche |
| Application authorized by: | | |

## Index Sheet:

| | |
|---|---|
| Context: | |
| Keywords: | [Mots clés ] |
| Hyperlink: | |

## Distribution:

| Company | Means of distribution | Names |
|---|---|---|
| CNES | Electronic file | N Picot, C Garcia, M-L Frery |

# List of tables and figures

## List of tables:

**Aucune entrée de table d'illustration n'a été trouvée.**

## List of figures:

# List of Contents

# 1. Presentation

The purpose of this document is to present the different functions of the GECO module, to describe their interfaces and their algorithm baseline. The pseudo-code of the functions is not detailed here, see the Doxygen documentation for further information (generated at compilation, cf. the SUM, in the **docs/docs/html** folder, the entry point file is **index.html**).

The orbitography functions are set apart in the CLS-GECO-2-09 document.

## 1.1. General considerations

From a high point of view, the functions are distributed into two main categories: the readers and the algorithms. They both have classes with attributes and methods, but they don't work in the same way.

The readers class read and load the data immediately at instantiation.

The Algorithms class has one or many methods to compute the associated correction.

So:

- all the reader's actions are done in one unique step: instantiation.
- all the algorithms' actions are done in two steps: instantiation + interpolation.

That's why only the constructor interface is described for the readers; for the algorithms, the interpolation methods are also described.

Some functions parameters are optional, they can be identified when the column "default value" of the interface table description isn't empty, when this column is empty, the parameter is mandatory.

For each algorithm the "**fill_value**" parameter allows the users to set their own DV. (cf. classes interfaces below).

When the outputs units of the algorithm aren't specified, it means it has same units as the input data.

## 1.2. About masked data

Some algorithms (such as bathymetry for example) in the legacy software bibli-alti formalism used to only consider values of some type of surface (like "ocean surface" for example) in the interpolation to compute the correction. This strategy is relevant with a point per point approach, but not with a vector approach.

Within the context of the GECO module it is still possible to apply a mask on the data, but it isn't done by the algorithm, it is left to the user choice, which made the GECO module more flexible. Cf. the following code to see an example of how to mask data, it is divided in two steps:

1 - getting the mask

2 - applying the mask

```
#include "geco/algorithm/surface/surface_type.hpp"

// We have a matrix of data named MSS_values and we want to mask it.

// 1 - first we need to load the mask data
auto mask_data = grid::DiscreteXY<int8_t>(SURFACE_TYPE, "mask");
// and to estimate it at position (the same position as for the MSS_values matrix)
Eigen::Matrix<surface::SurfaceType::Kind, -1, 1> mask_values(6);
auto mask_interpolator = surface::SurfaceType(mask_data);
mask_values = mask_interpolator.compute(lon, lat);

// 2 - Now, supposing we only want ocean values, we apply the mask with the help of
```

```
// the eigen lib: (Boolean test on mask).select(what if true, what if false)
auto masked_MSS =
(mask_values.array()==surface::SurfaceType::Kind::kOpenOcean).select(MSS_values,
std::numeric_limits<double>::quiet_NaN());
```

See https://eigen.tuxfamily.org/dox/classEigen_1_1Select.html for further information.

## 1.3. List of readers and algorithms per theme

| Theme | Reader | Algorithm | Algorithm's methods |
|---|---|---|---|
| **Atmospheric attenuation (Ocean)** | grib::ReducedGaussianXYList<br>grid::SampledClimatologyDataSet<float><br>tide::s1s2::WaveModels<float> | environment::Lillibridge | atmospheric_attenuation |
| **Atmospheric attenuation (Hydro)** | grib::ReducedGaussianXYList<br>grib::ReducedGaussianXYZList<br>grid::SampledClimatologyDataSet<float><br>tide::s1s2::WaveModels<float> | environment::AtmosphericAttenuation3D | compute |
| **Bathymetry** | grid::DiscreteXY<int16_t> | surface::Elevation | compute |
| **Distance to coastline** | grid::DiscreteXY<int16_t> | surface::DistanceToCoast | compute |
| **Dry tropospheric correction (Ocean)** | grib::ReducedGaussianXYList<br>grid::SampledClimatologyDataSet<float><br>tide::s1s2::WaveModels<float> | environment::AtmosphericSurface | dry_tropospheric |
| **Dry and Wet tropospheric corrections (Hydro)** | grib::ReducedGaussianXYList<br>grib::ReducedGaussianXYZList<br>grid::SampledClimatologyDataSet<float><br>tide::s1s2::WaveModels<float> | environment::Meteo3D | compute_dry_wet |

| | | | |
|---|---|---|---|
| **Dynamic Atmospheric Correction** | grid::GridsList<short> | environment::DAC | compute |
| **Earth tidal prediction** | N/A | tide::earth::CartwrightTayler | compute |
| **Geoid** | grid::DiscreteXY<float_t> | surface::Geoid | compute |
| **Grid's accuracy** | grid::DiscreteXY<int32_t> | surface::GridError | compute |
| **Ice map** | grid::OceanAndSeaIceSAFList | surface::IceMap | compute |
| **Ice flag** | N/A | surface::IceMap | ice_flag_duacs |
| **Internal tide** | tide::ocean::InternalM2RayZaron<br>tide::ocean::InternalHRET | tide::ocean::Internal<float> | compute |
| **Inverted Barometer** | grib::ReducedGaussianXYList<br>grid::SampledClimatologyDataSet<float><br>grid::DiscreteXY<int8_t> | environment::InvertedBarometer | compute |
| **Ionospheric delay along track** | binary::GIM | environment::GIM | compute_mono<br>compute_dual |
| **Ionospheric delay map** | ascii::TotalElectronContents | environment::GlobalIonosphericMap | compute |
| **Mean Dynamic Topography** | grid::DiscreteXY<int32_t> | surface::MeanDynamicTopography | compute |
| **Mean Sea Surface (*)** | grid::DiscreteXY<int32_t> | surface::MeanSeaSurface | compute |
| **MF WAM** | grib::MeshMfWamMp2<br>grib::MeshMfWamMwd | environment::Meteorology | compute |
| **Ocean tidal prediction (**)** | tide::ocean::Configuration<br>tide::ocean::ModelData<float> | tide::ocean::ModelHandler | compute |
| **Pole tidal prediction** | tide::pole::WaveModels<float><br>ascii::PolePosition | tide::pole::Desai | compute |
| **Rain flag** | NA | environment::RainRate | flag |
| **Rain rate** | grib::ReducedGaussianXYList | environment::RainRate | compute |

| | | | |
|---|---|---|---|
| **Sea State Bias** | grid::Scattered<float> | environment::SeaStateBias | compute |
| **Surface Type** | grid::DiscreteXY<int8_t> | surface::SurfaceType | compute |
| **Sea surface pressure** | grib::ReducedGaussianXYList | environment::Meteorology | compute |
| **Sea Wave Height** | grib::ReducedGaussianXYList | environment::Meteorology | compute |
| **Wet tropospheric corr (Ocean)** | grib::ReducedGaussianXYList | environment::Meteorology | compute |
| **Wet tropospheric corr (Hydro)** | Cf. Dry tropospheric correction (hydro) | | |
| **Wind meridional component** | grib::ReducedGaussianXYList | environment::Meteorology | compute |
| **Wind zonal component** | grib::ReducedGaussianXYList | environment::Meteorology | compute |
| **Xcal error** | binary::Xcal | interpolation::Xcal | height_error<br>baseline_errors |

**Table 1. Readers and Algorithms associated to all themes**

Note:

- (*) From an algorithm point of view, there is no distinction between mean sea surface solution 1 and 2 but the sad itself. So, both solutions use the same reader and algorithm.

- (**) The Ocean Tidal Compute solution 1 and 2 have same reader and algorithm, the differences only come from the sad files, see the section 2.21 for further information.

## 1.4. Auxiliary data nomenclature per theme

The following table describes the static and dynamic auxiliary data nomenclature associated to each theme (according to sad delivered in geco_sad item and dad used for validation in geco_val_data item).

| Theme | Auxiliary data nomenclature |
|---|---|
| **Atmospheric attenuation (Ocean)** | Surface pressure : SMM_PSA_AXV***<br>2 meters temperature: SMM_T2M_AXP***<br>Total column water vapour: SMM_CWV_AXP***<br>Total cloud liquid water content: SMM_CLW_AXP***<br>s1s2_surface_pressure_climatology_ECMWF_YYYYMMDDTHHMMSS_vXYZ/ |

| | |
|---|---|
| | s1s2_air_tide_model_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Atmospheric attenuation (Hydro)** | Surface pressure : SMM_PSA_AXV***<br>Geopotential: SMM_ORA_AXV***<br>Temperature profiles: SMM_T3A_AXV***<br>Specific humidity profiles: SMM_Q3A_AXV***<br>Specific cloud liquid water content profiles: SMM_L3A_AXV***<br>s1s2_surface_pressure_climatology_ECMWF_YYYYMMDDTHHMMSS_vXYZ/<br>s1s2_air_tide_model_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Dry tropospheric correction (Hydro)**<br>**Wet tropospheric corr (Hydro)** | Surface pressure : SMM_PSA_AXV***<br>Geopotential: SMM_ORA_AXV***<br>Temperature profiles: SMM_T3A_AXV***<br>Specific humidity profiles: SMM_Q3A_AXV***<br>s1s2_surface_pressure_climatology_ECMWF_YYYYMMDDTHHMMSS_vXYZ/<br>s1s2_air_tide_model_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Bathymetry** | depth_or_elevation_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Distance to coastline** | distance_to_coast_1m_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Dry tropospheric correction (Ocean)** | Sea surface pressure: SMM_PMA_AXV***<br>s1s2_surface_pressure_climatology_ECMWF_YYYYMMDDTHHMMSS_vXYZ/<br>s1s2_air_tide_model_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Dynamic Atmospheric Correction** | SMM_MOG_AXV*** |
| **Earth tidal prediction** | NA |
| **Geoid** | geoid_egm2008_wgs84_meantide_min1x1_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Ice map** | SMM_IC[NS]_AXV*** |
| **Internal tide (M2 only)** | internal_tide_m2_ray_zaron_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Internal tide (HRET)** | internal_tide_HRET_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Inverted Barometer** | Sea surface pressure: SMM_PMA_AXV***<br>surface_type_7states_YYYYMMDDTHHMMSS_vXYZ.nc<br>s1s2_surface_pressure_climatology_ECMWF_YYYYMMDDTHHMMSS_vXYZ/ |
| **Ionospheric delay along track (GIM)** | SWOT_ION_AXV*** |
| **Ionospheric delay map (TEC)** | JPLQ***.16I |
| **Mean Dynamic Topography (CLS13)** | mean_dynamic_topography_cnes_cls_2013_YYYYMMDDTHHMMSS_vXYZ.nc |

| | |
|---|---|
| **Mean Dynamic Topography (CLS18)** | mean_dynamic_topography_cnes_cls_2018_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Mean Sea Surface (DTU15)** | mean_sea_surface_dtu_2015_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Mean Sea Surface (CLS15)** | mean_sea_surface_cnes_cls_2015_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Mean Sea Surface (DTU18)** | mean_sea_surface_dtu_2018_YYYYMMDDTHHMMSS_vXYZ.nc |
| **MF WAM** | SMM_WMA_AXV*** |
| **Ocean tidal prediction (FES)** | oceantide_fes2014_YYYYMMDDTHHMMSS_vXYZ/ |
| **Ocean tidal prediction (GOT)** | oceantide_got410_10ondes_YYYYMMDDTHHMMSS_vXYZ/ |
| **Pole tidal prediction** | SMM_POL_AXX***<br>pole_tide_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Rain rate** | SMM_CRR_AXF***<br>SMM_LSR_AXF*** |
| **Sea State Bias** | sea_state_bias_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Surface Type (4 states mask)** | surface_type_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Surface Type (7 states mask)** | surface_type_7states_YYYYMMDDTHHMMSS_vXYZ.nc |
| **Sea Wave Height** | SMM_SWH_AXV*** |
| **Wet tropospheric corr (Ocean)** | SMM_WEA_AXV*** |
| **Wind zonal and meridional component** | SMM_[UV]WA_AXV*** |
| **Xcal error** | SWOT_INT_LR_XoverCal_***.nc |

**Table 2. Auxiliary data per theme**

# 2. Description of functions interfaces

## 2.1. Time management in geco

A dedicated library has been developed to manage time variable in the GECO module.

See the dedicated section in document 09b-Orbito_Functions_definition_and_specification-GECO.

## 2.2. About advanced reading mode

To limit the memory usage when loading the data, some readers have an advanced reading mode which only load the region covering the required inputs positions.

This advanced mode is activated by providing a “box” which defines the region of interest. So, this box must be defined before calling the readers.

The object include file is:

```
include/geco/geometry/box.hpp
```

The “box” class is:

```
geometry::Box
```

It needs latitude and longitude positions to define the region of interest, there is two ways to create the box: providing the latitude and longitude bounds or giving the latitude and longitude vectors, as shown in the following example:

```
#include “geco/geometry/box.hpp”
#include “geco/geometry/point.hpp”

namespace geometry = geco::geometry;

Eigen::VectorXd latitude(4), longitude(4);
longitude << -1., -2., -3., -4. ;
latitude << 10., 20., 30., 40. ;

double longitude_min = min(longitude) ;
double longitude_max = max(longitude) ;
double latitude_min  = min(latitude) ;
double latitude_max  = max(latitude) ;
geometry::point::SphericalEquatorial min_coord(longitude_min, latitude_min);
geometry::point::SphericalEquatorial max_coord(longitude_max, latitude_max);

// giving the bound
geometry::Box box(min_coord, max_coord);
// giving vectors
geometry::Box box(longitude, latitude);
```

**Important note**: The box automatically extends bounds in latitude and longitude (min and max) up to 2 degrees in order to anticipate the needs of interpolator such as bilinear, spline etc...

Example of reader without the advanced mode:

```
auto data = grid::DiscreteXY<int32_t>(MEANSEASURFACE, “mss”);
```

and with the advanced mode (using the “box”):

```
auto data = grid::DiscreteXY<int32_t>(MEANSEASURFACE, “mss”, box);
```

Note, when using the box option, if an interpolation is performed outside the box (at latitude 10° for example here), the compute cannot work, and the algorithm will return the default value.

Once this “box” is defined, it can be used and re-used for each reader function, so it doesn’t need more than one instantiation.

The readers which have advanced reading mode are those with the “box” parameters in their constructor interface description.

It is highly recommended to use this advanced reading mode to compute wet/dry tropospheric correction and attenuation atmospheric for hydrology.

## 2.3. Atmospheric attenuation (Ocean)

The atmospheric attenuation computed over ocean is deduced from the following ECMWF fields:

- Surface pressure (sp, ECMWF indicatorOfParameter = 134)
- 2 meters temperature (2t, ECMWF indicatorOfParameter = 167)
- Total column water vapour (tcwv, ECMWF indicatorOfParameter = 137)
- Total cloud liquid water content (tclw, ECMWF indicatorOfParameter = 78)

The reader used to load these data is the same, see the following section “ECMWF readers”.

It also needs static auxiliary data to correct the surface pressure from s1s2 effect.

### 2.3.1. Readers

#### 2.3.1.1. The S1S2 pressure climatology

Cf. 2.7.1.2

#### 2.3.1.2. The S1S2 climatological pressure model

Cf. 2.7.1.3

#### 2.3.1.3. ECMWF Readers

Include file: geco/io/grib/reduced_gaussian_xy_list.hpp

Class name: ReducedGaussianXYList

Role: Loading into memory a series of grids stored in GRIB format.

Note: The list of grids does not have to be ordered chronologically (the constructor sorts the grids in chronological order).

Warning: It is the user’s responsibility to ensure, if necessary, that the time difference between two grids constituting the time series is constant.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Meshes** | std::list<std::string> | The list of grids to store in memory. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

### 2.3.2. Algorithm

Include file: geco/algorithm/environment/meteo.hpp

#### 2.3.2.1. Class name: Lillibridge

Role: To compute the atmospheric attenuation over ocean for Ka or Ku bands (default Ka), according to *One- and Two-Dimensional Wind Speed Models for Ka-Band Altimetry Lillibridge, Scharroo, 2014.*

Constructor interface:

| Parameter | type | Description | Default value (if optional) |
|---|---|---|---|
| **meshes_sp** | interpolation::ReducedGaussianXYList | The time series to be taken into account for interpolation (must contain surface pressure field) | |
| **s1s2_climatology** | interpolation::SampledClimatologyDataSet< float > | S1/S2 climatology based on ECMWF pressure fields generated every 6 hours. | |
| **S1s2_model** | tide::s1s2::WaveModels< float > | S1/S2 Tide Model. | |
| **Meshes_2t** | interpolation::ReducedGaussianXYList | The time series to be taken into account for interpolation (must contain 2 meters temperature field) | |
| **meshes_tcwv** | interpolation::ReducedGaussianXYList | The time series to be taken into account for interpolation (must contain Total column water vapor field) | |
| **meshes_clwv** | interpolation::ReducedGaussianXYList | The time series to be taken into account for interpolation (must contain Cloud liquid water content field) | |
| **bound_error** | bool | If true, when interpolated values are requested outside of the domain of the input mesh, an exception is raised. If false, then fill_value is used. | false |
| **neighbors** | size_t | The number of nearest neighbors to be used for | 4 |

| | | | |
|---|---|---|---|
| | | calculating the interpolated value. | |
| **fill_value** | double | The value to use if the request point is outside of the interpolation domain. | NaN |

Note: If the number of neighbors is set to 1, then the calculation method will return the value of the nearest neighbor, otherwise the value will be interpolated using the "inverse distance weighted" method.

### 2.3.2.2. Process method: compute

Role: According to Lillibridge et al. 2014, the atmospheric attenuation is a combination of three components (coefficients below are for Ka band):

- The one-way dry atmospheric attenuation is empirically fit with a linear function of pressure Psurf and temperature Tsurf:

$$\delta\sigma_{\text{dry}} = 0.310\text{-}0.593\,\frac{P_{\text{surf}}}{1013} - 0.499\,\frac{288.15}{T_{\text{surf}}} + 0.956\,\frac{P_{\text{surf}}}{1013}\,\frac{288.15}{T_{\text{surf}}}$$

- The wet component of the one-way backscatter attenuation is fit as a quadratic function of total precipitable water w:

$$\delta\sigma_{\text{wet}} = 7.21\ 10^{-3}\ w + 4.43\ 10^{\text{-}5}\ w^2$$

- The third component is a linear function of liquid cloud water L:

$$\delta\sigma_{\text{rain}} = 1.070\ L$$

These algorithms are functions of four two-dimensional (surface or total atmosphere) meteorological parameters: sea surface pressure (Psurf, in hPa) corrected from s1s2 (if s1s2_cor parameter is set to true), 2 meters temperature (Tsurf, in K)), total column water vapor (w, in $kg.m^{-2}$), and cloud liquid water content (L, in $kg\ m^{-2}$). The ECMWF model is used for these four parameters.

The sum of dry, wet, and liquid water attenuations needs to be multiplied by 2 to account for the round-trip radar path to be added to the uncorrected backscatter.

$$\Delta\sigma = 2 \times (\delta\sigma_{\text{dry}} + \delta\sigma_{\text{wet}} + \delta\sigma_{\text{rain}})$$

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **dates** | geco::datetime::utc::Array | time coordinates (UTC) | |
| **longitude** | Eigen::VectorXd | The measurement longitude (in degrees). | |

| | | | |
|---|---|---|---|
| **Latitude** | Eigen::VectorXd | The measurement latitude (in degrees). | |
| **S1s2_cor** | bool | If true, the surface pressure will be corrected from s1s2. If false, surface pressure will not be corrected and kept as provided. | True |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |
| **Band** | environment::FrequencyBand | The band corresponding to the instrument's frequency (can be "Ka" or "Ku", default is "Ka"). | Ka |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| type | description |
|---|---|
| **Eige::VectorXd** | A vector that contains the backscatter coefficient (Sigma0 in dB) atmospheric attenuation for the specified frequency. |
| **Eigen::VectorXi** | number of points used when calculating the interpolated value |

### 2.3.3. Implementation example

```
#include "geco/algorithm/environment/meteo.hpp"

namespace env = geco::algorithm::environment;

int main(int argc, char** argv) {
  if (argc != 9) {
    std::cerr << "Usage: " << argv[0]
              << "<former_PS> <later_PS> <former_2T> <later_2T> "
                 "<former_TCWV> <later_ TCWV> <former_CLWV> <later_CLWV>"
                 "<climato_folder> <s1s2_air_tide_model>"
              << std::endl;
    return 1;
  }
```

```
  // defining the required readers
  // s1s2 files for correction are needed
  auto path = boost::filesystem::path{argv[9]};
  auto files = std::list<std::string>{};
  auto netcdf = std::string(".nc");
  for (auto &item : boost::filesystem::directory_iterator(path)) {
    auto path = item.path().string();
    if (std::equal(netcdf.rbegin(), netcdf.rend(), path.rbegin())) {
      files.emplace_back(item.path().string());
    }
  }
  auto s1s2_model =
      grid::SampledClimatologyDataSet<float>(files, "surface_pressure");
  auto s1s2_climatology = tide::s1s2::WaveModels(argv[10]);
  auto mesh_ps = geco::io::grib::ReducedGaussianXYList({argv[1], argv[2]});
  auto mesh_2t = geco::io::grib::ReducedGaussianXYList({argv[3], argv[4]});
  auto mesh_tcwv = geco::io::grib::ReducedGaussianXYList({argv[5], argv[6]});
  auto mesh_clwv = geco::io::grib::ReducedGaussianXYList({argv[7], argv[8]});

  // input parameters: lon, lat, psurf, height and time
  auto start = geco::datetime::UTC(2015, 7, 21, 2);
  auto step = geco::datetime::TimeDelta(0, 900, 0);
  auto end = geco::datetime::UTC(2015, 7, 21, 5);
  size_t size = (end - start) / step;
  size_t ix = 0;
  Eigen::VectorXd lon(size);
  Eigen::VectorXd lat(size);
  Eigen::VectorXd height(size);
  auto dates = geco::datetime::utc::Array(size);
  for (auto item = start; item < end; item += step) {
    lat[ix] = 88.5;
    lon[ix] = 13.2;
    height[ix] = 1.8;
    dates[ix++] = item;
  }

  // Perform algorithm
  Eigen::VectorXd att;
  auto interp = geco::algorithm::environment::Lillibridge(
      mesh_ps, s1s2_model, s1s2_climatology, mesh_2t, mesh_tcwv, mesh_clwv);
  std::tie(att, samples) = interp.compute(dates, lon, lat, true, 0,
FrequencyBand::Ka);
  std::cout << att << std::endl;
  return 0;
}
```

## 2.4. Atmospheric attenuation (Hydro)

The atmospheric attenuation computed for hydrology is deduced from the following ECMWF fields:

- Ps: the surface pressure (ECMWF indicatorOfParameter = 134)
- Z: the geopotential (ECMWF indicatorOfParameter = 129)
- T: the multi-level temperature (ECMWF indicatorOfParameter = 130)
- q : the multi-level specific humidity (ECMWF indicatorOfParameter = 133)
- l : the multi-level specific cloud liquid water content (ECMWF indicatorOfParameter = 246)

These fields need two different kinds of readers.

It also needs static auxiliary data to correct the surface pressure from s1s2 effect.

### 2.4.1. Readers

#### 2.4.1.1. Reader for geopotential, and surface pressure

Include file: geco/io/grib/reduced_gaussian_xy_list.hpp

Class name: ReducedGaussianXYList

Role: Loading into memory a series of grids stored in GRIB format.

Note: The list of grids does not have to be ordered chronologically (the constructor sorts the grids in chronological order).

Warning: It is the user's responsibility to ensure, if necessary, that the time difference between two grids constituting the time series is constant.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Meshes** | std::list< std::string > | The list of grids to store in memory. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

#### 2.4.1.2. Reader for s1s2 correction

Cf. 2.3.1.1 and 2.3.1.2

#### 2.4.1.3. Reader for humidity, temperature, and cloud liquid water content

Include file: geco/io/grib/reduced_gaussian_xyz_list.hpp

Class name: ReducedGaussianXYZList

Role: Loading into memory a series of N-level grids stored in GRIB format.

Note: The list of grids does not have to be ordered chronologically (the constructor sorts the grids in chronological order).

Warning: It is the user's responsibility to ensure, if necessary, that the time difference between two grids constituting the time series is constant.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Meshes** | std::list< std::string > | The list of N-level grids to store in memory. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

## 2.4.2. Algorithm

Include file: geco/algorithm/environment/meteo3d.hpp

### 2.4.2.1. Class name: Meteo3D

Role:

To interpolate a time series of meteorological fields represented by a multi-level reduced Gaussian grid stored in a GRIB file. The ECMWF fields required are:

- Surface pressure (ps, ECMWF indicatorOfParameter = 134)
- Geopotential (z, ECMWF indicatorOfParameter = 129)
- Specific humidity (q, ECMWF indicatorOfParameter = 133)
- Temperature (t, ECMWF indicatorOfParameter = 130)
- Optional: Specific cloud liquid water content (clwc, ECMWF indicatorOfParameter = 246)

and then to compute the wet and dry tropospheric correction at measurement altitude. The atmospheric attenuation is also computed if the Specific cloud liquid water content have been provided.

Constructor interface:

| Parameter | type | Description | Default value (if optional) |
|---|---|---|---|
| **meshes_sp** | interpolation::ReducedGaussianXYList | The time series to be taken into account for interpolation (must contain surface pressure field) | |
| **s1s2_climatology** | interpolation::SampledClimatologyDataSet< float > | S1/S2 climatology based on ECMWF pressure fields generated every 6 hours. | |
| **S1s2_model** | tide::s1s2::WaveModels< float > | S1/S2 Tide Model. | |
| **Mesh_xy_gpot** | interpolation::ReducedGaussianXYList | The time series of geopotential to be taken into account for interpolation. (ECMWF: z, 129) | |
| **mesh_xyz_q** | interpolation::ReducedGaussianXYZList | The time series of multi-level humidity to be taken into account for interpolation. (ECMWF: q, 133) | |

| | | | |
|---|---|---|---|
| **mesh_xyz_t** | interpolation::ReducedGaussianXYZList | The time series of multi-level temperature to be taken into account for interpolation. (ECMWF: t, 130) | |
| **mesh_xyz_l** | interpolation::ReducedGaussianXYZList | The time series of multi-level cloud liquid water content to be taken into account for interpolation. (ECMWF: clwc, 246) | Boost::none |
| **bound_error** | bool | If true, when interpolated values are requested outside of the domain of the input mesh, an exception is raised. If false, then fill_value is used. | false |
| **neighbors** | size_t | The number of nearest neighbors to be used for calculating the interpolated value. | 4 |
| **fill_value** | double | The value to use if the request point is outside of the interpolation domain. | NaN |

Note: If the number of neighbors is set to 1, then the calculation method will return the value of the nearest neighbor, otherwise the value will be interpolated using the “inverse distance weighted” method.

### 2.4.2.2. Process method: compute

Role: to compute the dry/wet tropospheric correction at measurement altitude, and the atmospheric attenuation backscatter coefficient on Ka-band (if Specific cloud liquid water content files have been provided) using a vertical integration of the meteo 3D parameters.

The dry tropospheric correction can be expressed as a function of the pressure:

$$\delta h_{dry} = -77.6 \times 10^{-6} \frac{R}{M_d} \int_{P_{surf}}^{P_{sat}} \frac{1}{g} . dP$$

The wet tropospheric correction can be expressed as a function of the pressure:

$$\delta h_{wet} = -23.7 \times 10^{-6} \frac{R}{M_w} \int_{P_{surf}}^{P_{sat}} \frac{1}{g} . q . dP - 0.375 \frac{R}{M_w} \int_{P_{surf}}^{P_{sat}} \frac{1}{g} . \frac{q}{T} . dP$$

For detailed steps that lead to this equation see the ANNEXE 1 at the end of this document.

This algorithm uses multi-level data from the ECMWF meteorological model fields to computes the dry tropospheric correction w.r.t. the real surface height. These data are defined over 137 pressure levels that can be converted into height levels (see below).

In term of level pressure, the previous equations give:

$$\delta h_{dry} = -77.6 \times 10^{-6} \frac{R}{M_d} \sum_{L=1}^{\substack{L\ above \\ real\ height}} \frac{1}{g_L} . \left(P_{L+1/2} - P_{L-1/2}\right) + \delta_{contribution}$$

And:

$$\delta h_{wet} = -23.7 \times 10^{-6} \frac{R}{M_w} \sum_{L=1}^{\substack{L\ above \\ real\ height}} \frac{1}{g_L} . q_L \left(P_{L+1/2} - P_{L-1/2}\right)$$
$$- 0.375 \times 10^{-6} \frac{R}{M_w} \sum_{L=1}^{\substack{L\ above \\ real\ height}} \frac{1}{g_L} . \frac{q_L}{T_L} \left(P_{L+1/2} - P_{L-1/2}\right) + \delta_{contribution}$$

With:

- **R**: the gas constant = 8.31434 J/mole.K
- **Md**: Molar mass of dry air = 0.0289644 kg/mole
- $\boldsymbol{g_L}$ : The gravity of the layer L (calculation details are described below)
- $\boldsymbol{P_{L+\frac{1}{2}}}$ : The pressure of the half level L + ½
- $\boldsymbol{\delta_{contribution}}$ : The remaining contribution between the real height and the layer above the real height (calculation details are described below).
- **L above real height**: The last layer above the real surface height that fully contributes to the sum. See the Figure 1 for ECMWF multi-level model description.

Note: The compute of the half level pressures, the gravity/height of each layer and the corrected (real) surface pressure is explained below.

Finally:

$$\boldsymbol{\delta_{contribution}} = -77.6 \times 10^{-6} \frac{R}{M_d} . \frac{1}{g_N} . \left(\boldsymbol{P_{real}} - P_{N-1/2}\right)$$

And for the wet tropospheric correction:

$$\boldsymbol{\delta_{contribution}} = -23.7 \times 10^{-6} \frac{R}{M_w} . \frac{1}{g_N} Q_N \left(\boldsymbol{P_{real}} - P_{N-1/2}\right)$$
$$- 0.375 \times 10^{-6} \frac{R}{M_w} . \frac{1}{g_N} . \frac{Q_N}{\boldsymbol{T_{mean}}} \left(\boldsymbol{P_{real}} - P_{N-1/2}\right)$$

With:

$\boldsymbol{N}$ = L above real height

$\boldsymbol{T_{mean} = (T_{surface} + T_N)/2}$ if N = 137
$\boldsymbol{T_{mean} = T_N}$ if N < 137
$\boldsymbol{P_{real}}$ The surface pressure corrected from height and from s1s2 if the apply_s1s2_correction **input parameter** has been set to true.

If the Specific cloud liquid water content has been provided, the atmospheric attenuation at measurement altitude is also computed, else, Nan is returned.

The atmospheric attenuation can be computed as a combination of dry, wet, and liquid water attenuations (Lillibridge 2014).

The one-way dry atmospheric attenuation is empirically fit with a linear function of pressure and temperature:

$$\delta\sigma_{dry} = 0.310\text{-}0.593\frac{P_{surf}}{1013} - 0.499\frac{288.15}{T_{surf}} + 0.956\frac{P_{surf}}{1013}\frac{288.15}{T_{surf}}$$

The wet component of the one-way backscatter attenuation is fit as a quadratic function of total precipitable water:

$$\delta\sigma_{wet} = 7.21\ 10^{-3}\ w + 4.43\ 10^{-5}\ w^2$$

The third component is a linear function of liquid cloud water:

$$\delta\sigma_{rain} = 1.070\ L$$

The sum of dry, wet, and liquid water attenuations needs to be multiplied by 2 to account for the round-trip radar path to be added to the uncorrected backscatter:

$$\delta\sigma = 2.(\delta\sigma_{dry} + \delta\sigma_{wet} + \delta\sigma_{rain})$$

With:

$$w = \int_{P_{surf}}^{P_{sat}} \frac{q}{g}.dP$$

$$L = \int_{P_{surf}}^{P_{sat}} \frac{l}{g}.dP$$

- q : the multi-level specific humidity (**input data**)

- l : the multi-level specific cloud liquid water content (**input data**)

- Tsurf: the temperature of the lowest layer of the multi-level temperature (**input data**)

- Psurf: the “real” surface pressure corrected from height and from s1s2 if the apply_s1s2_correction **input parameter** has been set to true (see below for detailed compute).

This algorithm uses multi-level data from the ECMWF meteorological model fields to computes the atmospheric attenuation w.r.t. the real surface height. These data are defined over 137 pressure levels that can be converted into height levels (see below).

In term of level pressure, the previous equation gives:

$$w = \sum_{L=1}^{L\ above\ real\ height} \frac{q_L}{g_L}.\left(P_{L+1/2} - P_{L-1/2}\right) + \delta w_{contribution}$$

$$L = \sum_{L=1}^{L\ above\ real\ height} \frac{l_L}{g_L}.\left(P_{L+1/2} - P_{L-1/2}\right) + \delta L_{contribution}$$

With:

- $\boldsymbol{g_L}$ : The gravity of the layer L (calculation details are described below)
- $\boldsymbol{P_{L+\frac{1}{2}}}$ : The pressure of the half level L + ½
- $\boldsymbol{\delta_{contribution}}$ : The remaining contribution between the real height and the layer above the real height (calculation details are described below).
- **L above real height**: The last layer above the real surface height that fully contributes to the sum. The following figure described the ECMWF levels model:

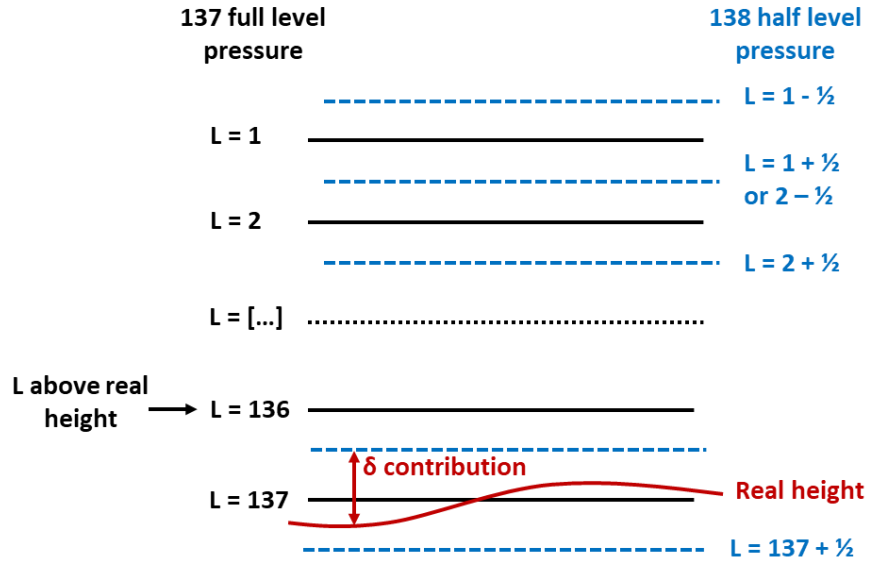


**Figure 1. The ECMWF L137 model level**

The level L = 137 + ½ corresponds to the **orography (input data)**.

The half level pressure is deduced from the coefficient a(L) and b(L) that define the level:

$$P_{L+1/2} = a_{L+1/2} \,/\, 100 \; + b_{L+1/2}.\boldsymbol{P_{surface}}$$

With:

- $\boldsymbol{a_L}$ and $\boldsymbol{b_L}$ : The coefficients defining the model levels (contained in the ECMWF input data).
- $\boldsymbol{P_{surface}}$: the surface pressure **(input parameter)** corrected from S1S2 if the apply_s1s2_correction **input parameter** has been set to true, but not from the height at this step.

The gravity $\boldsymbol{g_L}$ of each layer is computed such as:

$$\boldsymbol{g_L} = g_{surfEllips}\left[1 - 2\frac{\boldsymbol{h_L}}{\boldsymbol{R_\varphi}} + 3\frac{\boldsymbol{h_L^2}}{SemiMajorAxis^2}\right]$$

With:

- $\boldsymbol{h_L}$ The height of the full level L such as:

$$\boldsymbol{h_L} = \frac{\boldsymbol{R_\varphi}.\ \boldsymbol{Geopot_L}}{\frac{g_{surfEllips}.\boldsymbol{R_\varphi}}{g_0} - \boldsymbol{Geopot_L}}$$

- $SemiMajorAxis$ The semi major axis of ellipsoid, depends on the system (**optional input parameter**), default is WGS84 = 6,378,137.0 meters
- $g_0$ the normal gravity at mean sea level and 45° latitude (WMO definition) = 9.80665 m/s2
- $\boldsymbol{R_\varphi}$ the ellipsoid ray at a given latitude (m) such as:

$$\boldsymbol{R_\varphi} = \frac{SemiMajorAxis}{\left(1.006803 - 0.006706 \times sin^2(\boldsymbol{\varphi})\right)}$$

- $\boldsymbol{\varphi}$ the geodetic latitude (**input parameter**)

- $g_{surfEllips}$ the gravity at the surface of the ellipsoid (in m/s2) such as

$$g_{surfEllips} = \frac{g_{equator}(1 + K_s\, sin^2(\boldsymbol{\varphi}))}{\sqrt{1 - ecc^2.sin^2(\boldsymbol{\varphi})}}$$

- $g_{equator}$ The equatorial gravity = 9.7803253359 m/s2
- $K_s$ The Somigliana's constant = 0.001931853
- $ecc^2$ the eccentricity square (depends on the system, default is WGS84) = 0.00669438

- $\boldsymbol{Geopot_L}$ the geopotential of the layer L such as:

$$\boldsymbol{Geopot_L} = Geopot_{L+1/2} + 29.291 \times \boldsymbol{T_{virtual}}.log\left(2\frac{\boldsymbol{P_{L+1/2}}}{\boldsymbol{P_{L-1/2}} + \boldsymbol{P_{L+1/2}}}\right)$$

$$Geopot_{L-1/2} = Geopot_{L+1/2} + 29.291 \times \boldsymbol{T_{virtual}}.log\left(\frac{\boldsymbol{P_{L+1/2}}}{\boldsymbol{P_{L-1/2}}}\right)$$

$$\boldsymbol{T_{virtual}} = \boldsymbol{T_L}\left(\frac{1 + 1.608\ \boldsymbol{Q_L}}{1 + \boldsymbol{Q_L}}\right)$$

- $Geopot_{137+1/2}$ the orography = Surface Geopotential (**input data**) / $g_0$
- $\boldsymbol{Q_L}$ and $\boldsymbol{T_L}$ respectively the specific humidity (**input data**) and the temperature (**input data**) at layer L.

The compute of the remaining $\delta_{contribution}$ depends on where is located the real surface height $\boldsymbol{h_{real}}$ (**input parameter**) within the layers, which leads to correct the surface pressure from height $\boldsymbol{P_{real}}$, there are three cases:

| Case | L above real height = | $P_{real}$ = |
|---|---|---|
| $h_{real} > h_{1/2}$ | NA | NA → $\delta_{contribution} = 0$ |
| $h_{137+1/2} < h_{real} < h_{1/2}$ | N such as: $h_{N+1/2} < h_{real} < h_{N-1/2}$ | $P_{N-1/2} + \frac{(P_{N+1/2} - P_{N-1/2})}{(h_{N+1/2} - h_{N-1/2})} . (h_{real} - h_{N-1/2})$ |
| $h_{real} < h_{137+1/2}$ | 137 | $P_{surface} \left(1 + \frac{\gamma}{T_{surf}} (h_{real} - h_{137+1/2})\right)^{-\frac{g_0}{R_d \gamma}}$ |

With:

- $R_d$ the specific gas constant for dry air = 287.058 J/K/Kg
- $\gamma$ the mean vertical gradient of temperature = 0.0065K/m
- $P_{surface}$ the surface pressure corrected from height and from s1s2 if the apply_s1s2_correction input parameter has been set to true.
- $T_{surf}$ the surface temperature of the ECMWF model such as:

$$T_{surf} = T_{137} + \gamma(h_{137+1/2} - h_{137})$$

Finally:

$$\delta w_{contribution} = \frac{q_N}{g_N} . (P_{real} - P_{N-1/2})$$

$$\delta L_{contribution} = \frac{l_N}{g_N} . (P_{real} - P_{N-1/2})$$

Hypothesis: the gravity, the humidity, and the liquid cloud water of the lowest layer are supposed to be valid below this layer.

These computes are realized for all the selected neighbors in space and time, then results are interpolated such as:

- Bilinear interpolation in space (inverse distance weighting method)
- Linear interpolation in time.

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **t** | datetime::utc::Array | time coordinates (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees. | |
| **Lat** | Eigen::VectorXd | Latitude in degrees. | |

| | | | |
|---|---|---|---|
| **Real_height** | Eigen::VectorXd | The real surface Height (meters) usually H=SatelliteAltitude−TrakerRange−Geoid | |
| **system** | geometry::geodetic::System | WGS system used to calculate the geodetic positions (default is WGS84) | WGS84 |
| **apply_s1s2_correction** | bool | If true, the surface pressure will be corrected from s1s2. If false, surface pressure will not be corrected and kept as provided. | True |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |
| **band** | environment::FrequencyBand | To determine the frequency of the instrument. Required for atmospheric attenuation (default is Ka band). | Ka |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXd, Eigen::VectorXd, Eigen::VectorXi>

| Type | description |
|---|---|
| **Eigen::VectorXd** | the dry tropospheric correction at measurement altitude (in meters) |
| **Eigen::VectorXd** | the wet tropospheric correction at measurement altitude (in meters) |
| **Eigen::VectorXd** | the interpolated atmospheric attenuation values (in dB). NaN if cloud liquid water content has not been provided. |
| **Eigen::VectorXi** | number of points used when calculating the interpolated value |

### 2.4.3. Implementation example

```
#include "geco/algorithm/environment/meteo3d.hpp"
#include "geco/io/grib/reduced_gaussian_xy_list.hpp"
#include "geco/io/grib/reduced_gaussian_xyz_list.hpp"
#include <Eigen/Core>
#include <iostream>

int main(int argc, char** argv) {
  if (argc != 9) {
    std::cerr << "Usage: " << argv[0]
              << "<former_Q3D> <later_Q3D> <former_T3D> <later_T3D> "
                 "<former_gpot> <later_gpot> <former_L3D> <later_L3D>"
```

```
                "<former_psurf> <later_psurf> <climato_folder> <s1s2_model>"
             << std::endl;
    return 1;
  }

  // defining the required readers
  // s1s2 files for correction are needed
  auto path = boost::filesystem::path{argv[11]};
  auto files = std::list<std::string>{};
  auto netcdf = std::string(".nc");
  for (auto &item : boost::filesystem::directory_iterator(path)) {
    auto path = item.path().string();
    if (std::equal(netcdf.rbegin(), netcdf.rend(), path.rbegin())) {
      files.emplace_back(item.path().string());
    }
  }
  auto s1s2_model =
      grid::SampledClimatologyDataSet<float>(files, "surface_pressure");
  auto s1s2_climatology = tide::s1s2::WaveModels(argv[12]);

  auto mesh_gpot = geco::io::grib::ReducedGaussianXYList({argv[5], argv[6]});
  auto mesh_psurf = geco::io::grib::ReducedGaussianXYList({argv[9], argv[10]});
  auto mesh_Q3D = geco::io::grib::ReducedGaussianXYZList({argv[1], argv[2]});
  auto mesh_T3D = geco::io::grib::ReducedGaussianXYZList({argv[3], argv[4]});
  auto mesh_L3D = geco::io::grib::ReducedGaussianXYZList({argv[7], argv[8]});

  // input parameters: lon, lat, psurf, height and time
  auto start = geco::datetime::UTC(2015, 7, 21, 2);
  auto step = geco::datetime::TimeDelta(0, 900, 0);
  auto end = geco::datetime::UTC(2015, 7, 21, 5);
  size_t size = (end - start) / step;
  size_t ix = 0;
  Eigen::VectorXd lon(size);
  Eigen::VectorXd lat(size);
  Eigen::VectorXd height(size);
  auto dates = geco::datetime::utc::Array(size);
  for (auto item = start; item < end; item += step) {
    lat[ix] = 88.5;
    lon[ix] = 13.2;
    height[ix] = 1.8;
    dates[ix++] = item;
  }

  // Perform algorithm
  Eigen::VectorXd att;
  auto interp = geco::algorithm::environment::Meteo3D(
      mesh_psurf, s1s2_model, s1s2_climatology, mesh_gpot, mesh_Q3D, mesh_T3D,
mesh_L3D);
  std::tie(att, samples) = interp.compute_dry_wet_attatm(dates, lon, lat, height);
  std::cout << att << std::endl;
  return 0;
}
```

## 2.5. Bathymetry

### 2.5.1. Reader

Include file: geco/io/discrete.hpp

Class name: DiscreteXY<int16_t>

Role: To read the netCDF file containing the grid information. Note that the data will be stored as integer. This class load the whole map by default, it also includes the advance reading option described in the chapter above (cf. 2.1).

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Path** | std::string | Path to the netCDF grid to open. | |
| **Name** | std::string | netCDF variable that represents the discrete grid. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | Geometry::Box()<br>Which means that all the content is stored. |

## 2.5.2. Algorithm

Include file: geco/algorithm/surface/elevation.hpp

### 2.5.2.1. Class name: Elevation

Role: To compute the ocean/land depth/elevation at given point by bilinear interpolation. If a point or more is missing, the interpolation is still done with the other valid points (for example if there is only one valid point, the returned value is the value of this point, thus the algorithm works as a nearest algorithm). The number of points used is also returned by the compute method. Note that this method is automatically multi-threaded (can be deactivated).

Constructor interface:

| Parameter | type | Description | Default value (if optional) |
|---|---|---|---|
| **mesh** | geometry::grid::DiscreteXY<int32_t> | Mesh representing the global relief on the earth's surface. | |
| **Fill_value** | double | The value to use if the request point is outside of the interpolation domain. | NaN |

### 2.5.2.2. Process method: compute

Role: Interpolate over a 2-D grid at the request point in three steps:

1. Select the cell of the four surrounding values of the input point (x,y)
2. determine the weight of each point of the cell according to the following formula:

$$Weight(x,y)=(lx-dx)/lx*(ly-dy)/dy$$

Where:

lx=abs(cell(0,0).x()−cell(1,0).x())

dx=abs(cell(0,0).x()−point.x())

ly=abs(cell(0,0).y()−cell(0,1).y())

dy=abs(cell(0,0).y()−point.y())

5. Compute the interpolate value according to the following formula:

value=sum(Weight(x,y)∗cell(x,y).value())

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **lon** | Eigen::VectorXd | Longitude in degrees. | |
| **Lat** | Eigen::VectorXd | Latitude in degrees. | |
| **Num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std ::tuple< Eigen::VectorXd, Eigen ::VectorXi>

| type | description |
|---|---|
| **Eigen::VectorXd** | interpolated values at input coordinates |
| **Eigen::VectorXi** | number of points used when calculating the interpolated value |

### 2.5.3. Implementation example

```
#include "geco/algorithm/surface/elevation.hpp"
#include "geco/io/grid/discrete.hpp"

namespace grid = geco::io::grid;
namespace surface = geco::algorithm::surface;

auto data = grid::DiscreteXY<int16_t>(ELEVATION, "elevation");

Eigen::VectorXd values;
Eigen::VectorXi 24imples;
Eigen::VectorXd lon(1);
Eigen::VectorXd lat(1);

lon << 130.;
lat << 0.;

auto interpolator = surface::Elevation(data);
std::tie(values, samples) = interpolator.compute(lon, lat);
```

```
// example changing the optional parameters
auto interpolator = surface:: Elevation(data, fill_value=9999999);
std::tie(values, samples) = interpolator.compute(lon, lat, num_threads=2);
```

## 2.6. Distance to coastline

### 2.6.1. Reader

Include file: geco/io/discrete.hpp

Class name: DiscreteXY<int16_t>

Role: To read the netCDF file containing the grid information. Note that the data will be stored as integer. This class load the whole map by default, it also includes the advance reading option described in the chapter above (cf. 2.1).

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Path** | std::string | Path to the netCDF grid to open. | |
| **Name** | std::string | netCDF variable that represents the discrete grid. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | Geometry::Box()<br>Which means that all the content is stored. |

### 2.6.2. Algorithm

Include file: geco/algorithm/surface/shorelines.hpp

#### 2.6.2.1. Class name: DistanceToCoast

Role: To determine the distance to the coast at given position. The nearest of the four surrounding grid points is taken. Note that this method is automatically multi-threaded (can be deactivated).

Constructor interface:

| Parameter | type | Description | Default value (if optional) |
|---|---|---|---|
| **mesh** | geometry::grid::DiscreteXY<int32_t> | Mesh representing the distance to coast on the earth's surface. | |
| **Fill_value** | double | The value to use if the request point is outside of the interpolation domain. | NaN |

### 2.6.2.2. Process method: compute

Role: Determine over a 2-D grid the nearest neighbors in three steps:

1. Select the cell of the four surrounding values of the input point (x,y)
2. determine the weight of each point of the cell according to the following formula:

Weight(x,y)=(lx−dx)/lx∗(ly−dy)/dy

Where:

lx=abs(cell(0,0).x()−cell(1,0).x())

dx=abs(cell(0,0).x()−point.x())

ly=abs(cell(0,0).y()−cell(0,1).y())

dy=abs(cell(0,0).y()−point.y())

5. Select the nearest point which corresponds to the highest weight.

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **lon** | Eigen::VectorXd | Longitude in degrees. | |
| **Lat** | Eigen::VectorXd | Latitude in degrees. | |
| **Num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std ::tuple< Eigen::VectorXd Eigen ::VectorXi>

| type | description |
|---|---|
| **Eigen::VectorXd** | Determined values at input coordinates |
| **Eigen::VectorXi** | number of points used when selecting the nearest value |

## 2.6.3. Implementation example

```
#include "geco/algorithm/surface/shorelines.hpp"
#include "geco/io/grid/discrete.hpp"

namespace grid = geco::io::grid;
namespace surface = geco::algorithm::surface;

auto data = grid::DiscreteXY<int16_t>(DISTANCE, "distance");
```

```
Eigen::VectorXd values;
Eigen::VectorXi 27imples;
Eigen::VectorXd lon(1);
Eigen::VectorXd lat(1);

lon << 130.;
lat << 0.;

auto interpolator = surface::DistanceToCoast(data);
std::tie(values, samples) = interpolator.compute(lon, lat);

// example changing the optional parameters
auto interpolator = surface::DistanceToCoast(data, fill_value=9999999);
std::tie(values, samples) = interpolator.compute(lon, lat, num_threads=2);
```

## 2.7. Dry Tropospheric Correction (Ocean)

Note that dry tropospheric correction estimation requires three different kind of data, so three reader classes are described in the readers section. To be computed, this correction also needs an intermediary parameter which is the sea surface pressure corrected from the S1S2 pressure, so the algorithm section also contains the description of this process.

### 2.7.1. Readers

#### 2.7.1.1. The atmospheric Sea surface Pressure (ECMWF)

Include file: geco/io/grib/reduced_gaussian_xy_list.hpp

Class name: ReducedGaussianXYList

Role: Loading into memory a series of grids stored in GRIB format.

Constructor interface:

| Parameter | Type | Description | Default value (if optional) |
|---|---|---|---|
| **meshes** | std::list< std::string > | The list of grids to store in memory. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

#### 2.7.1.2. The S1S2 pressure climatology

Include file: geco/io/grid/sampled_climatology.hpp

Class name: SampledClimatologyDataSet<float>

Role: This object allows to load in memory a monthly climatology of a physical variable represented at regular intervals every 6 hours by a numerical grid.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **Path** | std::list< std::string > | The list of NetCDF grids defining the climatology to read. | |
| **Name** | std::string | The NetCDF variable to read | |
| **Box** | geometry::Box | Geographic area to be loaded into memory | geometry::Box()<br>Which means that all the content is stored |

Or

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **Directory_path** | std::string | The directory containing the list of NetCDF grids defining the climatology to read. | |
| **Name** | std::string | The NetCDF variable to read | |
| **Box** | geometry::Box | Geographic area to be loaded into memory | geometry::Box()<br>Which means that all the content is stored |

### 2.7.1.3. The S1S2 climatological pressure model

Include file: geco/io/tide/s1s2.hpp

Class name: WaveModels<float>

Role: Loads the Ray & Ponte climatological model. This model includes only diurnal (S1) and semidiurnal (S2) periods.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **Path** | std::string | Path to the file containing the S1S2 tide model to load into memory | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

## 2.7.2. Algorithm

Include file: geco/algorithm/environment/atmospheric_surface_pressure.hpp

### 2.7.2.1. Class name: AtmosphericSurface

Role: To compute:

- the atmospheric surface pressure corrected by the atmospheric pressure S1S2 (resolved from S1S2 climatology figured out ECMWF pressure fields generated every 6 hours) and

- the dry tropospheric correction (corrected from S1S2).

Note: We recommend to provide a sea surface pressure list as pressure_fields parameter, not a surface pressure list.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **pressure_fields** | interpolation::ReducedGaussianXYList | The list of pressure fields to interpolate the pressure from the space-time coordinates provided during the calculation. | |
| **S1s2_climatology** | Interpolation::SampledClimatologyDataSet< float > | S1/S2 climatology based on ECMWF pressure fields generated every 6 hours. | |
| **S1s2_model** | tide::s1s2::WaveModels< float > | S1/S2 Tide Model. | |
| **Bound_error** | bool | If true, when interpolated values are requested outside of the domain of the input grid, an exception is raised. If false, then fill_value is used. | false |
| **fill_value** | double | The value to use if the request point is outside of the interpolation domain (implies bound_error false). | NaN |

### 2.7.2.2. Process method: pressure_s1s2_model

Role: Space-time calculation of the atmospheric surface pressure for the requested spatial time coordinates to which the climatological pressure is removed and a model of S1S2 pressure is added. The pressure is obtained by linear interpolation in time between two consecutive data files, and by bilinear interpolation in space from the four nearest model grid values. The climatological pressure to remove and the S1S2 model to add are obtained by the same procedure.

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **t** | datetime::utc::Array | time coordinates (UTC) | |
| **lon** | Eigen::VectorXd | Longitude indegrees | |
| **lat** | Eigen::VectorXd | Latitude in degrees | |
| **num_threads** | size_t | number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| Type | Description |
|---|---|
| **Eigen::VectorXd** | the corrected atmospheric surface pressure |
| **Eigen::VectorXi** | the minimum number of points used to perform spatial pressure and climatology interpolations S1S2 |

### 2.7.2.3. Process method: dry_tropospheric

Role: Space-time calculation of the dry tropospheric correction for the requested spatial time coordinates request. It is derived from the sea surface pressure according to the formula:

$$DryTropo = -2.27710e-5 * (PsurfCor) * [1 + 0.0026 * \cos(latitude * \pi / 90)]$$

Where PsurfCor (pressure parameter) is the atmospheric pressure as performed by the AtmosphericSurface::pressure_s1s2_model method.

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **Pressure** | Eigen::VectorXd > | Atmospheric sea surface pressure corrected for atmospheric tides S1/S2 as performed by the pressure_s1s2_model method. | |
| **Lat** | Eigen::VectorXd > | Latitude in degrees | |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| type | description |
|---|---|
| **Eigen::VectorXd** | The dry tropospheric correction at input coordinates in meter. |
| **Eigen::VectorXi** | number of points used when calculating the interpolated value |

### 2.7.3. Implementation example

```
#include "geco/algorithm/surface/atmospheric_surface_pressure.hpp"
#include "geco/io/grib/reduced_gaussian_xy_list.hpp"
#include "geco/io/grid/sampled_climatology.hpp"
#include "geco/io/tide/s1s2.hpp"
#include "geco/datetime/julian_day.hpp"
#include <boost/filesystem.hpp>

namespace environment   = geco::algorithm::environment;
namespace grib          = geco::io::grib;
namespace grid          = geco::io::grid;
namespace tide          = geco::io::tide;
namespace datetime      = geco::datetime;

// Reader part
// ----------------------------------------
// the data needed are:
//   - Pressure fields (ECMWF)    // to compute the pressure corrected from S1S2
//   - The S1S2 surface pressure  // (needed for dry tropo)
//   - The S1S2 modeled surface pressure  // to compute the dry tropo
auto gribs = std::list<std::string>();
gribs.push_back(FILE_00);
gribs.push_back(FILE_06);
gribs.push_back(FILE_12);
auto pressure_fields = grib::ReducedGaussianXYList(gribs);

auto path = boost::filesystem::path{S1S2_SAMPLED_CLIMATOLOGY};
auto files = std::list<std::string>{};
auto netcdf = std::string(".nc");
for (auto& item : boost::filesystem::directory_iterator(path)) {
  auto path = item.path().string();
  if (std::equal(netcdf.rbegin(), netcdf.rend(), path.rbegin())) {
    files.emplace_back(item.path().string());
  }
}
auto s1s2_climatology = grid::SampledClimatologyDataSet<float>(files,
"surface_pressure");

auto s1s2_model       = tide::s1s2::WaveModels(S1S2_CLIMATOLOGY);
// ----------------------------------------


// Algo part (GECO)
// ----------------------------------------
// Initialization
Eigen::VectorXd Psurf_corr;
Eigen::VectorXi samples;
auto date  = datetime::utc::Array(1);
auto date1 = datetime::utc::Array(1);
auto lon   = Eigen::VectorXd(1);
auto lat   = Eigen::VectorXd(1);

lon << 197.773373;
lat << 33.529607 ;
```

```
date << datetime ::UTC(2015, 7, 21, 5, 12, 32, 14) ;

auto AtmSurf = environment::AtmosphericSurface(pressure_fields,
                                               s1s2_climatology,
                                               s1s2_model);
std::tie(Psurf_corr, samples) = AtmSurf.pressure_s1s2_model(date, lon, lat, 1);
auto dry_tropo                = AtmSurf.dry_tropospheric(Psurf_corr,  lat);
```

## 2.8. Dry and Wet Tropospheric Correction (Hydro)

See 2.4

## 2.9. Dynamic Atmospheric Correction

### 2.9.1. Readers

Include file: geco/io/grid/grids_list.hpp

Class name: GridsList<short>

Role: Load into memory a time series consisting of numerical grids, stored in NetCDF files, representing a physical variable over time.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **grids** | std::list< std::string > | The list of grids to store in memory. | |
| **Name** | std::string | netCDF variable that represents the discrete grid | |
| **box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

### 2.9.2. Algorithm

Include file: geco/algorithm/environment/dynamic_atmospheric_correction.hpp

#### 2.9.2.1. Class name: DAC

Role: Dynamic Atmospheric Correction (DAC) parameter (sum of high frequency variability of the sea surface height and of the low frequency part of the inverted barometer effect) at the altimeter measurement are obtained by linear interpolation in time between two consecutive MOG2D model data files, and by bilinear interpolation in space from the four nearest model grid values.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **grids** | interpolation::GridsList< int16_t > | The time series to be taken into account for interpolation. | |
| **Fill_value** | double | The value to use if the request point is outside the time and space interpolation range. | NaN |

### 2.9.2.2. Process method: compute

Role: Space-time interpolation (as described in the class role) of the time series handled by this instance, for the requested spatial time coordinates request.

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **t** | datetime::utc::Array | time coordinates (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees | |
| **lat** | Eigen::VectorXd | Latitude in degrees | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| type | description |
|---|---|
| **Eigen::VectorXd** | the interpolated values at input coordinates. |
| **Eigen::VectorXi** | the minimum number of points used to perform spatial interpolation on one of the grids. |

## 2.9.3. Implementation example

```
#include "geco/io/grid/grids_list.hpp"
#include "geco/algorithm/environment/dynamic_atmospheric_correction.hpp"

namespace grid        = geco::io::grid;
namespace datetime    = geco::datetime;
namespace environment = geco::algorithm::environment;
```

```
// Read part
// ----------------------------------------
std::list<std::string> string_list{MOG2D_00, MOG2D_06, MOG2D_12};
auto result = grid::GridsList<short>(string_list, "Grid_0001");
// ----------------------------------------


// Algo part (GECO)
// ----------------------------------------
// Initialization
Eigen::VectorXd values;
Eigen ::VectorXi samples ;
Eigen ::VectorXd lon(1) ;
Eigen::VectorXd lat(1);
datetime::utc::Array date(1);

lon << 130. ;
lat << 0. ;
date << datetime ::UTC(2004, 12, 30, 3) ;

auto interpolator = environment::DAC(result);
// Compute the DAC interpolation
std ::tie(values, samples) = interpolator.compute(date, lon, lat) ;
```

## 2.10. Earth Tidal Prediction

### 2.10.1. Readers

The Earth Tidal Prediction theme does not have any reader. Indeed, all the parameters needed to compute this correction are part of the code.

### 2.10.2. Algorithm

Include file: geco/algorithm/tide/earth.hpp

#### 2.10.2.1. Class name: CartwrightTayler

Role: Calculate the earth tide using the Cartwright et Edden model.

Note that the constructor of this class has no arguments. Indeed, all the needed parameters (Cartwright and Edden tables of tide potential amplitudes, frequencies and phases of 5 astronomical variables and Love numbers) are included in of the GECO code, and instantly loaded at instantiation of this class. The height of the long equilibrium period ocean tide isn't computed here but in ocean tide algorithm (more accurate, cf. **Erreur ! Source du renvoi introuvable.**).

#### 2.10.2.2. Process method: compute

Role: The gravitational potential V induced by an astronomical body can be decomposed into harmonic constituents, each characterized by an amplitude, a phase and a frequency. Thus, the tide potential can be expressed as:

$$V = \sum_{n=2}^{\infty} \sum_{s} V_n(s) \qquad (1)$$

where the tide potential of constituents $V_n(s)$, is given by:

$$V_n(s) = g.c_n(s).\sum_{m=0}^{n} W_n^m.\cos[\omega(s).t + \phi(s) + m.\lambda] \leftarrow \text{for}(m+n) = \text{even}$$
$$V_n(s) = g.c_n(s).\sum_{m=0}^{n} W_n^m.\sin[\omega(s).t + \phi(s) + m.\lambda] \leftarrow \text{for}(m+n) = \text{odd} \quad (2)$$

where the phase $\omega(s).t + \phi(s)$ of constituents at altimeter time tag t (relative to the reference epoch), is given by a linear combination of the corresponding phases of the 6 astronomical variables $\omega_i.t + \phi_i$:

$$\omega(s).t + \phi(s) = \sum_{i=1}^{6} k_i(s).[\varpi_i.t + \phi_i] \quad (3)$$

where $\lambda$ is the altimeter longitude, where $W_m^n$ is the associated Legendre polynomial (spherical harmonic) of degree n and order m ( $W_m^n(\sin\theta)$, with $\theta$ altimeter latitude), and where g is gravity.

The Cartwright and Edden tables provide for degree n=2 and order m=0,1,2, and for degree n=3 and order m=0,1,2,3 the $k_i(s)$ coefficients and the amplitudes $c_n(s)$ for each constituent s (only amplitudes exceeding about 0.004 mm have been computed by Cartwright and Tayler (1971), and Cartwright and Edden (1973)). This allows for the potential to be computed.

The solid Earth tide height is proportional to the potential. The proportionality factors are the so-called Love number $H_n$ and $K_n$.

The solid Earth tide height H_solid is thus:

$$H\_Solid = H_2.\frac{V_2}{g} + H_3.\frac{V_3}{g} \quad (4)$$

with: $H_2 = 0.609$
$H_3 = 0.291$
$g = 9.80$
$V_2 = V_{20} + V_{21} + V_{22}$
$V_3 = V_{30} + V_{31} + V_{32} + V_{33}$

The above described tide contributions do not take into account the permanent tide.

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | time coordinates (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees for the position at which tide is computed | |
| **lat** | Eigen::VectorXd | Latitude in degrees (positive north) for the position at which tide is computed | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel | 0 |

| | | | |
|---|---|---|---|
| | | computing code is used at all, which is useful for debugging. | |

Parameters are returned as:

| type | description |
|---|---|
| **Eigen::VectorXd** | the earth tide height (m). |

### 2.10.3. Implementation example

```
#include "geco/algorithm/tide/earth.hpp"

namespace datetime = geco::datetime;
namespace geometry = geco::geometry;
namespace math = geco::detail::math;
namespace earth = geco::algorithm::tide::earth;

// Read part
// ----------------------------------------
// !!!!! No reader part for earth tide computation !!!!!
// ----------------------------------------


// Algo part (GECO)
// ----------------------------------------
// Initialization
auto date = datetime::utc::Array(1);
auto lon  = Eigen::VectorXd(1);
auto lat  = Eigen::VectorXd(1);

lon << 130.;
lat << 0.;
date << datetime::UTC(2015, 8, 20, 7, 50, 34, 329853);

auto model = earth::CartwrightTayler();
// Compute the nearest interpolation
auto res = model.compute(date, lon, lat, 1);
// ----------------------------------------
```

## 2.11. Geoid

### 2.11.1. Reader

Include file: geco/io/discrete.hpp

Class name: DiscreteXY<float_t>

Role: To read the netCDF file containing the grid information. Note that the data will be stored as long integer. This class load the whole map by default, but it includes also the advance reading option described in the chapter above (cf. 2.1).

Constructor interface:

| Parameter | Type | description | Default value |
|---|---|---|---|

| | | | (if optional) |
|---|---|---|---|
| **path** | std::string | Path to the netCDF grid to open. | |
| **Name** | std::string | netCDF variable that represents the discrete grid. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

## 2.11.2. Algorithm

Include file: geco/algorithm/surface/geoid.hpp

### 2.11.2.1. Class name: Geoid

Role: To compute the height of the geoid above the reference ellipsoid. The height of the geoid is computed at the altimeter measurement using a squared window of NxN geoid grid points (typically N = 6) centered on the altimeter point. Spline functions are calculated within the window as function of grid point latitude for each geoid column. Each of these spline functions is evaluated at the altimeter latitude. The resulting values are then used for calculating a spline function of grid point longitude. The height of the geoid is derived by evaluating the spline at the altimeter longitude. Note that this method is automatically multi-threaded (can be deactivated).

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **mesh** | geometry::grid::DiscreteXY<float_t> | Mesh representing the geoid. | |
| **Fill_value** | double | The value to use if the request point is outside of the interpolation domain (implies bound_error false). | NaN |

### 2.11.2.2. Process method: compute

Role: Evaluate the 2D spline at given points in three steps.

1. For each lat/lon point of the inputs lat/lon vectors: Select the frame (of size nx,ny) of values surrounding the input point.

2. For each latitude of the frame (i.e. ny times) select the corresponding longitude vector of values and interpolate the value by spline at input longitude value to obtain a vector of size (ny).

3. Then, interpolate by spline at input latitude value this vector.

Note: the spline is performed by the gsl_interp_eval function of the gsl library.

Process method interface:

| Parameter | type | Description | Default value (if optional) |
|---|---|---|---|
| **lon** | Eigen::VectorXd | longitudes in degrees | |
| **lat** | Eigen::VectorXd | latitudes in degrees | |
| **nx** | size_t | number of longitude values required to perform the spline interpolation. | 3 |
| **ny** | size_t | number of latitude values required to perform the spline interpolation | 3 |
| **num_thread** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std ::tuple< Eigen::VectorXd, Eigen ::VectorXi>

| type | description |
|---|---|
| **Eigen::VectorXd** | interpolated values at input coordinates |
| **Eigen::VectorXi** | number of points used when calculating the interpolated value |

### 2.11.3. Implementation example

```
#include "geco/algorithm/surface/geoid.hpp"
#include "geco/io/grid/discrete.hpp"

namespace grid = geco::io::grid;
namespace surface = geco::algorithm::surface;

auto data = grid::DiscreteXY<float_t>(GEOID, "geoid");

Eigen::VectorXd values;
Eigen::VectorXi samples;
Eigen::VectorXd lon(1);
Eigen::VectorXd lat(1);

lon << 130.;
lat << 0.;

auto interpolator = surface::Geoid(data);
std::tie(values, samples) = interpolator.compute(lon, lat);

// example changing the optional parameters
auto interpolator = surface:: Geoid(data, fill_value=9999999);
```

```
std::tie(values, samples) = interpolator.compute(lon, lat, nx=6, ny=6,
num_thread=2);
```

## 2.12. Grid's accuracy

### 2.12.1. Reader

Include file: geco/io/discrete.hpp

Class name: DiscreteXY<int32_t>

Role: To read the netCDF file containing the grid information. Note that the data will be stored as long integer. This class load the whole map by default, but it includes also the advance reading option described in the chapter above (cf. 2.1).

Constructor interface:

| Parameter | type | Description | Default value (if optional) |
|---|---|---|---|
| **Path** | std::string | Path to the netCDF grid to open. | |
| **Name** | std::string | netCDF variable that represents the discrete grid. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

### 2.12.2. Algorithm

Include file: geco/algorithm/surface/grid_error.hpp

#### 2.12.2.1. Class name: GridError

Role: To compute the error associated to a grid at given point by bilinear interpolation. If a point or more is set to NaN, the interpolation is still done with the other valid points (for example if there is only one valid point, the returned value is the value of this point, thus the algorithm works as a nearest algorithm). The number of points used is also returned. Note that this method is automatically multi-threaded (can be deactivated).

Constructor interface:

| Parameter | Type | Description | Default value (if optional) |
|---|---|---|---|
| **Mesh** | geometry::grid::DiscreteXY< int32_t > | Mesh representing the error associated to a grid on the earth's surface. | |
| **Fill_value** | double | optional: The value to use if the request point is outside | NaN |

| | | | |
|---|---|---|---|
| | | of the interpolation domain. | |

### 2.12.2.2. Process method: compute

Role: Interpolate over a 2-D grid at the request point in three steps:

1. Select the cell of the four surrounding values of the input point (x,y), removing the non-valid points (i.e. point at NaN). If all the points are invalids, then NaN is returned.

2. Determine the weight of each point of the cell according to the following formula:

Weight(x,y)=(lx−dx)/lx∗(ly−dy)/dy

Where:

lx=abs(cell(0,0).x()−cell(1,0).x())

dx=abs(cell(0,0).x()−point.x())

ly=abs(cell(0,0).y()−cell(0,1).y())

dy=abs(cell(0,0).y()−point.y())

5. Compute the interpolate value according to the following formula:

value=sum(Weight(x,y)∗cell(x,y).value())

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **lon** | Eigen::VectorXd > | Longitude in degrees. | |
| **Lat** | Eigen::VectorXd > | Latitude in degrees. | |
| **Num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std ::tuple< Eigen::VectorXd, Eigen ::VectorXi>

| Type | Description |
|---|---|
| **Eigen::VectorXd** | interpolated values at input coordinates |
| **Eigen::VectorXi** | number of points used when calculating the interpolated value |

## 2.12.3. Implementation example

```
#include "geco/algorithm/surface/mean_sea_surface.hpp"
#include "geco/algorithm/surface/grid_error.hpp"
#include "geco/io/grid/discrete.hpp"
```

```
namespace grid = geco::io::grid;
namespace surface = geco::algorithm::surface;

auto data       = grid::DiscreteXY<int32_t>(MEANSEASURFACE, "mss");
auto data_error = grid::DiscreteXY<int32_t>(MEANSEASURFACE, "error");

Eigen::VectorXd values;
Eigen::VectorXi 41imples;
Eigen::VectorXd values_error;
Eigen::VectorXi 41imples_error;
Eigen::VectorXd lon(1);
Eigen::VectorXd lat(1);

lon << 130.;
lat << 0.;

auto interpolator = surface::MeanSeaSurface(data);
std ::tie(values, samples) = interpolator.compute(lon, lat) ;

auto inter_error = surface::GridError(data_error);
std::tie(values_error, samples_error) = interp_error.compute(lon, lat);

// example changing the optional parameters
auto inter_error = surface::GridError (data_error, fill_value=9999999);
std::tie(values_error, samples_error) = inter_error.compute(lon, lat,
num_threads=2);
```

## 2.13. Ice concentration

### 2.13.1. Readers

Include file: geco/io/grid/ocean_and_sea_ice_saf.hpp

Class name: OceanAndSeaIceSAFList

Role: Reading a time series of OSI SAF grids.

Note: The list of grids does not have to be ordered chronologically (the constructor sorts the grids in chronological order).

Warning: It is the user's responsibility to ensure, if necessary, that the time difference between two grids constituting the time series is constant.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **dataset** | std::list< std::tuple< std::string, std::string >> | The list of the two hemispheres to be read. | |
| **Varname** | std::string | The netCDF variable to read | |
| **box** | geometry::Box | Geographic area to be loaded into memory. However, the entire file must be read in | geometry::Box()<br>Which means that all the content is stored |

| | | | |
|---|---|---|---|
| | | order to select this spatial area. | |

## 2.13.2. Algorithm

Include file: geco/algorithm/surface/ice_map.hpp

### 2.13.2.1. Class name: IceMap

Role: Provides methods to compute ice concentration in percentage from OSI SAF time series of sea ice concentration fields and to estimate associated flag.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **meshes** | OceanAndSeaIceSAFList | The ice map time series to be taken into account for interpolation. | |
| **Bound_error** | Bool | If true, when interpolated values are requested outside of the domain of the input mesh, an exception is raised. If false, then fill_value is used. | false |
| **neighbors** | size_t | number of nearest neighbors to be used for calculating the interpolated value. | 4 |
| **fill_value** | double | The value to use if the request point is outside the time and space interpolation range (implies bound_error false). | NaN |

### 2.13.2.2. Process method: compute

Role: The ice concentration is obtained by bilinear interpolation in space from the four nearest model grid values of the grid corresponding to the day of the measurement (no interpolation in time is done, the nearest grid in time is taken). The bilinear interpolation is done only with the valid nearest values (NaN aren't considered), if all the nearest values are NaN, then NaN is returned.

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | date of estimate (UTC) | |

| | | | |
|---|---|---|---|
| **lon** | Eigen::VectorXd | Longitude in degrees for the position at which tide is computed | |
| **lat** | Eigen::VectorXd | Latitude in degrees (positive north) for the position at which tide is computed | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| Type | description |
|---|---|
| **Eigen::VectorXd** | the interpolated values at input coordinates (in percentage). |
| **Eigen::VectorXi** | the minimum number of points used to perform spatial interpolation on one of the grids. |

### 2.13.2.3. Process static method: ice_flag_duacs

Role: To estimate the ice flag from the ice concentration according to the DUACS method:

If ice_concentration > 0 and abs(latitude) > 45 then flag = 1

Else flag = 0

Where 1 means ice and 0 no-ice.

Note: The ice concentration input of this function can be the output of the compute method of the IceMap class (Cf. compute method above).

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **ice_conc** | Eigen::VectorXd | The ice concentration (in percentage) | |
| **latitude** | Eigen::VectorXd | The latitudes of the points (in degrees) | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all | 0 |

| | | | |
|---|---|---|---|
| | | CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | |

Parameters are returned as:

| Type | description |
|---|---|
| **Eigen::VectorXd** | The ice flag vector (0 means no ice, 1: ice) |

### 2.13.3. Implementation example

```
#include "geco/io/grid/ocean_and_sea_ice_saf.hpp"
#include "geco/algorithm/surface/ice_map.hpp"

namespace surface = geco::algorithm::surface;
namespace grid        = geco::io::grid;
namespace datetime    = geco::datetime;

// Read part
// ----------------------------------------
auto mesh = geco::io::grid::OceanAndSeaIceSAFList(
      {std::tuple<std::string, std::string>{argv[1],
                                            argv[2]},
       std::tuple<std::string, std::string>{argv[3],
                                            argv[4]}},
      "ice_conc"));
// ----------------------------------------


// Algo part (GECO)
// ----------------------------------------
// Initialization
Eigen::VectorXd grid_data;
Eigen::VectorXi samples;
auto date  = datetime::utc::Array(1);
auto lon   = Eigen::VectorXd(1);
auto lat   = Eigen::VectorXd(1);

lon << 13.2;
lat << -13.5;
date << datetime::UTC(2015, 7, 21, 5, 12, 32, 14);

auto interpolator = 44imples::IceMap(mesh);
std::tie(grid_data, 44imples) = interpolator.compute(date, lon, lat, 1);
// ----------------------------------------

// calling the static method ice_flag:
auto thresholds  = Eigen::VectorXd(2);
auto flag_meanings = Eigen::VectorXi(1);
thresholds << 15, 56.0;
flag_meanings << -1, 0, 2;
auto flags = IceMap.ice_flag_duacs(grid_data, lat)
```

## 2.14. Internal tide

### 2.14.1. Readers

#### 2.14.1.1. Only M2 model

Include file: geco/io/tide/ocean.hpp

Class name: InternalM2RayZaron

Role: Load specific M2 tidal waves contained in NetCDF grids into memory for internal tide of Ray and Zaron model.

For further information about waves definitions (and associated combination of Doodson's coefficients) see the geco::algorithm::tide::ocean section of the Doxygen documentation.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **path** | std::string | The path to the Ray and Zaron netCDF M2 internal wave file. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. However, the entire file must be read in order to select this spatial area. | geometry::Box()<br>Which means that all the content is stored |

#### 2.14.1.2. Four waves model (HRET)

Include file: geco/io/tide/ocean.hpp

Class name: InternalHRET

Role: Load the specific M2, K1, S2 and O1 tidal waves contained in the High-Resolution Empirical Tide (HRET) Models.

For further information about waves definitions (and associated combination of Doodson's coefficients) see the geco::algorithm::tide::ocean section of the Doxygen documentation.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **path** | std::string | The path to the HRET netCDF file. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. However, the entire file must be read in order to select this spatial area. | geometry::Box()<br>Which means that all the content is stored |

## 2.14.2. Algorithm

Include file: geco/algorithm/tide/ocean.hpp

### 2.14.2.1. Class name: Internal

Role: Internal tide calculation. Internal tides are tides created in a stratified ocean by the interaction of the external tide with sharp bathymetric gradients. They propagate at the interfaces between ocean layers. The internal tides surface signature can reach several centimeters in some locations. The other major source of internal waves is the wind which produces internal waves near the inertial frequency. When a small water parcel is displaced from its equilibrium position, it will return either downwards due to gravity or upwards due to buoyancy. The water parcel will overshoot its original equilibrium position and this disturbance will set off an internal gravity wave.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **wave_models** | WaveModels | List of tidal wave models used by this digital model. | |
| **Fill_value** | double | The value to use if the request point is outside the time and space interpolation range (implies bound_error false). | NaN |

Note: The InternalM2RayZaron class inherit from the WaveModels class, so it can be used as wave_models parameter.

### 2.14.2.2. Process method: compute

Role: Internal tide calculation.

The height h of the ocean internal tide due to a set of n tidal constituents is:

$$h = \sum_{i=0}^{N} \sqrt{|z_i^2|}.cos(arg(z_i) - \varphi_t)$$

With:

- $z_i$: The complex value of the tide I at measurement location determined by bilinear interpolation.
- $\varphi_t = a.\tau + b.s + c.h + d.p + e.p_l + f.2\pi$
- $a, b, c, d, e, f$ : the tide's parameters (fixed for each tide) to associate with the Doodson's coefficients:
  - $\tau$: the 'Mean Lunar Time', the Greenwich Hour Angle of the mean Moon plus 12 hours.
  - $s$: the mean longitude of the Moon.
  - $h$: the mean longitude of the Sun.
  - $p$: the longitude of the Moo''s mean perigee.

- $p_l$: the longitude of the Su"s mean perigee.

There is no tide computed by admittance in internal tide computation.

These computations are no longer realized by geco since v2.1.0, but by the pyfes library (new dependency since v2.1.0, delivered with geco).

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | date of estimate (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees for the position at which tide is computed | |
| **lat** | Eigen::VectorXd | Latitude in degrees (positive north) for the position at which tide is computed | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| Type | description |
|---|---|
| **Eigen::VectorXd** | Computed height of the internal tidal spectrum (m). |
| **Eigen::VectorXi** | The minimum number of valid data used in the bi-linear interpolation of tidal waves from their numerical models. |

## 2.14.3. Implementation example

```
#include""geco/io/tide/ocean.hp""
#include""geco/algorithm/tide/ocean.hp""

namespace io       = geco::io::tide::ocean;
namespace datetime = geco::datetime;
namespace ocean    = geco::algorithm::tide::ocean;

// Read part
```

```
// ----------------------------------------
auto internal_grid = io::InternalM2RayZaron(M2_INTERNAL);
auto internal_hret_grid = io::InternalHRET(HRET_INTERNAL);
// ----------------------------------------


// Algo part (GECO)
// ----------------------------------------
// Initialization
Eigen::VectorXd grid_data;
Eigen::VectorXi samples;
auto date  = datetime::utc::Array(1);
auto lon  = Eigen::VectorXd(1);
auto lat  = Eigen::VectorXd(1);

lon << 13.2;
lat << -13.5;
date << datetime::UTC(2015, 7, 21, 5, 12, 32, 14);

auto interpolator = ocean::Internal<float>((internal_grid);
std::tie(intide,48impless) = interpolator.compute(date, lon, lat, 1);
// ----------------------------------------
auto interpolator_hret = ocean::Internal<float>(( internal_hret_grid);
std::tie(intide,48impless) = interpolator_hret.compute(date, lon, lat, 1);
// ----------------------------------------
```

## 2.15. Inverted barometer

Inverted barometer correction requires sea surface pressure and surface type auxiliary data.

### 2.15.1. Readers

#### 2.15.1.1. sea surface pressure

Cf. 2.4.1.1

#### 2.15.1.2. surface type

Cf. 2.25.1

#### 2.15.1.3. s1s2 ECMWF climatology

Cf. 2.7.1.2

### 2.15.2. Algorithm

Include file: geco/algorithm/environment/meteo.hpp

#### 2.15.2.1. Class name: InvertedBarometer

Role: Extensive modeling work by Ponte et al. (1991) confirms that over most open ocean regions the ocean response to atmospheric pressure forcing is mostly static. Typical deviations from the inverted

barometer response are in the range of 1 to 3 cm rms, with most of the variance occurring at high frequencies. The Inverted barometer correction is not reliable for pressure variations with very short periods (< 2 days) and in coastal regions. This inverted barometer height calculation uses a non-constant mean reference sea surface pressure. As stated by Dorandeu and Le Traon (1999), this improved inverted barometer height correction reduces the standard deviation of mean sea level variations (relative to an annual cycle and slope) by more than 20% when compared with the standard inverted barometer height correction (i.e. with constant reference pressure) and no correction at all. It also slightly reduces the variance of sea surface height differences at altimeter crossover points, and the impact of the improved correction on the mean sea level annual cycle and slope is also significant.

Constructor interface:

| Parameter | type | Description | Default value (if optional) |
|---|---|---|---|
| **meshes_pressure** | interpolation::ReducedGaussianXYList | The pressure time series to be taken into account for interpolation (**sea surface pressure is recommended**) | |
| **s1s2_climatology** | interpolation::SampledClimatologyDataSet< float > | S1/S2 climatology based on ECMWF pressure fields generated every 6 hours. | |
| **meshes_surface_type (*)** | geometry::grid::DiscreteXY< int8_t > | The land/sea mask to be taken into account to compute the mean sea level pressure | |
| **bound_error** | bool | If true, when interpolated values are requested outside of the domain of the input mesh, an exception is raised. If false, then fill_value is used. | false |
| **neighbors** | size_t | The number of nearest neighbors to be used for calculating the interpolated value. | 4 |
| **fill_value** | double | The value to use if the request point is outside of the interpolation domain. | NaN |

Note : (*) The surface pressure auxiliary file must be a 7-state surface type mask such as : 0 means open ocean and 5 means floating ice.

### 2.15.2.2. Process method: compute

Role: To compute the inverted barometer correction from sea surface pressure such as:

$$Inverted\ barometer = -b \times (P - \bar{P})$$

With:

- b: The coefficient to apply to the pressure to obtain the IB (default: 9.948e-5 m/Pa [Wunsch, 1972])
- P: the sea surface pressure interpolated at input position. The result is obtained by:
  - linear interpolation in time between the two surrounding ECMWF files in time from the meshes_pressure input parameter
  - and by bilinear interpolation in space from the four nearest model grid values.
  - Then the ECMWF climatological pressure computed at input measurement's time and location is removed if the s1s2_cor option is set to true.
- $\bar{P}$: The mean sea surface pressure at input time-tag computed such as:
  - the nearest ECMWF grid in time from meshes_pressure parameter is selected
  - then, with the help of the meshes_surface_type grid, the mean pressure of the Open Ocean (type = 0) and floating ice (type = 5) points of the selected pressure grid is computed.

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | time coordinates (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees | |
| **lat** | Eigen::VectorXd | Latitude in degrees | |
| **s1s2_cor** | bool | If true, the surface pressure will be corrected from s1s2.<br>If false, surface pressure will not be corrected and kept as provided. | True |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |
| **b_coef** | double | The coefficient to apply to the pressure to obtain the IB (default: 9.948e-5 m/Pa [Wunsch, 1972]) | 9.948e-5 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| Type | description |
|---|---|
| **Eige n::VectorXd** | the estimated inverted barometer (in meters) at input coordinates and |
| **Eigen::VectorXi** | the number of points used when calculating the interpolated value |

## 2.15.3. Implementation example

```
#include""geco/algorithm/environment/meteorology.hp""
#include""geco/io/grib/reduced_gaussian_xy_list.hpp"
#include""geco/io/grid/discrete.hp""
```

```
int main(int argc, char** argv) {
  if (argc != 4) {
    std::cerr <<""Usage:"" << argv[0] <<"" <former> <later> <ecmwf_s1s2_dir>
<surface_type"" << std::endl;
    return 1;
  }
  // Defining readers
  auto pressure = geco::io::grib::ReducedGaussianXYList({argv[1], argv[2]});
  auto climatology = grid::SampledClimatologyDataSet<float>(argv[3],
"surface_pressure");  auto surface =
geco::io::grid::DiscreteXY<int8_t>(argv[4],""mas"");

  // Inputs
  auto start = geco::datetime::UTC(2015, 7, 21);
  auto step = geco::datetime::TimeDelta(0, 900, 0);
  auto end = geco::datetime::UTC(2015, 7, 21, 6);
  size_t size = (end-- start) / step;
  size_t ix = 0;
  Eigen::VectorXd lon(size);
  Eigen::VectorXd lat(size);
  auto dates = geco::datetime::utc::Array(size);
  for (auto item = start; item < end; item += step) {
    lon[ix] = lat[ix] = 0;
    dates[ix++] = item;
  }

  // Algorithm
  auto interpolator =
      geco::algorithm::environment::InvertedBarometer(pressure, climatology,
surface, true, 4);
  Eigen::VectorXd values;
  Eigen::VectorXi samples;
  std::tie(values, samples) = interpolator.compute(dates, lon, lat, true, 2,
9.948e-05);
  std::cout << values << std::endl;
  return 0;
}
```

## 2.16. Ionospheric delay along track

### 2.16.1. Reader

Include file: io/binary/gim.hpp

Class name: GIM

Role: Reads the binary data generated by the GIM component and containing the value of the ionospheric correction as a function of time for a given half-orbit.

This file is a binary file containing a MPH header of 1625 bytes followed by a vector containing 86400 measurements stored on signed 4-byte integers occupying 345600 bytes.

Loads one or two files containing one or two successive days into memory.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **before** | std::string | the first GIM file to load | |
| **after** | boost::optional< std::string > | the next GIM file to load | boost::optional< std::string >()<br>Which means none. |

## 2.16.2. Algorithm

Include file: algorithm/environment/gim.hpp

### 2.16.2.1. Class name: GIM

Role: Calculates the global ionospheric correction. This class has two types of compute method, for one or two frequencies:

- compute_mono: The purpose of this function is to compute the ionospheric correction. The returned value is the nearest GIM value at input date.
- compute_dual: The purpose of this function is to compute the ionospheric corrections associated to 2 frequencies: the first associated to the GIM file (main) the second associated to an auxiliary instrument. The formula used to estimate this second value is: AuxCorrection = (MainFrequency/AuxFrequency)^2.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **var** | math::variable::TimeDepend< int > | Variable containing the value of the GIM correction as a function of time. | |

### 2.16.2.2. Process method: compute_mono

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Date** | Eigen::Ref< DateTimeVector > | The date on which the ionospheric correction of the satellite will be estimated. | |

Parameter is returned as:

A single Eigen vector.

| type | description |
|---|---|
| **Eigen::VectorXd** | the ionospheric correction calculated |

### 2.16.2.3. Process method: compute_dual

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **date** | DateTimeVector | The date on which the ionospheric correction of the satellite will be estimated. | |
| **main_frequency** | double | Band radar frequency associated to the ionospheric GIM file data read. | |
| **aux_frequency** | double | Auxiliary band radar frequency for which the ionospheric correction is also wanted. | |

Parameter is returned as:

A tuple containing the two ionospheric corrections calculated for the main and the aux frequencies.

std::tuple<Eigen::VectorXd, Eigen::VectorXd>

| type | description |
|---|---|
| **Eigen::VectorXd** | The ionospheric correction calculated for the main frequency. |
| **Eigen::VectorXd** | The ionospheric correction calculated for the auxiliary frequency. |

## 2.16.3. Implementation example

```
#include""geco/algorithm/environment/gim.hp""
#include""geco/io/binary/gim.hp""

namespace binary      = geco::io::binary;
namespace environment = gec o::algorith m::environmen t;
namespace math        = geco::math;
namespace datetime    = geco::datetime;

auto data = binary::GIM(GIM1, boost::optional<std::string>(GIM2));

# defining dates
auto range = math::TemporalAxis<datetime::UTC>(
    datetime::UTC(2018, 2, 19, 12), datetime::UTC(2018, 2, 19, 12, 0, 1),
    datetime::TimeDelta(0, 0, 100000));
auto dates = Eigen::Matrix<datetime::UTC, Eigen::Dynamic, 1>(range.size());
for (size_t ix = 0; ix < dates.size(); ++ix) {
  dates[ix] = range.coordinate_value(ix);
}
```

```
double freq_main = 10.0e 9;
double freq_aux  = 20.0e 9;
auto result  = Eigen::VectorXd(range.size());
auto result1 = Eigen::VectorXd(range.size());
auto result2 = Eigen::VectorXd(range.size());

auto gim = environment::GIM(data);
result = gim.compute_mono(dates);
std::tie(result1, result2) = gim.compute_dual(dates, freq_main, freq_aux);
```

## 2.17. Ionospheric delay map

### 2.17.1. Reader

Include file: io/ascii/gim.hpp

Class name: TotalElectronContents

Role: Load Global ionospheric maps (GIM) into memory. The constructor will load all the provided IONEX files

The following checks are performed to test the integrity of the data:

- All grids cover the same area :
  - Longitude: -180 to +180 with intervals of 5 degrees (73 grid lines)
  - Latitude: -87.5 to +87.5 with intervals of 2.5 degrees (71 grid lines)
  - TEC: In units of 0.1 TEC Units, or 10e15 electrons.m−2
- The spacing between the grids is two hours.

The type of IONEX file to processed is a parameter such as:

| Kind | |
|---|---|
| **kJPLG** | JPL analysis files. |
| **kJPLQ** | JPL quick-look files. |
| **kCODG** | CODE analysis files. |

First constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **TEC files** | std::vector<std::string> | The files to be processed | |
| **kind** | Kind | Type of IONEX file processed. | |

Important note: this constructor is sensitive to the file name. If the pattern of a file provided in input does not contain the value of the “kind” parameter provided in input, the file is excluded.

The constructor described below is not sensitive to the name. When using it, it is the responsibility to the caller to ensure the integrity and the coherence of the files provided.

Second constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **TEC files** | std::vector<std::string> | The files to be processed | |

## 2.17.2. Algorithm

Include file: algorithm/environment/gim.hpp

### 2.17.2.1. Class name: GlobalIonosphericMap

Role: Interpolation of the total electron content (TEC) from the TEC grids observed by GPS/GLONASS satellites.

Space-time interpolation of the TEC from the GIM maps loaded into memory according to the following steps:

- The two surrounding grids in time are selected.
- The time shift between input time-tag and the grids times is computed to rotate them and compensate the sun rotation.
- Bilinear interpolation on space is done for the two grids.
- Then linear interpolation in time is done.

The extrapolation beyond the last grid loaded is authorized up to delta_t (parameter).

Note The last date (00h00) of each file is not considered, since the first date of the next file is the same (and has different content). The consequence is that to interpolate between 23h00 and 00H00 of a D day, you must also provide the D+1 day file.

For detailed pseudo-code see ANNEXE 2 – Pseudo-code of TEC grids interpolation.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **tec** | TotalElectronContents | GIM grids loaded into memory | |
| **delta_t** | datetime::TimeDelta | The time interval during which extrapolation is allowed after the last date defined by the time series loaded into memory. By default, the extrapolation is not allowed. | 0 |
| **fill_value** | double | The value to use if the request point is outside TEC maps. | NaN |

### 2.17.2.2. Process method: compute

Role: Space-time interpolation of the TEC from the GIM maps loaded into memory.

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | time coordinates (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees | |
| **lat** | Eigen::VectorXd | Latitude in degrees | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

| type | description |
|---|---|
| **Eigen::VectorXd** | the TEC interpolated for the requested positions |

## 2.17.3. Implementation example

```
#include "geco/algorithm/environment/gim.hpp"
#include "geco/io/ascii/gim.hpp"


int main(int argc, char** argv) {
  if (argc != 4) {
    std::cerr <<« "Usage:« " << argv[0] <<« " <tec_1> <tec_2 »" << st d::end l;
    return 1;
  }

  // to load the IONEX files
  std::vector<string> tec_files = {argv[1], argv[2]}
  auto data = geco::io::ascii::TotalElectronContents(
      argv[1], start, end,
      geco::algorithm::environment::TotalElectronContents::Kind::kJPLQ);
  // Or !
  auto data = geco::io::ascii::TotalElectronContents(argv[1]);

  // inputs
  geco::datetime::utc::Array date(3);
  Eigen::VectorXd lon(3), lat(3);
  date << geco::datetime::UTC(2018, 12, 6, 13, 14, 4, 520158),
      geco::datetime::UTC(2018, 12, 6, 13, 14, 4, 520158),
      geco::datetime::UTC(2018, 12, 6, 13, 14, 4, 520158);
  lon << -120.3, 0.5, 12.7745215;
  lat << -56.3, 0.5, 89.7745215;

  // algorithm
  auto interpolator = geco::algorithm::environment::GlobalIonosphericMap(data);
  auto result = interpolator.compute(date, lon, lat);
```

```
  return 0;
}
```

## 2.18. Mean Dynamic Topography

The following algorithms have same usage for all the MDT model: CNES/CLS 2013, CNES/CLS2018 and DTU15.

### 2.18.1. Reader

Include file: geco/io/discrete.hpp

Class name: DiscreteXY<int32_t>

Role: To read the netCDF file containing the grid information. Note that the data will be stored as long integer. This class load the whole map by default, but it includes also the advance reading option described in the chapter above (cf. 2.1).

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **Path** | std::string | Path to the netCDF grid to open. | |
| **Name** | std::string | netCDF variable that represents the discrete grid. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

### 2.18.2. Algorithm

Include file: geco/algorithm/surface/mean_dynamic_topography.hpp

#### 2.18.2.1. Class name: MeanDynamicTopography

Role: To compute the mean dynamic topography height at given point by bilinear interpolation. If a point or more is missing, the interpolation is still done with the other valid points (for example if there is only one valid point, the returned value is the value of this point, thus the algorithm works as a nearest algorithm). The number of points used is also returned by the compute method. Note that this method is automatically multi-threaded (can be deactivated).

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **mesh** | geometry::grid::DiscreteXY<int32_t> | Mesh representing the MDT. | |
| **Fill_value** | double | The value to use if the request point is outside of the interpolation domain (implies bound_error false). | NaN |

### 2.18.2.2. Process method: compute

Role: Evaluate the 2D spline at given points in three steps.

1. For each lat/lon point of the inputs lat/lon vectors: Select the frame (of size nx,ny) of values surrounding the input point.

2. For each latitude of the frame (i.e. ny times) select the corresponding longitude vector of values and interpolate the value by spline at longitude to obtain a vector of size (ny).

3. Then, interpolate by spline at latitude this vector to obtain a single 2d interpolated by spline value.

Note: the spline is performed by the gsl_interp_eval function of the gsl library.

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **lon** | Eigen::VectorXd | longitudes in degrees | |
| **lat** | Eigen::VectorXd | latitudes in degrees | |
| **num_thread** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| type | description |
|---|---|
| **Eigen::VectorXd** | interpolated values at input coordinates |
| **Eigen::VectorXi** | number of points used when calculating the interpolated value |

## 2.18.3. Implementation example

```
#include""geco/algorithm/surface/mean_dynamic_topography.hp""
#include""geco/io/grid/discrete.hp""
```

```
namespace grid = geco::io::grid;
namespace surface = geco::algorithm::surface;

auto data = grid::DiscreteXY<int32_t>(MEANDYNTOPO,"”md””);

Eigen::VectorXd values;
Eigen::VectorXi samples;
Eigen::VectorXd lon(1);
Eigen::VectorXd lat(1);

lon << 130.;
lat << 0.;

auto interpolator = surface::MeanDynamicTopography(data);
std::tie(values, samples) = interpolator.compute(lon, lat);

// example changing the optional parameters
auto interpolator = surface::MeanDynamicTopography(data, fill_value=9999999);
std::tie(values, samples) = interpolator.compute(lon, lat, nx=6, ny=6,
num_thread=2);
```

## 2.19. Mean Sea Surface

The following algorithms have same usage for all the MSS model: CNES/CLS 2015, DTU15 and DTU18.

### 2.19.1. Reader

Include file: geco/io/discrete.hpp

Class name: DiscreteXY<int32_t>

Role: To read the netCDF file containing the grid information. This class load the whole map by default, but includes also the advance reading option described in the chapter above (cf. 2.1).

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Path** | std::string | Path to the netCDF grid to open. | |
| **Name** | std::string | netCDF variable that represents the discrete grid. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

### 2.19.2. Algorithm

Include file: geco/algorithm/surface/mean_sea_surface.hpp

### 2.19.2.1. Class name: MeanSeaSurface

Role: To compute the mean sea surface height. The height of the MSS is computed at the altimeter measurement using a squared window of NxN MSS grid points (typically N = 6) centered on the altimeter point. Spline functions are calculated within the window as function of grid point latitude for each MSS column. Each of these spline functions is evaluated at the altimeter latitude. The resulting values are then used for calculating a spline function of grid point longitude. The height of the MSS is derived by evaluating the spline at the altimeter longitude. The number of points used is also returned. Note that this method is automatically multi-threaded (can be deactivated).

Note in the Bibli_Alti legacy software we use a surface mask to select only ocean grid point over ocean surfaces This is not required any more as the MSS recent solutions includes meaningful and continuous geoid information over land surfaces.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **mesh** | geometry::grid::DiscreteXY<int32_t> | Mesh representing the MSS. | |
| **fill_value** | double | The value to use if the request point is outside of the interpolation domain. | NaN |

### 2.19.2.2. Process method: compute

Role: Evaluate the 2D spline at given points in three steps.

1. For each lat/lon point of the inputs lat/lon vectors: Select the frame (of size nx,ny) of values surrounding the input point.

2. For each latitude of the frame (i.e. ny times) select the corresponding longitude vector of values and interpolate the value by spline at input longitude value in order to obtain a vector of size (ny).

3. Then, interpolate by spline at input latitude value this vector.

Note: the spline is performed by the gsl_interp_eval function of the gsl library.

Process method interface:

| Parameter | Type | Description | Default value (if optional) |
|---|---|---|---|
| **lon** | Eigen::VectorXd | longitudes in degrees | |
| **lat** | Eigen::VectorXd | latitudes in degrees | |
| **nx** | size_t | number of longitude values required to perform the spline interpolation. | 3 |
| **ny** | size_t | number of latitude values required to perform the spline interpolation | 3 |

| | | | |
|---|---|---|---|
| **num_thread** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| Type | Description |
|---|---|
| **Eigen::VectorXd** | interpolated values at input coordinates |
| **Eigen::VectorXi** | number of points used when calculating the interpolated value |

### 2.19.3. Implementation example

```
#include""geco/algorithm/surface/mean_sea_surface.hp""
#include""geco/io/grid/discrete.hp""

namespace grid = geco::io::grid;
namespace surface = geco::algorithm::surface;

auto data = grid::DiscreteXY<int32_t>(MEANSEASURFACE,""ms"");

Eigen::VectorXd values;
Eigen::VectorXi samples;
Eigen::VectorXd lon(6);
Eigen::VectorXd lat(6);

lat << 66.618891, 47.376443, 51.275358, -1.115888, 39.895090, 61.356805;
lon << -46.136710, -36.457034, -74.962535, 32.739449, 51.282597, -114.756202;

auto interpolator = surface::MeanSeaSurface(data);
std::tie(values, samples) = interpolator.compute(lon, lat);

// example changing the optional parameters
auto interpolator = surface::MeanSeaSurface(data, fill_value=9999999);
std::tie(values, samples) = interpolator.compute(lon, lat, nx=6, ny=6,
num_thread=2);
```

## 2.20. MF WAM: mean wave period and direction

Each MF WAM file contains two fields:

- The Mean Wave Period based on second moment in seconds.

- The Mean Wave Direction in degrees.

There are two classes of reader, one per field, and only one class of algorithm (which is the same as the meteo correction's algorithm) for this theme.

## 2.20.1. Readers

### 2.20.1.1. To load Mean Wave Period

Include file: geco/io/grib/mfwam.hpp

Class name: MfWamMp2

Role: Loading into memory a series of grids stored in GRIB format that contain MF WAM Mean Wave Period (mp2) data.

Note: The list of grids does not have to be ordered chronologically (the constructor sorts the grids in chronological order).

Warning: It is the use''s responsibility to ensure, if necessary, that the time difference between two grids constituting the time series is constant.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Meshes** | std::list< std::string > | The list of grids to store in memory. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

### 2.20.1.2. To load Mean Wave Direction

Include file: geco/io/grib/mfwam.hpp

Class name: Mesh_MfWam_mwd

Role: Loading into memory a series of grids stored in GRIB format that contain MF WAM Mean Wave Direction (mwd) data.

Note: The list of grids does not have to be ordered chronologically (the constructor sorts the grids in chronological order).

Warning: It is the use''s responsibility to ensure, if necessary, that the time difference between two grids constituting the time series is constant.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Meshes** | std::list< std::string > | The list of grids to store in memory. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

## 2.20.2. Algorithm

Include file: geco/algorithm/interpolation/reduced_gaussian_xy_series.hpp

### 2.20.2.1. Class name: ReducedGaussianXYSeries

Role: To interpolate a time series of meteorological fields represented by a reduced Gaussian grid stored in a GRIB file. The result is obtained by linear interpolation in time between two consecutive data files, and by bilinear interpolation in space from the four nearest model grid values.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **meshes** | ReducedGaussianXYList | The time series to be taken into account for interpolation. | |
| **bound_error** | Bool | If true, when interpolated values are requested outside of the domain of the input mesh, an exception is raised. If false, then fill_value is used. | false |
| **neighbors** | size_t | number of nearest neighbors to be used for calculating the interpolated value. | 4 |
| **fill_value** | double | The value to use if the request point is outside the time and space interpolation range (implies bound_error false). | NaN |

### 2.20.2.2. Process method: compute

Role:

Spatial (and temporal if E=kTimeLinear) interpolation of the time series handled by this instance, for the requested spatial and time coordinates. The interpolation follows the specification below:

- The two grids surrounding the time-tag value are selected, then for each one:
  - the N nearest neighbors in space of the requested position are selected
  - The NaN values are not considered, if they all are NaN, then NaN value is returned for the current grid
  - The weight is computed for each valid value using the inverse distance weighting method
  - If the data are not in degrees, the bilinear mean is computed and returned, else another method is followed (to avoid encountered issues when values are around 0 and 360 degrees):
  - the mean of the valid neighbors cosine is computed

- the mean of the valid neighbors sinus is computed
- the arc-tangent of the previous mean cosine and mean sinus is computed
- this value is then returned [modulo 360 degrees]

- If temporal interpolation is required (i.e. if E=kTimeLinear) the linear interpolation between the two values is performed and returned (if one of them is NaN, then NaN is returned).
- Else, (i.e. E=kTimeNearest) the value of the nearest grid in time is returned.

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | date of estimate (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees for the position at which tide is computed | |
| **lat** | Eigen::VectorXd | Latitude in degrees (positive north) for the position at which tide is computed | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| Type | description |
|---|---|
| **Eigen::VectorXd** | the interpolated values at input coordinates (the unit depends on data read.) |
| **Eigen::VectorXi** | the minimum number of points used to perform spatial interpolation on one of the grids. |

## 2.20.3. Implementation example

```
#include""geco/io/grib/mfwam.hp""
#include""geco/algorithm/interpolation/reduced_gaussian_xy_series.hp""

namespace interpolation = geco::algorithm::interpolation;
namespace grib         = geco::io::grib;
namespace datetime     = geco::datetime;
```

```
// Read part
// ----------------------------------------
auto gribs = std::list<std::string>();
gribs.push_back(MFWAM_1);
gribs.push_back(MFWAM_2);
auto mwd_fields = grib::Mesh_MfWam_mwd(gribs);
auto mp2_fields = grib::Mesh_MfWam_mp2(gribs);
// ----------------------------------------


// Algo part (GECO)
// ----------------------------------------
// Initialization
Eigen::VectorXd grib_mp2, grib_mwd;
Eigen::VectorXi samples;
auto date  = datetime::utc::Array(1);
auto lon  = Eigen::VectorXd(1);
auto lat  = Eigen::VectorXd(1);

lon << 130;
lat << 90;
date << datetime::UTC(2019, 2, 11, 5, 0, 0, 0);

auto interpolator = interpolation::ReducedGaussianXYSeries(mp2_fields);
std::tie(grib_mp2, samples) = interpolator.compute(date, lon, lat, 1);
interpolator = interpolation::ReducedGaussianXYSeries(mwd_fields);
std::tie(grib_mwd, samples) = interpolator.compute(date, lon, lat, 1);
// ----------------------------------------
```

## 2.21. Ocean Tidal Prediction FES and GOT

The ocean tidal prediction can be calculated from two kinds of data: FES or GOT. As these data contains a lot of files, they both are defined in a simple configuration file which can be find in the **docs/configuration** folder. So, the reader doesn't mention the list of all the FES or GOT data files, but only this single configuration file, see the following section for further information about configuration file.

### 2.21.1. Readers

#### 2.21.1.1. Configuration file

Include file: geco/io/tide/ocean.hpp

Class name: Configuration

Role:

Builds a new object from the information stored in the JSON configuration file.

The structure of the file is as follows:

```
1 {
2    ""radia"": {
3        ""Q"": {
4            ""amplitud"":""amplitud"",
5            ""pat"":""/path/to/Q1_radial.n"",
6            ""phas"":""phas""
7         },
8        ""O"": {
```

```
 9            ""amplitud"":""amplitud"",
10            ""pat"":""/path/to/O1_radial.n"",
11            ""phas"":""phas""
12          }
13      },
14     ""tid"": {
15         ""Q"": {
16            ""amplitud"":""amplitud"",
17            ""pat"":""/path/to/Q1.n"",
18            ""phas"":""phas""
19         },
20         ""O"": {
21            ""amplitud"":""amplitud"",
22            ""pat"":""/path/to/O1.n"",
23            ""phas"":""phas""
24         }
25      },
26     ""long_perio"": [
27         ""M"",
28         « "M »",
29         « "Msq »",
30         « "Mt »",
31         « "S »",
32         « "Ss »"
33      ]
34 }
```

This structure defines:

-""tide"" the map of waves to be taken into account in the calculation of the ocean tide,

-""radial"" the map of waves to be taken into account in the calculation of the radial tide,

-""long_period"" the list of long period waves that should not be taken into account in order to calculate the equilibrium tide.

Each wave identifier in the map of waves defines an object describing the properties of the file to be read:

-""path"" the path to the file to be read and

-""amplitude"" the name of the NetCDF variables describing the amplitude the wave

-""phase"" the name of the NetCDF variables describing the phase lag of the wave

Note

The string describing the path to the NetCDF file to read can contain an environment variable specified using the following syntax: ${VARNAME}. This variable will be substituted by its value if it is defined, or by an empty string if it is not defined.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **filename** | std::string | Name of file from which to read in the configuration. | |

### 2.21.1.2. Reader

Include file: geco/io/tide/ocean.hpp

Class name: ModelData<float>

Role: Loads the different waves needed to calculate the:

ocean tide height,

tidal loading height

total long period ocean tide height

equilibrium long period ocean tide height

This class takes a path to a configuration file as input parameter. It is this configuration file that defines what kind of data is used to compute the ocean tide heights (FES or GOT).

For further information about waves definitions (and associated combination of Doodson's coefficients) see the geco::algorithm::tide::ocean section of the Doxygen documentation.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **configuration** | io::tide::ocean::Configuration | The instance describing the different waves to be used to calculate the different ocean tides. | |
| **box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

## 2.21.2. Algorithm

Include file: geco/algorithm/tide/ocean.hpp

### 2.21.2.1. Class name: ModelHandler

Role: Performing the ocean tide calculation for a given model using the FES library (converted from C to C++) distributed by AVISO, with support from CNES.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **model_data** | ModelData< T > | Container for the various data needed to calculate the ocean tide. | |

| | | | |
|---|---|---|---|
| **fill_value** | Double | The value to be used if the tide is not defined for the requested point. | NaN |

### 2.21.2.2. Process method: compute

Role: This method makes it easy to calculate the different ocean tides, namely:

- SP_Ocean: short period ocean tides (semi_diurnal and diurnal tides),
- LP_Ocean: dynamic long period ocean tides,
- SP_Load: short period loading effects,
- LP_Load: long period loading effects,
- LP_Equilibrium: equilibrium long period ocean tide,
- N: minimum number of cell points used during the interpolation process of one of the model waves.

For SWOT mission, the different solutions (with FES or GOT model) for the tides are:

- **ocean_tide_got** = SP_Ocean(**GOT**) + SP_Load(**GOT**) + LP_Equilibrium
- **ocean_tide_fes** = SP_Ocean(**FES**) + SP_Load(**FES**) + LP_Equilibrium
- **ocean_tide_non_equ** = LP_Ocean(**FES**) + LP_Load(**FES**) – LP_Equilibrium
- **load_tide_got** = SP_Load(**GOT**)
- **load_tide_fes** = SP_Load(**FES**) + LP_Load(**FES**)
- **ocean_tide_eq** = LP_Equilibrium

Special treatment over land/ocean:

*Over ocean :*

| | *Ocean Tide effects* | | | *Load Tide effects* | | |
|---|---|---|---|---|---|---|
| | *short period* | *long period eq* | *long period non eq* | *short period* | *long period eq* | *long period non eq* |
| *ocean_tide* | *Yes* | *Yes* | *No* | *Yes* | *No* | *No* |
| *load_tide* | *No* | *No* | *No* | *Yes* | *No* | *Yes* |
| *ocean_tide_eq* | *No* | *Yes* | *No* | *No* | *No* | *No* |
| *ocean_tide_non_eq* | *No* | *No* | *Yes* | *No* | *No* | *Yes* |

*Over lands :*

| | *Load Tide effects* | | |
|---|---|---|---|
| | *short period* | *long period eq* | *long period non eq* |
| *ocean_tide* | *Yes* | *No* | *No* |
| *load_tide* | *Yes* | *No* | *Yes* |
| *ocean_tide_eq* | *No* | *No* | *No* |

| | | | |
|---|---|---|---|
| *ocean_tide_non_eq* | *No* | *No* | *No* |

The computations are no longer realized by geco since v2.1.0, but by the pyfes library (new dependency since v2.1.0, delivered with geco).

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | date of estimate (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees for the position at which tide is computed | |
| **lat** | Eigen::VectorXd | Latitude in degrees (positive north) for the position at which tide is computed | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXd, Eigen::VectorXd, Eigen::VectorXd, Eigen::VectorXd , Eigen::VectorXi>

| type | Description |
|---|---|
| **Eigen::VectorXd** | SP_Ocean (m) |
| **Eigen::VectorXd** | LP_Ocean (m) |
| **Eigen::VectorXd** | SP_Load (m) |
| **Eigen::VectorXd** | LP_Load (m) |
| **Eigen::VectorXd** | LP_Equilibrium (m) |
| **Eigen::VectorXi** | The quality of the tide calculation. Could be Undefined (0) if the point is not defined by the model, Interpolated (4) if the model is interpolated, Extrapolated (1 with one data point, 2 with two data points, 3 with three data points) if the model is extrapolated. |

### 2.21.3. Implementation example

```
#include""geco/io/tide/ocean.hp""
#include""geco/algorithm/tide/ocean.hp""

namespace iocean   = geco::io::tide::ocean;
namespace algocean = geco::algorithm::tide::ocean;
namespace datetime = geco::datetime;

// Read part
// ----------------------------------------
auto model_data = iocean::ModelData<float>(
    iocean::Configuration(TIDE_CONFIGFILE));
// ----------------------------------------


// Algo part (GECO)
// ----------------------------------------
// Initialization
Eigen::VectorXd sp_ocean;
Eigen::VectorXd lp_ocean;
Eigen::VectorXd sp_tidal;
Eigen::VectorXd lp_tidal;
Eigen::VectorXd lp_eq;
Eigen::VectorXi70impless;
auto date  = datetime::utc::Array(1);
auto lon   = Eigen::VectorXd(1);
auto lat   = Eigen::VectorXd(1);

lon << 0.0;
lat << 0.0;
date << datetime::UTC(2015, 7, 21, 5, 12, 32, 14);

auto interpolator = algocean::ModelHandler<float>(model_data);
std::tie(sp_ocean, lp_ocean, sp_tidal, lp_tidal, lp_eq, samples) =
interpolator.compute(date, lon, lat);
```

## 2.22. Pole Tidal Prediction

The pole tidal prediction needs to load two auxiliary files to be performed, one static and one dynamic.

### 2.22.1. Readers

#### 2.22.1.1. The pole tide wave models (static)

Include file: geco/io/tide/pole.hpp

Class name: WaveModels<float>

Role: Loads the grids of the polar tide model into memory.

Constructor interface:

| Parameter | type | description | Default value |
|---|---|---|---|

| | | | (if optional) |
|---|---|---|---|
| **Path** | std::string | Path to the nc file containing the polar tide to load into memory | |
| **ocean_real** | std::string | Var name of the real admittance for ocean tide component | |
| **ocean_imag** | std::string | Var name of the imaginary admittance for ocean tide component | |
| **load_real** | std::string | Var name of the real admittance for load tide component | |
| **load_imag** | std::string | Var name of the imaginary admittance for load tide component | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

#### 2.22.1.2. The pole position (dynamic)

Include file: geco/io/ascii/pole.hpp

Class name: PolePosition

Role: Pole Location: at instantiation the file is directly open, load, read and closed.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **path** | std::string | the path to the file | |

### 2.22.2. Algorithm

Include file: geco/algorithm/tide/pole.hpp

#### 2.22.2.1. Class name: Desai

Role: This class aims to calculate the geocentric pole tide height displacement due to polar motion of the surface of the Earth for satellite altimetry, using the Desai 2017 algorithm. Only radial component of displacement is computed.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **wave_models** | WaveModels< float > | Digital model of the polar tide to use | |
| **position** | Position | Object used to obtain the location of the poles for a given date. | |
| **fill_value** | double | optional: the value to be used if the tide is not defined for the requested point. | NaN |

### 2.22.2.2. Process method: compute

Role:

The Earth's rotational axis oscillates around its nominal direction with apparent periods of 12 and 14 months. This results in an additional centrifugal force which displaces the surface. The effect is called the pole tide. It is easily computed if the location of the pole and the ocean/load pole tide model are known (Desai, 2015).

The pole location and the average pole coordinates are corrected to account for a Glacial Isostatic Adjustment (GIA) model (Desai, 2015).

The pole tide is given in meters by the sum of three components:

$$H_{PoleTide} = H_{BodyPoleTide} + H_{OceanPoleTide} + H_{RadialLoadPoleTide} \quad (1)$$

Where :

* The body (Solid Earth) pole tide $H_{BodyPoleTide}$ is given in meters by:

$$H_{BodyPoleTide} = -A.H2.\sin\phi.\cos\phi.[(x - x_{avg})\cos\lambda - (y - y_{avg})\sin\lambda] \quad (2)$$

* The ocean pole tide with respect to crust $H_{OceanPoleTide}$ is given in meters with respect to (Desaï, 2015) ocean pole tide model $H_{OceanPoleTideModel}$ by:

$$\begin{aligned} H_{OceanPoleTide} = A.\sqrt{\frac{8\pi}{15}}[(1 + K_{2,x} - H_2).H_{OceanPoleTideModel}(x) \\ - K2_y.H_{OceanPoleTideModel}(y)](x - x_{avg}) \\ - A.\sqrt{\frac{8\pi}{15}}[(1 + K_{2,x} - H_2).H_{OceanPoleTideModel}(y) \\ + K_{2,y}.H_{OceanPoleTideModel}(x)](y - y_{avg})] \end{aligned} \quad (3)$$

* The load pole tide (radial) with respect to center of mass of total Earth system $H_{RadialLoadPoleTide}$ is given in meters with respect to (Desaï, 2015) radial load pole tide model $H_{RadialLoadPoleTideModel}$ by:

$$H_{RadialLoadPoleTide} = A.\sqrt{\frac{8\pi}{15}}\left[\left(1 + K_{2,x} - H_2\right).H_{RadialLoadPoleTideModel}(x) - K_{2,y}.H_{RadialLoadPoleTideModel}(y)\right]\left(x - x_{avg}\right) - A.\sqrt{\frac{8\pi}{15}}\left[\left(1 + K_{2,x} - H_2\right).H_{RadialLoadPoleTideModel}(y) + K_{2,y}.H_{RadialLoadPoleTideModel}(x)\right]\left(y - y_{avg}\right)] \quad (4)$$

and where λ and ϕ are respectively the longitude and latitude of the measurement. x and y are the nearest previous pole location data relative to the altimeter time t.

A is the scaled amplitude factor:

$$A = \frac{\Omega^2 R^2}{g} * \frac{\pi}{180 * 3600} \quad (5)$$

where: Ω = 7.292115e-5 is the nominal earth rotation angular velocity in radian/s

- R = 6378136.6 is the earth radius in m
- g = 398600.4415 is the geocentric gravitational coefficient in km3/s2
- $\frac{\pi}{180*3600}$ is a conversion factor from arc second to radian.
- H2, K2,x and K2,y are Love numbers (H2 = 0.6207, K2,x = 0.3077, K2,y =0.0036).

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | date of estimate (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees for the position at which tide is computed | |
| **lat** | Eigen::VectorXd | Latitude in degrees (positive north) for the position at which tide is computed | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

| type | description |
|---|---|
| **Eigen::VectorXd** | the pole tide height (m). |

### 2.22.2.3. Process method: compute_components

Role:

Same principle as Desai::compute method, see 2.22.2.2, but this method returns the pole tide components (Body, Ocean and Radial Load, in this order) such as:

Pole tide = Body Pole Tide + Ocean Pole Tide + Radial Load Pole Tide

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | date of estimate (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees for the position at which tide is computed | |
| **lat** | Eigen::VectorXd | Latitude in degrees (positive north) for the position at which tide is computed | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

| type | description |
|---|---|
| **Eigen::VectorXd** | Body Pole Tide (m). |
| **Eigen::VectorXd** | Ocean Pole Tide (m). |
| **Eigen::VectorXd** | Radial Load Pole Tide (m). |

### 2.22.3. Implementation example

```
#include""geco/io/tide/pole.hp""
#include""geco/io/ascii/pole.hp""
#include""geco/algorithm/tide/pole.hp""

namespace datetime = geco::datetime;
namespace algo = geco::algorithm::tide::pole;
namespace io = geco::io;

// Read part
// ----------------------------------------
auto model = io::tide::pole::WaveModels<float>(
    POLE_TIDE,""ocean_rea"","ocean_ima"","load_rea"","load_ima"");
auto position = io::ascii::PolePosition(SMM_POL);
// ----------------------------------------


// Algo part (GECO)
// ----------------------------------------
// Initialization
auto date = datetime::utc::Array(1);
auto lon  = Eigen::VectorXd(1);
auto lat  = Eigen::VectorXd(1);

lon << 130.;
lat << 0.;
date << datetime::UTC(2015, 8, 20, 7, 50, 34, 329853);

auto desai = algo::Desai(model, position);
// Compute the nearest interpolation
auto res = desai.compute(date, lon, lat, 1);
// ----------------------------------------
```

## 2.23. Rain rate

The rain rate parameter is a combination of two ECMWF fields: CRR convective rain rate and LSP large scale rain rate, so the reader is the same as for meteorological data.

### 2.23.1. Readers

Include file: geco/io/grib/reduced_gaussian_xy_list.hpp

Class name: ReducedGaussianXYList

Role: Loading into memory a series of grids stored in GRIB format.

Note: The list of grids does not have to be ordered chronologically (the constructor sorts the grids in chronological order).

Warning: It is the use"s responsibility to ensure, if necessary, that the time difference between two grids constituting the time series is constant.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Meshes** | std::list< std::string > | The list of grids to store in memory. | |

| | | | |
|---|---|---|---|
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

## 2.23.2. Algorithm

Include file: geco/algorithm/environment/meteo.hpp

### 2.23.2.1. Class name: RainRate

Role: To interpolate a time series of rain rate fields represented by two reduced Gaussian grids stored in GRIB files that contain Convective Rain Rate and Large Scale Precipitation values. The rain rate is obtained by combination of these two fields (cf. role of the compute method).

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **meshes_CRR** | ReducedGaussianXYSeries | The time series to be taken into account for interpolation (must contain CRR convective rain rate field) | |
| **meshes_LSP** | ReducedGaussianXYSeries | The time series to be taken into account for interpolation (must contain LSP large scale rain) | |
| **bound_error** | Bool | If true, when interpolated values are requested outside of the domain of the input mesh, an exception is raised. If false, then fill_value is used. | false |
| **neighbors** | size_t | number of nearest neighbors to be used for calculating the interpolated value. | 4 |
| **fill_value** | double | The value to use if the request point is outside the time and space interpolation range (implies bound_error false). | NaN |

### 2.23.2.2. Process method: compute

Role: The rain rate is a linear composition of two ECMWF fields (CRR convective rain rate and LSP large scale rain). Each field is taken from the nearest grid in time and spatially interpolated with bilinear method using the four nearest model grid values. The two fields are then combined following the formula:

Rainrate=3600×(LSP+CRR)

Note: The 3600 factor is to convert LSP and CRR from mm/s into mm/h. Rainrate is in mm/h

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | date of estimate (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees for the position at which tide is computed | |
| **lat** | Eigen::VectorXd | Latitude in degrees (positive north) for the position at which tide is computed | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| Type | description |
|---|---|
| **Eigen::VectorXd** | the interpolated rain rate values (in mm/h) at input coordinates and |
| **Eigen::VectorXi** | the minimum number of points used to perform the spatial interpolation |

### 2.23.2.3. Process static method: rain_flag

Role: To compute the rain flag such as::

rain flag = 1 if rain rate > 0.5 mm/h, else 0

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **rain_rate** | Eigen::VectorXd | Rain rate in mm/h | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

Eigen::Matrix<bool, -1, 1>

| Type | description |
|---|---|
| **Eigen::Matrix<bool, -1, 1>** | A vector that contains the rain flag such as 1 means rain, 0 means no-rain. |

### 2.23.3. Implementation example

```
#include""geco/io/grib/reduced_gaussian_xy_list.hp""
#include""geco/algorithm/environment/meteorology.hp""

namespace environment = geco::algorithm::environment;
namespace grib = geco::io::grib;
namespace datetime = geco::datetime;

// Read part
// ----------------------------------------
auto gribs_CRR = std::list<std::string>();
gribs_CRR.push_back(CRR_grib_file_00);
gribs_CRR.push_back(CRR_grib_file_06);
auto CRR_field = grib::ReducedGaussianXYList(gribs_CRR);

auto gribs_LSP = std::list<std::string>();
gribs_LSP.push_back(LSP_grib_file_00);
gribs_LSP.push_back(LSP_grib_file_00);
auto LSP_field = grib::ReducedGaussianXYList(gribs_LSP);
// ----------------------------------------


// Algo part (GECO)
// ----------------------------------------
// Initialization
Eigen::VectorXd grid_data;
Eigen::VectorXi samples;
auto date  = datetime::utc::Array(1);
auto lon   = Eigen::VectorXd(1);
auto lat   = Eigen::VectorXd(1);

lon << 13.2;
lat << -13.5;
date << datetime::UTC(2015, 7, 21, 5, 12, 32, 14);

auto interpolator = environment::RainRate(CRR_field, LSP_field);
```

```
std::tie(grid_data, samples) = interpolator.compute(date, lon, lat, 1);
// ----------------------------------------
```

## 2.24. Sea State Bias

### 2.24.1. Readers

Include file: geco/io/grid/scattered.hpp

Class name: Scattered<float>

Role: To load a discrete grid containing data associated to an ordinate and an abscissa.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **path** | std::string | path the netCDF grid to open | |
| **name** | std::string | netCDF variable that represents the discrete grid | |

### 2.24.2. Algorithm

Include file: geco/algorithm/environment/ssb.hpp

#### 2.24.2.1. Class name: SeaStateBias

Role: To compute the sea state bias (SSB) from a table that is provided as a function of significant wave height and wind speed. The SSB is the difference between the apparent sea level as“"see”" by an altimeter and the actual mean sea level.

Note: If the requested value is outside the swh/wind domain of definition, the returned value is the nearest valid value. The SSB returned value must never be NaN.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **grid** | geometry::grid::Scattered< float > | data on a uniform grid of points defining the table swh/wind with associated ssb value | |

#### 2.24.2.2. Process method: compute

Role: Interpolate over a 2-D grid at the request point in three steps:

select the cell of the four surrounding values of the input point (x,y)

determine the weight of each point of the cell calculated by the interpolation function

compute the interpolate value

Process method interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **swh** | Eigen::VectorXd | sea wave height in m/s | |
| **wind** | Eigen::VectorXd | wind in m/s | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| type | description |
|---|---|
| **Eigen::VectorXd** | the interpolated values at input coordinates |
| **Eigen::VectorXi** | the number of defined points used when calculating the interpolated value |

## 2.24.3. Implementation example

```
#include""geco/io/grid/scattered.hp""
#include« "geco/algorithm/environment/ssb.hp »"

namespace environment = gec o::algorith m::environmen t;
namespace grid = geco::io::grid;

// Read part
// ----------------------------------------
auto scattered = grid::Scattered<float>(SSB,""ss"");
// ----------------------------------------


// Algo part (GECO)
// ----------------------------------------
// Initialization
Eigen::Matrix<double, -1, 1> values;
Eigen::VectorXi samples;
Eigen::VectorXd swh(1);
Eigen::VectorXd alt_wind(1);

swh << 3;
alt_wind << 1.25;

auto interpolator = environment::SeaStateBias(scattered);
```

```
std::tie(values, samples) = interpolator.compute(swh, alt_wind, 1);
// ----------------------------------------
```

## 2.25. Surface Type

The following algorithms have same usage for all the surface type model: DTM2000.1 and GlobCover.

### 2.25.1. Reader

Include file: geco/io/discrete.hpp

Class name: DiscreteXY<int8_t>

Role: To read the netCDF file containing the grid information. Note that the data will be stored as short integer. This class load the whole map by default, but it includes also the advance reading option described in the chapter above (cf. 2.1).

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **path** | std::string | Path to the netCDF grid to open. | |
| **name** | std::string | netCDF variable that represents the discrete grid. | |
| **box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

### 2.25.2. Algorithm

Include file: geco/algorithm/surface/surface_type.hpp

#### 2.25.2.1. Class name: SurfaceType

Role: To determine the surface type at given position by using the nearest value. Note that this method is automatically multi-threaded (can be deactivated).

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **mesh** | geometry::grid::<br>DiscreteXY<int8_t> | Meshing representing the type of surface on earth. | |

### 2.25.2.2. Process method: compute

Role: Determine the nearest value over a 2-D grid at the given position.

Process method interface:

| Parameter | type | Description | Default value (if optional) |
|---|---|---|---|
| **Lon** | Eigen::VectorXd | longitudes in degrees | |
| **Lat** | Eigen::VectorXd | latitudes in degrees | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

A single vector of double.

| Type | Description |
|---|---|
| **Eigen::VectorXd** | contains the surface type at input coordinates. |

## 2.25.3. Implementation example

```
#include""geco/algorithm/surface/surface_type.hp""
#include""geco/io/grid/discrete.hp""

namespace grid = geco::io::grid;
namespace surface = geco::algorithm::surface;

auto data = grid::DiscreteXY<int8_t>(SURFACE_TYPE,""mas"");

Eigen::Matrix<surface::SurfaceType::Kind, -1, 1> values(1);
Eigen::VectorXd lon(1);
Eigen::VectorXd lat(1);

lon << 130.;
lat << 0.;

auto interpolator = surface::SurfaceType(data);
values = interpolator.compute(lon, lat);

// example changing the optional parameters
values = interpolator.compute(lon, lat, num_threads=2);
```

## 2.26. Sea Wave Height

### 2.26.1. Readers

Include file: geco/io/grib/reduced_gaussian_xy_list.hpp

Class name: ReducedGaussianXYList

Role: Loading into memory a series of grids stored in GRIB format.

Note: The list of grids does not have to be ordered chronologically (the constructor sorts the grids in chronological order).

Warning: It is the use''s responsibility to ensure, if necessary, that the time difference between two grids constituting the time series is constant.

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **Meshes** | std::list< std::string > | The list of grids to store in memory. | |
| **Box** | geometry::Box | Geographic area to be loaded into memory. | geometry::Box()<br>Which means that all the content is stored |

### 2.26.2. Algorithm

Include file: geco/algorithm/interpolation/reduced_gaussian_xy_series.hpp

#### 2.26.2.1. Class name: ReducedGaussianXYSeries

Role: To interpolate a time series of meteorological fields represented by a reduced Gaussian grid stored in a GRIB file. The result is obtained by linear interpolation in time between two consecutive data files, and by bilinear interpolation in space from the four nearest model grid values.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **meshes** | ReducedGaussianXYList | The time series to be taken into account for interpolation. | |
| **bound_error** | Bool | If true, when interpolated values are requested outside of the domain of the input mesh, an exception is raised. If false, then fill_value is used. | false |

| | | | |
|---|---|---|---|
| **neighbors** | size_t | number of nearest neighbors to be used for calculating the interpolated value. | 4 |
| **fill_value** | double | The value to use if the request point is outside the time and space interpolation range (implies bound_error false). | NaN |

### 2.26.2.2. Process method: compute

Role: Space-time interpolation (as described in class role) of the time series handled by this instance, for the requested spatial time coordinates request.

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **time** | datetime::utc::Array | date of estimate (UTC) | |
| **lon** | Eigen::VectorXd | Longitude in degrees for the position at which tide is computed | |
| **lat** | Eigen::VectorXd | Latitude in degrees (positive north) for the position at which tide is computed | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXi>

| Type | description |
|---|---|
| **Eigen::VectorXd** | the interpolated values at input coordinates (the unit depends on data read) |
| **Eigen::VectorXi** | the minimum number of points used to perform spatial interpolation on one of the grids. |

### 2.26.3. Implementation example

```
#include""geco/io/grib/reduced_gaussian_xy_list.hp""
#include""geco/algorithm/interpolation/reduced_gaussian_xy_series.hp""

namespace interpolation = geco::algorithm::interpolation;
namespace grib         = geco::io::grib;
namespace datetime     = geco::datetime;

// Read part
auto gribs = std::list<std::string>();
gribs.push_back(SMM_00);
gribs.push_back(SMM_06);
gribs.push_back(SMM_12);
auto grib_fields = grib::ReducedGaussianXYList(gribs);


// Algo part
Eigen::VectorXd grib_data;
Eigen::VectorXi samples;
auto date  = datetime::utc::Array(1);
auto lon   = Eigen::VectorXd(1);
auto lat   = Eigen::VectorXd(1);

lon << 13.2;
lat << -13.5;
date << datetime::UTC(2015, 7, 21, 5, 12, 32, 14);

auto int_press = interpolation::ReducedGaussianXYSeries(grib_fields);
std::tie(grib_data, samples) = int_press.compute(date, lon, lat, 1);
```

## 2.27. Wet tropospheric correction (Ocean)

### 2.27.1. Readers

Cf. 2.26.1

### 2.27.2. Algorithm

Cf. 2.26.2

### 2.27.3. Implementation example

Cf. 2.26.3

## 2.28. Wet tropospheric correction (Hydro)

Cf. 2.8

## 2.29. Wind meridional and zonal component

### 2.29.1. Readers

Cf. 2.26.1

### 2.29.2. Algorithm

Cf. 2.26.2

### 2.29.3. Implementation example

Cf. 2.26.3

## 2.30. xCal error

### 2.30.1. Readers

Include file: geco/io/binary/xcal.hpp

Class name: xCal

Role: This class aims to load a Cross Calibration (xCal) file in netCDF 4 format. At instantiation the file is directly opened, loaded, read and closed. The time variable loaded is time (in UTC).

Constructor interface:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **path** | std::string | the path to the xCal netCDF 4 file. | |

Can also take a list of LR_INT_XCROSSCAL files:

| Parameter | type | description | Default value (if optional) |
|---|---|---|---|
| **paths** | list(std::string) | List of paths to the xCal netCDF 4 file. | |

Warning : if two files have time-tag in common, the reader will keep the one of the last file of the list.

### 2.30.2. Algorithm

Include file: geco/algorithm/interpolation/xcal.hpp

### 2.30.2.1. Class name: xCal

Role: This class aims to interpolate the data contained in the KaRIn crossover calibration product. The goal of the INT_LR_XoverCal product is to provide a correction to minimize systematic errors (timing, range bias, residual roll, phase error, baseline length error) using empirical methods. This product will be used in both the LR and HR processing chains to compute the calibration correction. The crossover correction is applied to the reported surface elevation values in the HR products. On the LR products it is provided as an additional correction but not applied to reported SSH values.

This class has the following methods (Cf. below for further details):

- xcal_quality_flag: to compute the flag value at time measurement
- baseline_errors: LR method, to compute the errors due to baseline dilation
- height_error: LR method, to compute the height errors
- interpolate_param: HR method, to interpolate each xcal product's variables without any combination

"LR method" means that the across-track vector is supposed to be constant in time, baseline and height errors are combination of xcal parameters and across-track vector. HR method only returns interpolated xcal parameters, since the across-track vector is not constant in time.

Constructor interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **xcal** | geometry::XCal | XCal data loaded into memory. | |

### 2.30.2.2. xcal_quality_flag

Role: To compute the associated flag at input time-tag measurement. Xcal input product contains two different flags, so two flags are returned such as:

For both kind of flag:

- if there is no ambiguity between the two flags surrounding the measurement, the same flag is then returned.
- If the input time-tag corresponds to an event transition, which means the two surrounding flags are different, the worst of them (i.e. the highest value) is then returned.

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **date** | datetime::utc::Array | The date UTC on which the baseline correction must be interpolated | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is | 0 |

| | | | |
|---|---|---|---|
| | | given, no parallel computing code is used at all, which is useful for debugging. | |

Parameters are returned as:

A tuple of two Eigen's vector of integers.

| Type | description |
|---|---|
| **Eigen::VectorXi** | The interpolation of range_model_flag at measurement time. Meanings are:<br>0, nominal: Corrections have been well calculated<br>1, out_of_spec: The flag value indicates that there are missing crossover points to calculate a correction consistent with the hydrological specifications of the mission.<br>2, undoubtedly_wrong: The flag value indicates that there is not enough data to calculate the orbital harmonics. Therefore, the correction is undoubtedly wrong.<br>3, do_not_use: The flag value indicates that the interpolator has extrapolated the processed signal. Therefore, do not use the calculated correction.<br>4, replaced_with_zero: The flag value indicates that the undefined values generated by the interpolator have been replaced with zeros.<br>9, missing_measurement: input pass file was missing, no correction can be computed for the entire pass period. Or the time-tag in the computed corrections is missing, corrections will be then set to 0.0 and flagged as missing measurement. |
| **Eigen::VectorXi** | The interpolation of roll_phase_flag at measurement time. Meanings are:<br>0, nominal: Corrections have been well calculated<br>1, out_of_spec: The flag value indicates that there are missing crossover points to calculate a correction consistent with the hydrological specifications of the mission.<br>4, replaced_with_zero: The flag value indicates that the undefined values generated by the interpolator have been replaced with zeros.<br>9, missing_measurement: input pass file was missing, no correction can be computed for the entire pass period. Or the time-tag in the computed corrections is missing, corrections |

| | |
|---|---|
| | will be then set to 0.0 and flagged as missing measurement. |

### 2.30.2.3. LR Process method: baseline_errors

Role: To compute the errors due to baseline dilation to be applied to the measured height. The baseline dilation error can be estimated applying the following formula using the baseline parameter already converted in meters per square kilometer:

$$baseline_{error} = \begin{cases} baseline_{left}.x_{ac}^2 & if\ x_{ac} < 0 \\ baseline_{right}.x_{ac}^2 & else \end{cases}$$

With:

$x_{ac}$: The across-track position $\in [-65;65] km$

$baseline$: Coefficient for quadratic height error (m/km2) due to baseline dilation.

The baseline parameter is taken from SWOT-TN-CDM-0679-CNES-Product_Description_INT_LR_XOverCal and linearly interpolated at input time-tag.

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **date** | datetime::utc::Array | The date UTC on which the baseline correction must be interpolated | |
| **across_track** | Eigen::MatrixXd | The across track position (in Km, between -65 and 65, positive for the right swath, negative for the left) | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Note: The terms “left” and “right” are defined as if standing on the Earth surface at the spacecraft nadir point facing in the direction of the spacecraft velocity vector. The test results suggest that the crossover corrections above is still useful even when the flags (in XCal data) indicate “from_degraded_data”, so flags values are not considered in equations. The baseline dilation error

on the SWOT measurement of SSH, is not expected to be significant compared to the contribution of the linear and inverse slope parameters (timing error).

Parameters are returned as:

| Type | description |
|---|---|
| **Eigen::MatrixXd** | A matrix (date length, across_track length) which contains the baseline dilation errors to be applied to the measured wave height. |

### 2.30.2.4. LR Process method: height_error

Role: To compute the height errors to be applied to the measured wave height. The height errors is a combination of four parameters contained in the XCal product such as:

$$Timing_{error} = \begin{cases} slope_{left}.10^{-3}.x_{ac} + timing_{left} & if\ x_{ac} < 0 \\ slope_{right}.10^{-3}.x_{ac} + timing_{right} & else \end{cases}$$

With:

$x_{ac}$: The across-track position ∈[−65;65]km

$slope_{left}=sl1-sl2$

$slope_{right}=sl1+sl2$

$sl1$: The linear slope across swath.

$sl2$: The inverse slope across swath.

$timing_{left}$: The timing offset error on the left swath (negative cross-track distance).

$timing_{right}$: The timing offset error on the right swath (positive cross-track distance).

Those four last parameters are taken from input SWOT-TN-CDM-0679-CNES-Product_Description_INT_LR_XOverCal data and linearly interpolated at time tag. The factor of 10e−3 encompasses a factor of 10e3 to convert the cross-track distance from kilometers to meters and a factor of 10e−6 to convert the slope from microradians to radians. These errors on both sides of the swath (i.e. left and right) can be applied to correct the measured water height expressed in meters.

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **date** | datetime::utc::Array | The date UTC on which the height error must be interpolated | |
| **across_track** | Eigen::MatrixXd | The across track position (in Km, between -65 and 65, positive for the right swath, negative for the left) | |
| **num_threads** | size_t | The number of threads to use for the | 0 |

| | | | |
|---|---|---|---|
| | | computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | |

Note: The terms “left” and “right” are defined as if standing on the Earth surface at the spacecraft nadir point facing in the direction of the spacecraft velocity vector. The test results suggest that the crossover corrections above is still useful even when the flags (in XCal data) indicate “from_degraded_data”, so flags values are not considered in equations. The baseline dilation error on the SWOT measurement of SSH, is not expected to be significant compared to the contribution of the linear and inverse slope parameters (timing error).

Parameters are returned as:

| Type | description |
|---|---|
| **Eigen::MatrixXd** | A matrix (date length, across_track length) which contains the timing errors to be applied to the measured wave height. |

### 2.30.2.5. HR Process method: interpolate_param

Role: To compute by linear interpolation at input time-tag the following calibration parameters:

- *sl1*: Linear slope (microradians) across swath
- *sl2*: Inverse linear slope (microradians) across swath
- *baseline_left*: Coefficient for quadratic height error (m/km$^2$) due to baseline dilation for the left swath
- *baseline_right*: Coefficient for quadratic height error (m/km$^2$) due to baseline dilation for the right swath
- $timing_{left}$: timing error on the left (negative cross-track distance) swath.
- $timing_{right}$: timing error on the right (positive cross-track distance) swath

Process method interface:

| Parameter | Type | description | Default value (if optional) |
|---|---|---|---|
| **date** | datetime::utc::Array | The date UTC on which the parameters must be interpolated | |
| **num_threads** | size_t | The number of threads to use for the computation. If 0 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. | 0 |

Note: The terms “left” and “right” are defined as if standing on the Earth surface at the spacecraft nadir point facing in the direction of the spacecraft velocity vector. The test results suggest that the

crossover corrections above is still useful even when the flags (in XCal data) indicate “from_degraded_data”, so flags values are not considered in equations. The baseline dilation error on the SWOT measurement of SSH, is not expected to be significant compared to the contribution of the linear and inverse slope parameters (timing error).

Parameters are returned as:

std::tuple<Eigen::VectorXd, Eigen::VectorXd, Eigen::VectorXd, Eigen::VectorXd, Eigen::VectorXd >

| Type | Description |
|---|---|
| **Eigen::VectorXd** | The linear slope (microradians) across swath. |
| **Eigen::VectorXd** | The inverse linear slope (microradians) across swath. |
| **Eigen::VectorXd** | The coefficient for quadratic height error ($m/km^2$) due to baseline dilation for the left swath. |
| **Eigen::VectorXd** | The coefficient for quadratic height error ($m/km^2$) due to baseline dilation for the right swath. |
| **Eigen::VectorXd** | The timing error (meters) on the right (positive cross-track distance) swath. |
| **Eigen::VectorXd** | The timing error (meters) on the left (negative cross-track distance) swath. |

## 2.30.3. Implementation example

```
#include "geco/algorithm/interpolation/xcal.hpp"
#include "geco/datetime/utc.hpp"
#include "geco/io/binary/xcal.hpp"

int main(int argc, char** argv) {
  if (argc != 2) {
    std::cerr << "Usage: " << argv[0] << " <xcal>" << std::endl;
    return 1;
  }

  // read data
  auto data = geco::io::binary::XCal(argv[1]);

  // inputs
  auto dates = geco::datetime::utc::Array(1);
  dates[0] =
      geco::datetime::UTC(2022, 7, 31, 4, 12, 51, 935900);
  auto Eigen::MatrixXd across(2, 60);
  across.row(0) = Eigen::VectorXd::LinSpaced(60, 5, 65);
  across.row(1) = Eigen::VectorXd::LinSpaced(60, 5, 65);

  // algorithm
  Eigen::VectorXi range_model_flag, roll_phase_flag;
  Eigen::VectorXd sl1, sl2, baseline_left, baseline_right, timing_left,
timing_right;
  auto interpolator = geco::algorithm::interpolation::XCal(data);
    Eigen::MatrixXd timing_err, baseline_err;
  std::tie(range_model_flag, roll_phase_flag) =
interpolator.xcal_quality_flag(dates);
  auto timing_err = interpolator.height_error(dates, across, 2);
  auto baseline_err = interpolator.baseline_errors(dates, across, 2);
  std::tie(sl1, sl2, baseline_left, baseline_right, timing_right, timing_left) =
    interpolator.interpolate_param(dates, 2);
  return 0;
}
```

## ANNEXE 1 – Equations of dry and wet tropospheric corrections w.r.t. the real surface height

***Definitions of the refractive index and of the wet and dry tropospheric corrections***

The excess propagation path, also called path delay, induced by the neutral gases of the atmosphere between the backscattering surface and the satellite is given by:

$$\delta h = \int_{Hsurf}^{Hsat} (n(z) - 1)\, dz \qquad (1)$$

where n(z) is the index of refraction of air, $H_{surf}$ and $H_{sat}$ are respectively the altitudes of the surface and of the satellite above mean sea level.

The index of refraction is conveniently expressed in terms of the refractivity N(z), defined as:

$$10^{-6} N(z) = n(z) - 1 \qquad (2)$$

N(z) is given by Bean and Dutton (1966):

$$N(z) = 77.6 \frac{P_d}{T} + 72 \frac{e}{T} + 3.75\ 10^5 \frac{e}{T^2} \qquad (3)$$

where $P_d$ is the partial pressure of dry air in hPa (1 hPa = 100 Pa), e is the partial pressure of water vapor in hPa, and T is temperature in K.

As the partial pressure of dry air is not easily measured, it is desirable to obtain an expression function of the total pressure of air. For deriving it, we have to consider that the dry air and the water vapor are ideal gases, i.e., they obey to the Mariotte-Gay Lussac law:

$$\text{For dry air: } \frac{P_d}{\rho_d} = \frac{RT}{M_d} \qquad (4)$$

$$\text{For water vapor: } \frac{e}{\rho_w} = \frac{RT}{M_w} \qquad (5)$$

where $\rho_d$ and $\rho_w$ are the volumic masses of dry air and water vapor respectively, $M_d$ and $M_w$ are the molar masses of dry air ($28.9644\ 10^{-3}$ kg) and water vapor ($18.0153\ 10^{-3}$ kg) respectively, R is the universal gas constant (8.31434 $J.mole^{-1}.K^{-1}$).

Combining (4), (5) and (3) leads to:

$$N(z) = 77.6\ R \frac{\rho_d}{M_d} + 72\ R \frac{\rho_w}{M_w} + 3.75\ 10^5 \frac{e}{T^2} \qquad (6)$$

The volumic mass of wet air is the sum of the volumic masses of dry air and water vapor:

$$\rho = \rho_d + \rho_w \qquad (7)$$

Introducing the volumic mass of wet air given by (7) into (6) leads to:

$$N(z) = 77.6\ R \frac{\rho}{M_d} + (72 - 77.6 \frac{M_w}{M_d})\ R \frac{\rho_w}{M_w} + 3.75\ 10^5 \frac{e}{T^2} \qquad (8)$$

Reintroducing (5) into (8) leads to the final expression of refractivity N(z):

$$N(z) = 77.6\ R \frac{\rho}{M_d} + (72 - 77.6 \frac{M_w}{M_d}) \frac{e}{T} + 3.75\ 10^5 \frac{e}{T^2} \qquad (9)$$

Combining this expression with (1) and (2) leads to the following equation for $\delta h$:

$$\delta h = 77.6\ 10^{-6}\ \frac{R}{M_d}\int_{Hsurf}^{Hsat}\rho\,dz + (72 - 77.6\ \frac{M_w}{M_d})10^{-6}\int_{Hsurf}^{Hsat}\frac{e}{T}dz + 3.75\ 10^{-1}\int_{Hsurf}^{Hsat}\frac{e}{T^2}dz \quad (10)$$

The first term is called the dry tropospheric correction $\delta h_{dry}$:

$$\delta h_{dry} = 77.6\ 10^{-6}\ \frac{R}{M_d}\int_{Hsurf}^{Hsat}\rho\,dz \quad (11)$$

The sum of the two remaining terms is called the wet tropospheric correction $\delta h_{wet}$:

$$\delta h_{wet} = (72 - 77.6\ \frac{M_w}{M_d})10^{-6}\int_{Hsurf}^{Hsat}\frac{e}{T}dz + 3.75\ 10^{-1}\int_{Hsurf}^{Hsat}\frac{e}{T^2}dz \quad (12)$$

Introducing the numerical values for $M_d$ and $M_w$ into (12), and multiplying $\delta h_{wet}$ by -1 to get a negative quantity to be added to the altimeter range, leads to the following equation for $\delta h_{wet}$ in m:

$$\delta h_{wet} = -23.7\ 10^{-6}\int_{Hsurf}^{Hsat}\frac{e}{T}dz - 3.75\ 10^{-1}\int_{Hsurf}^{Hsat}\frac{e}{T^2}dz \quad (13)$$

***Calculation of the dry tropospheric correction as function of the surface pressure***

It is commonly assumed that the atmosphere is in hydrostatic equilibrium, i.e. g being the acceleration due to gravity:

$$\frac{dP}{dz} = -\rho\,g \quad (14)$$

Combining (11) and (14) leads to the following equation for $\delta h_{dry}$, where $P_{surf}$ is the atmospheric pressure at the ground surface:

$$\delta h_{dry} = 77.6\ 10^{-6}\ \frac{R}{M_d}\int_{0}^{Psurf}\frac{1}{g}dP \quad (15)$$

***Calculation of the wet tropospheric correction map at Météo-France:***

The humidity variable output by the model is the specific humidity q, given by :

$$q = \frac{\rho_w}{\rho_d + \rho_w} \quad (20)$$

Thus, (13) must be rewritten as function of q. After introducing (5), (7), (14) and (20) into (13), one gets:

$$\delta h_{wet} = -23.7\ 10^{-6}\ \frac{R}{M_w}\int_{Psat}^{Psurf}\frac{1}{g}q\,dp - 3.75\ 10^{-1}\ \frac{R}{M_w}\int_{Psat}^{Psurf}\frac{1}{g}\frac{q}{T}dp \quad (21)$$

## ANNEXE 2 - Pseudo-code of TEC grids interpolation

```
// Defining epoch of the two surrounding grids (t0 and t1) and time delta.
// If requested time-tag (date) is after the last grid, and difference < delta_t,
// then t1 and i1 refers to the last grid, and t0 and i0 the previous one.
delta = (t1 - t0);

// Compute grid indices within the first Map (after rotation)
// lon refers to longitude and lat to latitude in degrees
shift =fractional_part((date - t0) / delta);
x = lon + shift * 30.0;
if (x < -180.0) {
  x += 360.0;
} else if (x > 180.0) {
  x -= 360.0;
}
x = (x + 180.0) / 5.0;
ix = x;
y = (lat + 87.5) / 2.5;
iy = y;

if (iy == TEC_ROWS - 1) {
  // If upper limit of the GIM grid, move down one GIM index in latitude
  --iy;
} else if (iy >= TEC_ROWS || iy < 0) {
  // The point is located outside the grid in latitude
  return std::numeric_limits<double>::quiet_NaN();
}

// Calculation of the weight of the points of the bilinear interpolation.
dx = x - ix;
dy = y - iy;
wll = (1 - dx) * (1 - dy);
wlr = dx * (1 - dy);
wul = (1 - dx) * dy;
wur = dx * dy;

// Interpolate the TEC in the first map
tec_map1 = maps_[i0];
tec1 = (wll * tec_map1(iy, ix) + wlr * tec_map1(iy, (ix + 1) % TEC_COLS) +
             wul * tec_map1(iy + 1, ix) +
             wur * tec_map1(iy + 1, (ix + 1) % TEC_COLS));

// Rotate the X coordinate in the second Map and interpolate
ix -= 6;  // NOLINT
if (ix < 0) {
  ix += (TEC_COLS - 1);
}

// Interpolate the TEC in the next map
tec_map2 = maps_[i1];
tec2 = (wll * tec_map2(iy, ix) + wlr * tec_map2(iy, ix + 1) +
             wul * tec_map2(iy + 1, ix) + wur * tec_map2(iy + 1, ix + 1));

// Finally, interpolate in time between the two values
return 0.1 * (tec1 * (1. - shift) + tec2 * shift);
```

## List of acronyms

| | |
|---|---|
| TBC | To be confirmed |
| TBD | To be defined |
| AD | Applicable Document |
| RD | Reference Document |
| DAC | Dynamic Atmospheric Correction |
| ECMWF | European Centre for Medium-Range Weather Forecasts |
| GIM | Global Ionospheric Maps |
| GRIB | GRIdded Binary |
| TAI | International Atomic Time |
| SSB | Sea State Bias |
| MSS | Mean Sea Surface |
| MDT | Mean Dynamic Topography |
| netCDF | Network Common Data Form defined by Unidata group |
| UTC | Universal Time Coordinated |