OGGM Documentation

Release 1.0.0

OGGM Developers

Jan 16, 2018

Principles

Extending Marzeion et al., (2012), the model accounts for glacier geometry (including contributory branches) and includes an explicit ice dynamics module. It can simulate past and future mass-balance, volume and geometry of (almost) any glacier in the world in a fully automated workflow. We rely exclusively on publicly available data for calibration and validation.

This is the model's documentation for users and developers as of version 1.0. For the documentation of the latest (cutting-edge) repository version, visit oggm.readthedocs.io/en/latest.

For more information about the OGGM project and for the latest news, visit oggm.org.

CHAPTER 1

Index

1.1 Principles

1.1.1 Introduction

We illustrate with an example how the OGGM workflow is applied to the Tasman Glacier in New Zealand. Here we describe shortly the purpose of each processing step, while more details are provided in the other sections:

Preprocessing The glacier outlines extracted from the RGI are projected onto a local gridded map of the glacier (Fig. a). Depending on the glacier's location, a suitable source for the topographical data is downloaded automatically (here SRTM) and interpolated to the local grid. The map's spatial resolution depends on the size of the glacier.

Flowlines The glacier centerlines are computed using a geometrical routing algorithm and then filtered and slightly adapted modified to become glacier "flowlines" with a fixed grid spacing (Fig. c).

Catchment areas and widths The geometrical widths along the flowlines are obtained by intersecting the normals at each grid point with the glacier outlines and the tributaries' catchment areas. Each tributary and the main flowline has a catchment area, which is then used to correct the geometrical widths so that the flowline representation of the glacier is in close accordance with the actual altitude-area distribution of the glacier (Fig. d, note that the normals are now corrected and centred).

Climate data and mass-balance Gridded climate data (monthly temperature and precipitation) are interpolated to the glacier location and corrected for altitude at each flowline's grid point. A carefully calibrated temperature-index model is used to compute the mass-balance for any month in the past.

Ice thickness inversion Using the mass-balance data computed above and relying on mass-conservation considerations, an estimate of the ice flux along each glacier cross-section can be computed. by making assumptions about the shape of the cross-section (parabolic or rectangular) and using the physics of ice flow, the model computes the thickness of the glacier along the flowlines and the total volume of the glacier (Fig. e).

Glacier evolution A dynamical flowline model is used to simulate the advance and retreat of the glacier under preselected climate time series. Here (Fig. f), a 120-yrs long random climate sequence leads to a glacier advance.

1.1.2 Glacier flowlines

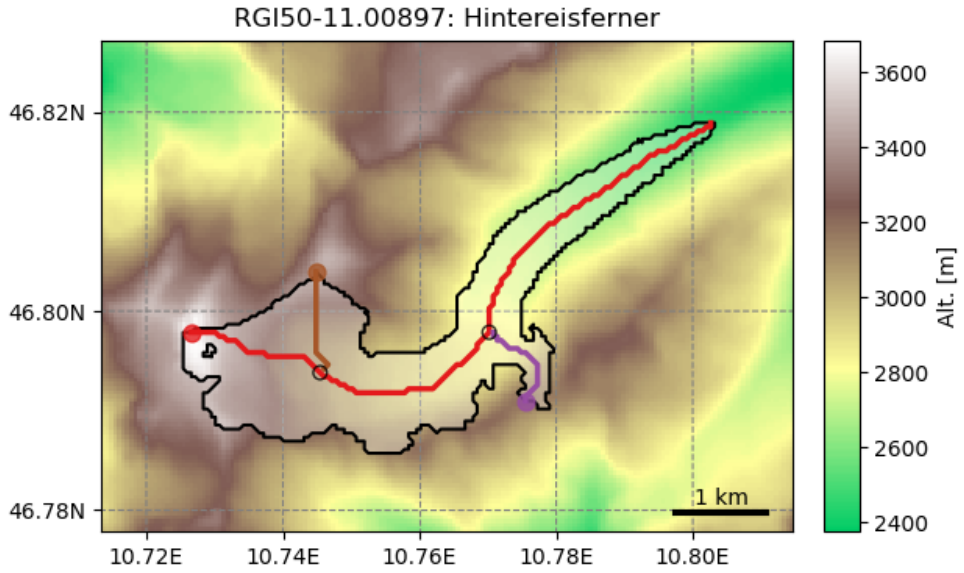
Centerlines

Computing the centerlines is the first task to run after the initialisation of the local glacier directories and of the local topography.

Our algorithm is an implementation of the procedure described by Kienholz et al., (2014). Appart from some minor changes (mostly the choice of certain parameters), we stayed close to the original algorithm.

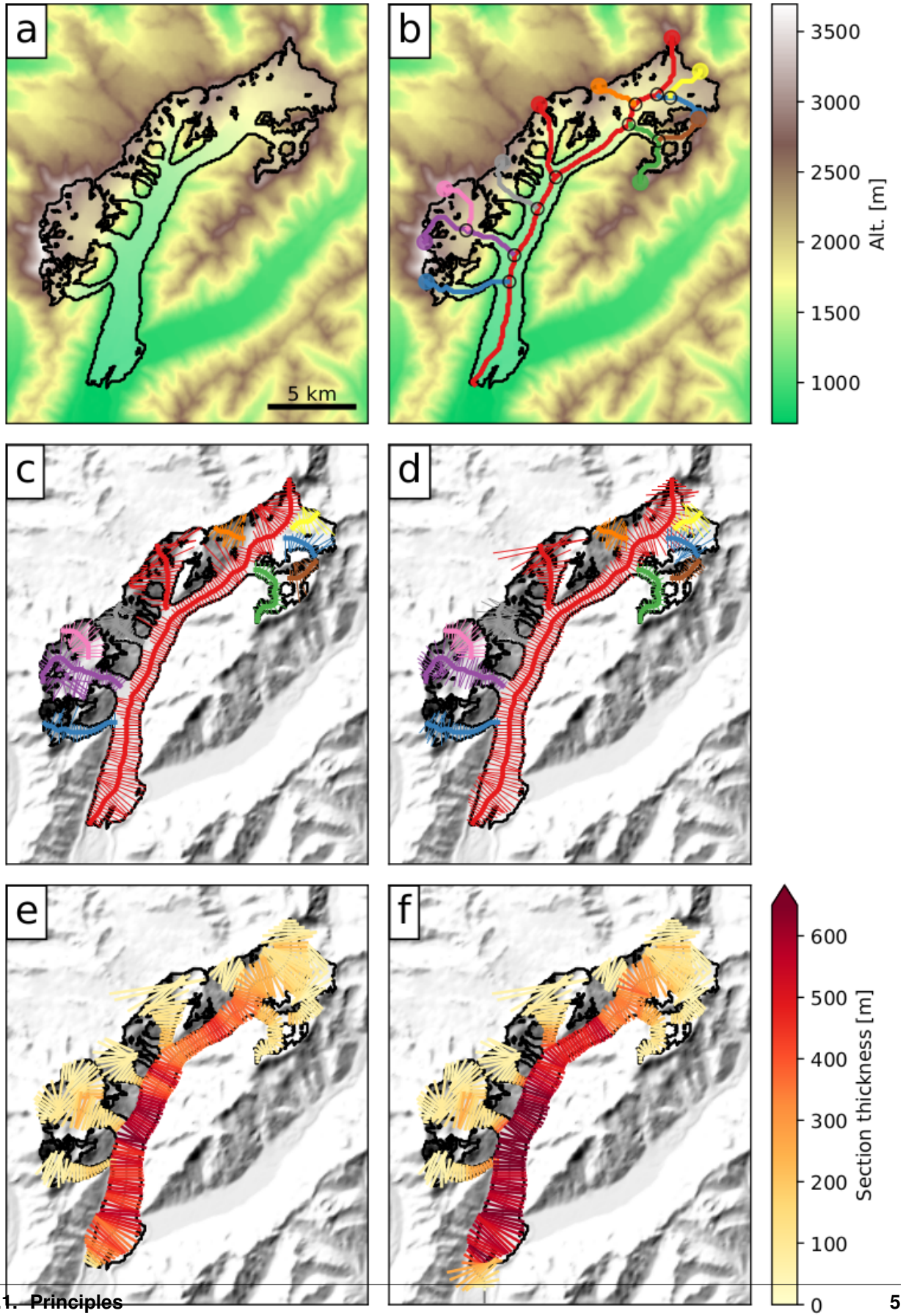
The relevant task is *tasks.compute_centerlines()*:

```
In [1]: graphics.plot_centerlines(gdir)
```



The glacier has a major centerline (the longest one), and tributaries (in this case two). The *Centerline* objects are stored as a list, the last one being the major one. Navigation between inflows (can be more than one) and outflow (only one or none) is facilitated by the `inflows` and `flows_to` attributes:

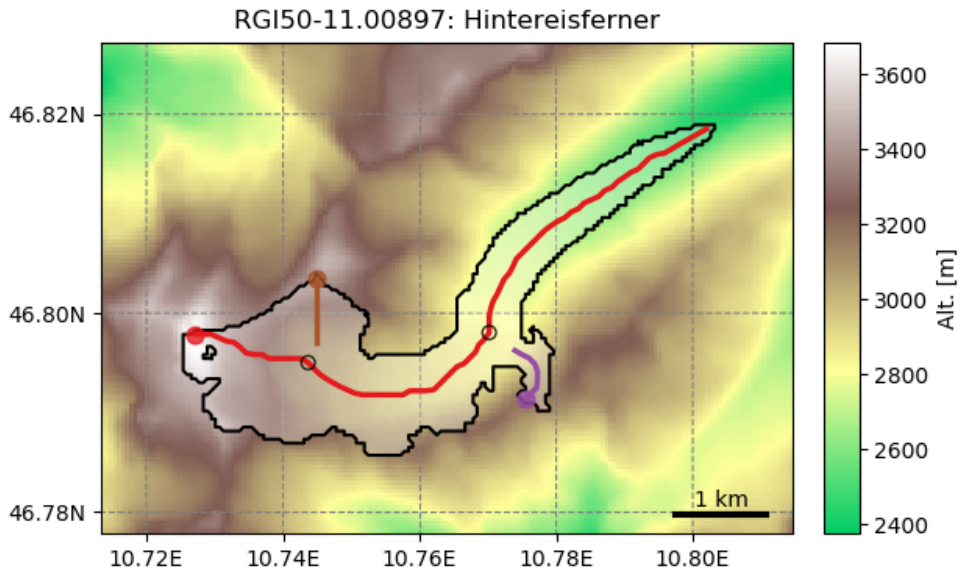
```
In [2]: fls = gdir.read_pickle('centerlines')

In [3]: fls[0]  # a Centerline object
Out[3]: <oggm.core.centerlines.Centerline at 0x7fa64eb4a978>
```


a
b
c
d
e
f
5 km
3500
3000
2500
2000
1500
1000
Alt. [m]
600
500
400
300
200
100
0
Section thickness [m]

```
# make sure the first flowline realy flows into the major one:
In [4]: assert fls[0].flows_to is fls[-1]
```

At this stage, the centerline coordinates are still defined on the original grid, and they are not considered as "flowlines" by OGGM. A rather simple task (*tasks.initialize_flowlines()*) converts them to flowlines which now have a regular coordinate spacing along the flowline (which they will keep for the rest of the workflow). The tail of the tributaries are cut according to a distance threshold rule:

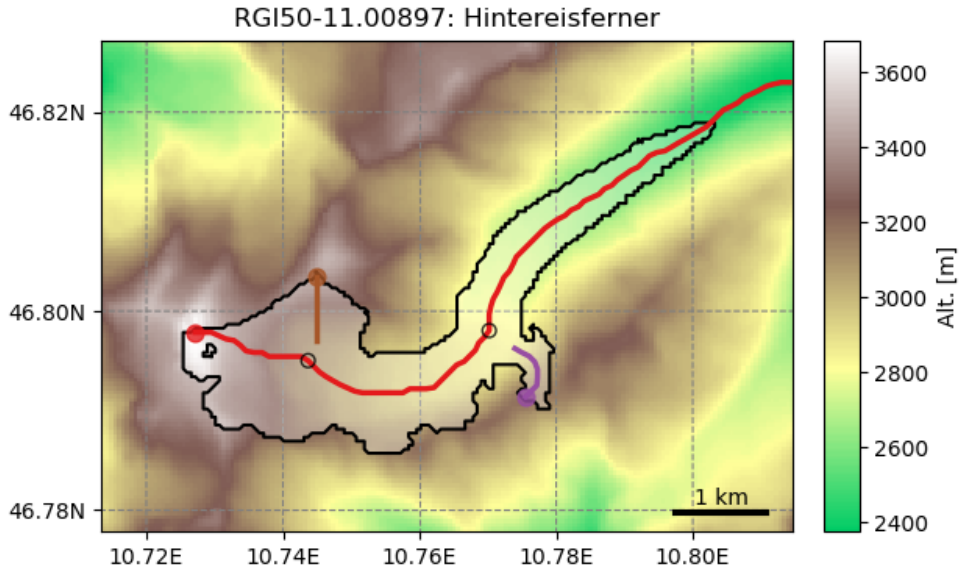
```
In [5]: graphics.plot_centerlines(gdir, use_flowlines=True)
```



Downstream lines

For the glacier to be able to grow we need to determine the flowlines downstream of the current glacier geometry. This is done by the *tasks.compute_downstream_line()* task:

```
In [6]: graphics.plot_centerlines(gdir, use_flowlines=True, add_downstream=True)
```

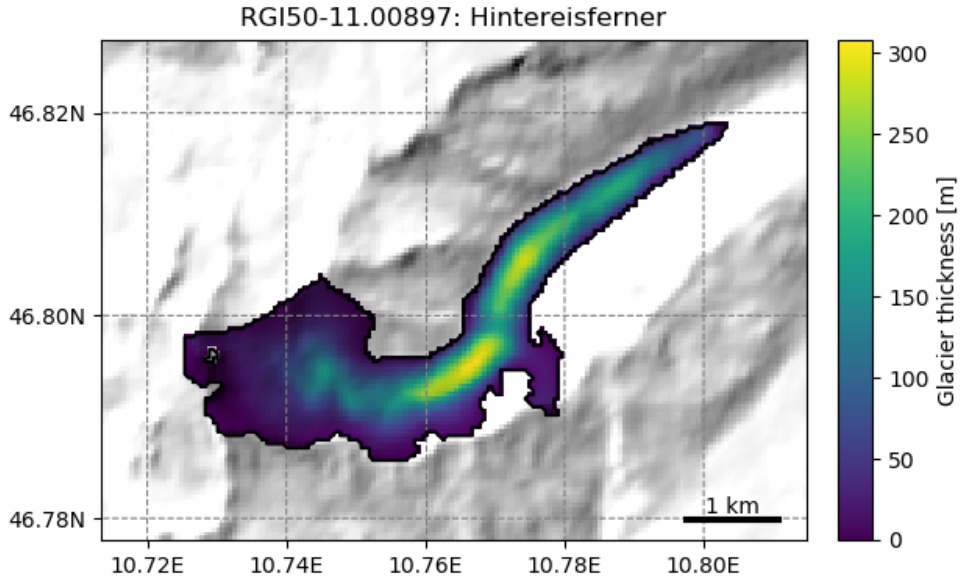


The downsteam lines area also computed using a routing algorithm minimizing the distance to cover and upward slopes.

Catchment areas

Each flowline has it's own "catchment area". These areas are computed using similar flow routing methods as the one used for determining the flowlines. Their role is to attribute each glacier pixel to the right tributory (this will also influence the later computation of the glacier widths).

```
In [7]: tasks.catchment_area(gdir)

In [8]: graphics.plot_catchment_areas(gdir)
```

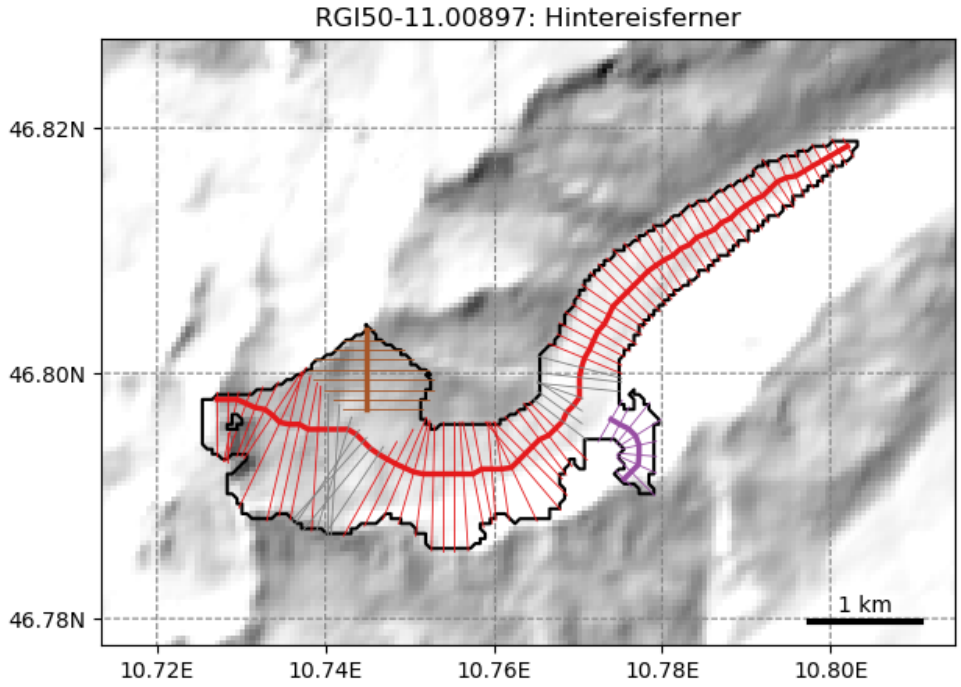


Flowline widths

Finally, the glacier widths are computed in two steps.

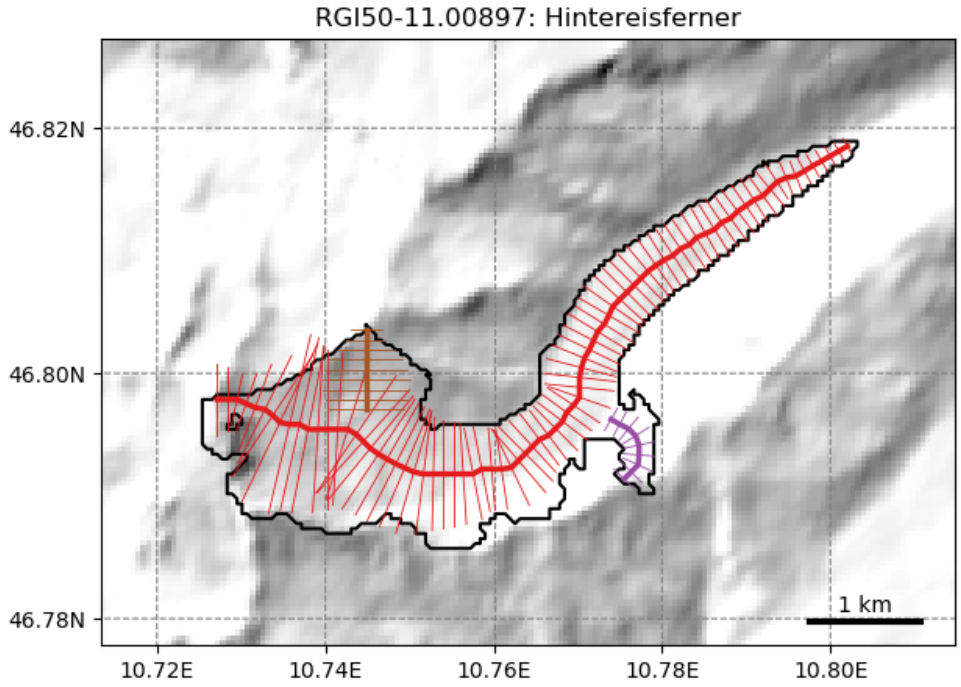
First, we compute the geometrical width at each grid point. The width is drawn from the intersection of a line normal to the flowline and either the glacier or the catchment outlines (when there are tributaries):

```
In [9]: tasks.catchment_width_geom(gdir)

In [10]: graphics.plot_catchment_width(gdir)
```



Then, these geometrical widths are corrected so that the altitude-area distribution of the "flowline-glacier" is as close as possible as the actual distribution of the glacier using its full 2D geometry. This job is done by the *tasks.catchment_width_correction()* task:

```
In [11]: tasks.catchment_width_correction(gdir)

In [12]: graphics.plot_catchment_width(gdir, corrected=True)
```



Note that a perfect distribution is not possible since the sample size is not the same between the "1.5D" and the 2D representation of the glacier. OGGM deals with this by iteratively search for an altidute bin size which ensures that both representations have at least one element for each bin.

Implementation details

Shared setup for these examples:

```
import os
import geopandas as gpd
import oggm
from oggm import cfg, tasks, graphics
from oggm.utils import get_demo_file

cfg.initialize()
cfg.set_intersects_db(get_demo_file('rgi_intersect_oetztal.shp'))
cfg.PATHS['dem_file'] = get_demo_file('hef_srtm.tif')

base_dir = os.path.join(os.path.expanduser('~'), 'OGGM_docs', 'Flowlines')
entity = gpd.read_file(get_demo_file('HEF_MajDivide.shp')).iloc[0]
gdir = oggm.GlacierDirectory(entity, base_dir=base_dir)

tasks.define_glacier_region(gdir, entity=entity)
tasks.glacier_masks(gdir)
tasks.compute_centerlines(gdir)
tasks.initialize_flowlines(gdir)
tasks.compute_downstream_line(gdir)
```

1.1.3 Mass-balance

The mass-balance (MB) model implemented in OGGM is an extended version of the temperature index model presented by Marzeion et al., (2012). While the equation governing the mass-balance is that of a traditional temperature index model, our special approach to calibration requires that we spend some time describing it.

Climate data

The MB model implemented in OGGM needs monthly time series of temperature and precipitation. The current default is to download and use the CRU TS v3.24 data provided by the Climatic Research Unit of the University of East Anglia.

CRU (default)

If not specified otherwise, OGGM will automatically download and unpack the latest dataset from the CRU servers.

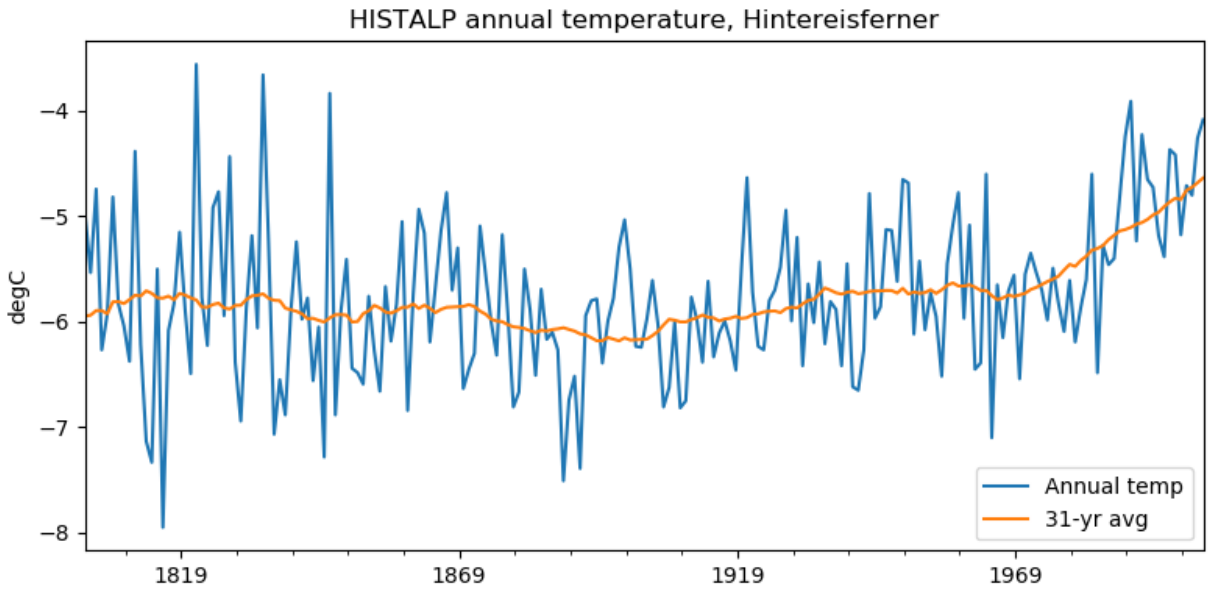
> **Warning:** While the downloaded zip files are ~370mb in size, they are ~5.6Gb large after decompression!

The raw, coarse (0.5°) dataset is then downscaled to a higher resolution grid (CRU CL v2.0 at 10' resolution) following the anomaly mapping approach described by Tim Mitchell in his CRU faq (Q25). Note that we don't expect this downscaling to add any new information than already available at the original resolution, but this allows us to have an elevation-dependent dataset, from which we can compute the temperature at the elevation of the glacier.

User-provided dataset

You can provide any other dataset to OGGM by setting the `climate_file` parameter in `params.cfg`. See the HISTALP data file in the sample-data folder for an example.

```
In [1]: example_plot_temp_ts()  # the code for these examples is posted below
```



Elevation dependency

OGGM finally needs to compute the temperature and precipitation at the altitude of the glacier grid points. The default is to use a fixed gradient of -6.5K km $^{-1}$ and no gradient for precipitation. However, OGGM implements a module which computes the local gradient by linear regression of the 9 surrounding grid points. This method requires that the near-surface temperature lapse-rates provided by the climate dataset are good (in most of the cases you should probably use a fixed gradient). The default config parameters are:

```
In [2]: cfg.PARAMS['temp_use_local_gradient']  # use the regression method?
Out[2]: False

In [3]: cfg.PARAMS['temp_default_gradient']  # constant gradiant
\\\\\\\\\\\\\\Out[3]: -0.0065
```

Temperature index model

The monthly mass-balance B_i at elevation z is computed as:

$$B_i(z) = P_i^{Solid}(z) - \mu^* \, max\left(T_i(z) - T_{Melt}, 0\right)$$

where P_i^{Solid} is the monthly solid precipitation, T_i the monthly temperature and T_{Melt} is the monthly mean air temperature above which ice melt is assumed to occur (0°C per default). Solid precipitation is computed out of the total precipitation. The fraction of solid precipitation is based on the monthly mean temperature: all solid below `temp_all_solid` (default: 0°C) all liquid above `temp_all_liq` (default: 2°C), linear in between.

The parameter μ^* indicates the temperature sensitivity of the glacier, and it needs to be calibrated.

Calibration

We will start by making two observations:

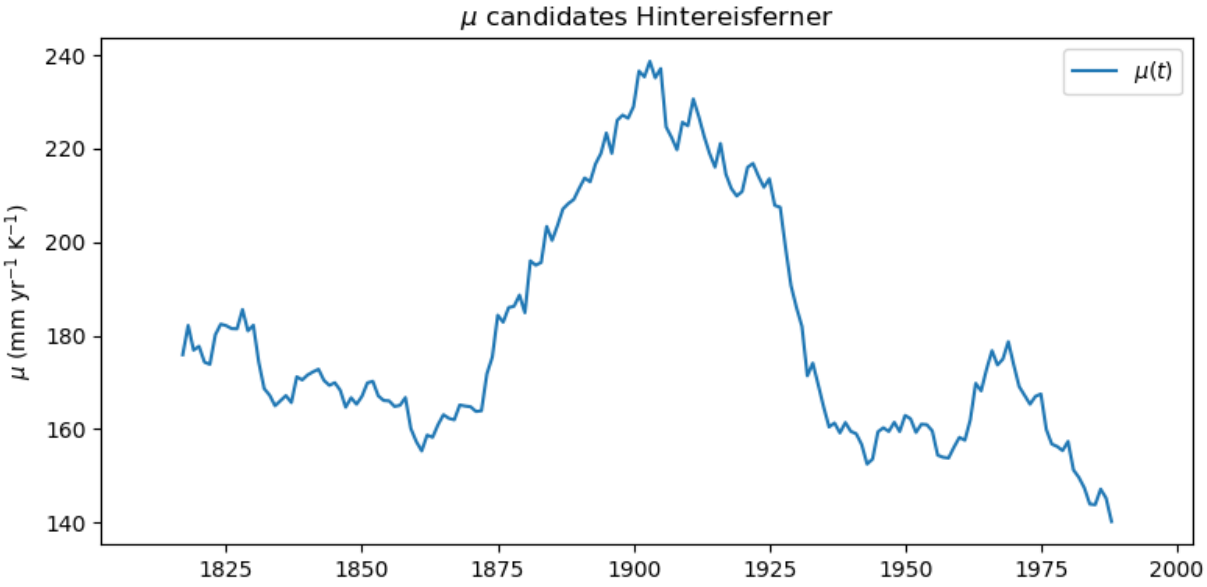
- the sensitivity parameter μ^* is depending on many parameters, most of them being glacier-specific (e.g. avalanches, topographical shading, cloudiness…).
- the sensitivity parameter μ^* will be affected by uncertainties and systematic biases in the input climate data.

As a result, μ^* can vary greatly between neighboring glaciers. The calibration procedure introduced by Marzeion et al., (2012) and implemented in OGGM makes full use of these apparent handicaps by turning them into assets.

The calibration procedure starts with glaciers for which we have direct observations of the annual specific mass-balance SMB. We use the WGMS FoG (shipped with OGGM) for this purpose.

For each of these glaciers, time-dependent "candidate" temperature sensitivities $\mu(t)$ are estimated by requiring that the average specific mass-balance B_{31} is equal to zero. B_{31} is computed for a 31 yr period centered around the year t **and for a constant glacier geometry fixed at the RGI date** (e.g. 2003 for most glaciers in the European Alps).

```
In [4]: example_plot_mu_ts()  # the code for these examples is posted below
```

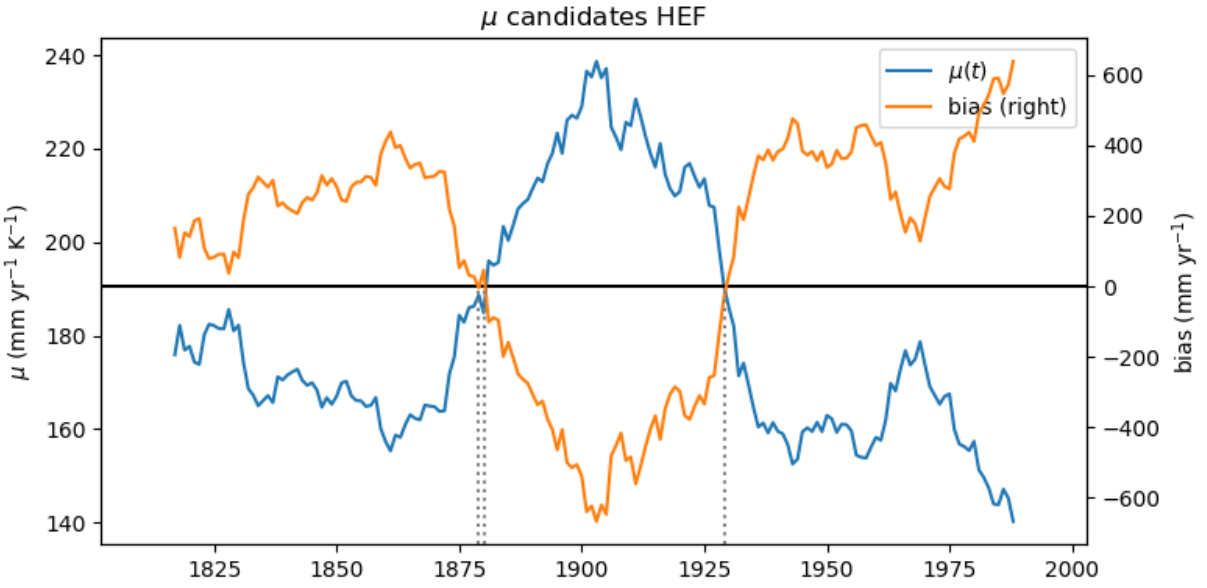


Around 1900, the climate was cold and wet. As a consequence, the temperature sensitivity required to maintain the 2003 glacier geometry is high. Inversely, the recent climate is warm and the glacier must have a small temperature sensitivity in order to preserve its geometry.

Note that these $\mu(t)$ are just hypothetical sensitivities necessary to maintain the glacier in equilibrium in an average climate at the year t. We call them "candidates", since one (or more) of them is likely to be close to the "real" sensitivity of the glacier.

This is when the mass-balance observations come into play: each of these candidates can be used to compute the mass-balance during the period were we have observations. We then compare the model output with the expected mass-balance and compute the model bias:

```
In [5]: example_plot_bias_ts()  # the code for these examples is posted below
```



The bias is positive when μ is too low, and negative when μ is too high. The vertical dashed lines mark the times

where the bias is closest to zero. They all correspond to approximately the same μ (but not exactly, as precipitation and temperature both influence μ). These dates at which the μ candidates are close to the real μ are called t^* (the associated sensitivities $\mu(t^*)$ are called μ^*).

At the glaciers where observations are available, this detour via the μ candidates is not necessary to find the correct μ^*. Indeed, the goal of these computations are in fact to find t^*, **which is the actual value to be interpolated to glaciers where no observations are available**.

The benefit of this approach is best shown with the results of a cross-validation study realized by Marzeion et al., (2012) (and confirmed by OGGM):

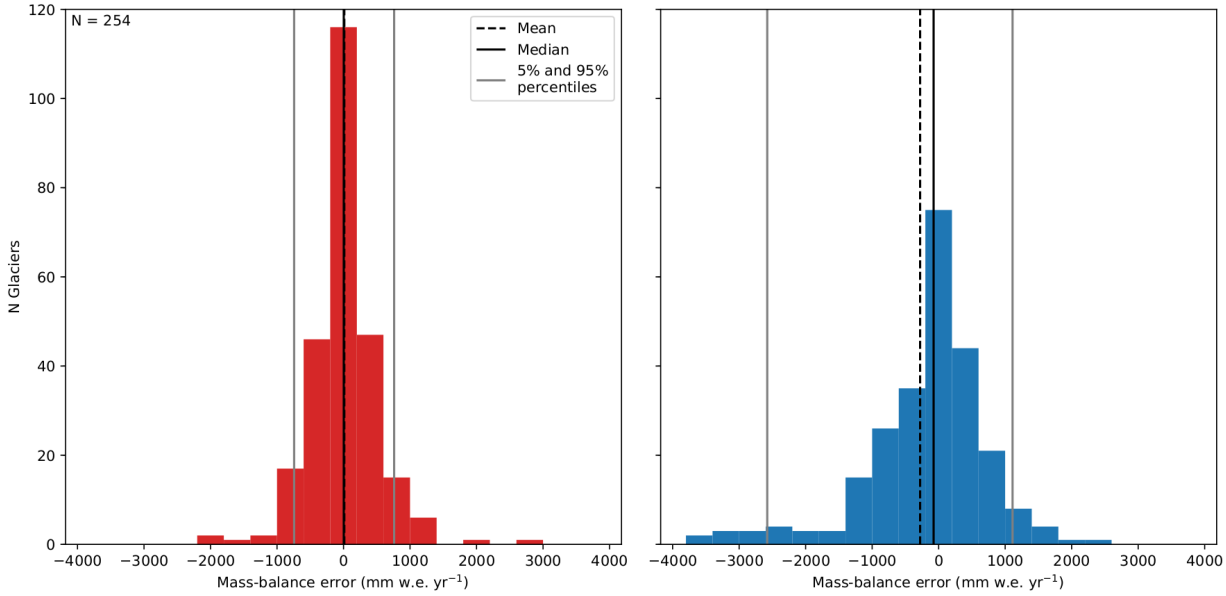


Fig. 1.1: Benefit of spatially interpolating t^* instead of μ^* as shown by leave-one-glacier-out cross-validation (N = 255). **Left**: error distribution of the computed mass-balance if determined by the interpolated t^*. **Right**: error distribution of the mass-balance if determined by interpolation of μ^*.

This substantial improvement in model performance is due to several factors:

- the equilibrium constraint applied on μ implies that the sensitivity cannot vary much during the last century. In fact, μ at one glacier varies far less in one century than between neighboring glaciers, because of all the factors mentioned above. In particular, it will vary comparatively little around a given year t : errors in t^* (even large) will result in small errors in μ^*.
- the equilibrium constraint will also imply that systematic biases in temperature and precipitation (no matter how large) will automatically be compensated by all $\mu(t)$, and therefore also by μ^*. In that sense, the calibration procedure can be seen as a empirically driven downscaling strategy: if a glacier is here, then the local climate (or the glacier temperature sensitivity) *must* allow a glacier to be there. For example, the effect of avalanches or a negative bias in precipitation input will have the same impact on calibration: μ^* should be reduced to take these effects into account, even though they are not resolved by the mass-balance model.

The most important drawback of this calibration method is that it assumes that two neighboring glaciers should have a similar t^*. This is not necessarily the case, as other factors than climate (such as the glacier size) will influence t^* too. Our results (and the arguments listed above) show however that this is an approximation we can cope with.

In a final note, it is important to mention that the μ^* and t^* should not be over-interpreted in terms of "real" temperature sensitivities or "real" response time of the glacier. This procedure is primarily a calibration method, and as such it can be statistically scrutinized (for example with cross-validation). It can also be noted that the MB observations play a

relatively minor role in the calibration: they could be entirely avoided by fixing a t^* for all glaciers in a region (or even worldwide). The resulting changes in calibrated μ^* will be comparatively small (again, because of the local constraints on μ). The MB observations, however, play a major role for the assessment of model uncertainty.

Implementation details

If you had the courage to read until here, it means that you have concrete questions about the implementation of the mass-balance model in OGGM. Here are some more details:

- the mass-balance in OGGM is computed from the altitudes and widths of the flowlines grid points (see *Glacier flowlines*). The easiest way to let OGGM compute the mass-balance for you is to use the *core.massbalance.PastMassBalance*.
- the interpolation of t^* is done with an inverse distance weighting algorithm (see *tasks.distribute_t_stars()*)
- if more than one t^* is found for some reference glaciers, than the glaciers with only one t^* will determine the most likely t^* for the other glaciers (see *tasks.compute_ref_t_stars()*)
- yes, the temperature gradients and the precipitation scaling factor will have an influence on the results, but it is small since any change will automatically be compensated by μ^*. We are currently quantifying these effects more precisely.
- the cross-validation procedure is done systematically (*tasks.crossval_t_stars()*). Currently, this cross-validation is done with fixed geometry (it will be extended to a full validation with the dynamical model at a later stage).

Code used to generate these examples:

```
import os

import geopandas as gpd
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import xarray as xr

import oggm
from oggm import cfg, tasks
from oggm.core.climate import (mb_yearly_climate_on_glacier,
                               t_star_from_refmb,
                               local_mustar, apparent_mb)
from oggm.core.massbalance import (ConstantMassBalance)
from oggm.utils import get_demo_file
from oggm import graphics

cfg.initialize()
cfg.set_intersects_db(get_demo_file('rgi_intersect_oetztal.shp'))
cfg.PATHS['dem_file'] = get_demo_file('hef_srtm.tif')
pcp_fac = 2.6
cfg.PARAMS['prcp_scaling_factor'] = pcp_fac

base_dir = os.path.join(os.path.expanduser('~'), 'Climate')
entity = gpd.read_file(get_demo_file('HEF_MajDivide.shp')).iloc[0]
gdir = oggm.GlacierDirectory(entity, base_dir=base_dir)

tasks.define_glacier_region(gdir, entity=entity)
tasks.glacier_masks(gdir)
tasks.compute_centerlines(gdir)
```

```
tasks.initialize_flowlines(gdir)
tasks.compute_downstream_line(gdir)
tasks.catchment_area(gdir)
tasks.catchment_width_geom(gdir)
tasks.catchment_width_correction(gdir)
cfg.PATHS['climate_file'] = get_demo_file('histalp_merged_hef.nc')
tasks.process_custom_climate_data(gdir)
tasks.mu_candidates(gdir)

mbdf = gdir.get_ref_mb_data()
res = t_star_from_refmb(gdir, mbdf.ANNUAL_BALANCE)
local_mustar(gdir, tstar=res['t_star'][-1], bias=res['bias'][-1],
             prcp_fac=res['prcp_fac'], reset=True)
apparent_mb(gdir, reset=True)

# For flux plot
tasks.prepare_for_inversion(gdir, add_debug_var=True)

# For plots
mu_yr_clim = gdir.read_pickle('mu_candidates')[pcp_fac]
years, temp_yr, prcp_yr = mb_yearly_climate_on_glacier(gdir, pcp_fac)

# which years to look at
selind = np.searchsorted(years, mbdf.index)
temp_yr = np.mean(temp_yr[selind])
prcp_yr = np.mean(prcp_yr[selind])

# Average oberved mass-balance
ref_mb = mbdf.ANNUAL_BALANCE.mean()
mb_per_mu = prcp_yr - mu_yr_clim * temp_yr

# Diff to reference
diff = mb_per_mu - ref_mb
pdf = pd.DataFrame()
pdf[r'$\mu (t)$'] = mu_yr_clim
pdf['bias'] = diff
res = t_star_from_refmb(gdir, mbdf.ANNUAL_BALANCE)

# For the mass flux
cl = gdir.read_pickle('inversion_input')[-1]
mbmod = ConstantMassBalance(gdir)
mbx = mbmod.get_annual_mb(cl['hgt']) * cfg.SEC_IN_YEAR * cfg.RHO
fdf = pd.DataFrame(index=np.arange(len(mbx))*cl['dx'])
fdf['Flux'] = cl['flux']
fdf['Mass balance'] = mbx

# For the distributed thickness
tasks.volume_inversion(gdir, glen_a=cfg.A*3, fs=0)
tasks.distribute_thickness(gdir, how='per_interpolation')


# plot functions
def example_plot_temp_ts():
    d = xr.open_dataset(gdir.get_filepath('climate_monthly'))
    temp = d.temp.resample(freq='12MS', dim='time', how=np.mean).to_series()
    del temp.index.name
    ax = temp.plot(figsize=(8, 4), label='Annual temp')
    temp.rolling(31, center=True, min_periods=15).mean().plot(label='31-yr avg')
```

```
    plt.legend(loc='best')
    plt.title('HISTALP annual temperature, Hintereisferner')
    plt.ylabel(r'degC')
    plt.tight_layout()
    plt.show()


def example_plot_mu_ts():
    ax = mu_yr_clim.plot(figsize=(8, 4), label=r'$\mu (t)$');
    plt.legend(loc='best'); plt.title(r'$\mu$ candidates Hintereisferner');
    plt.ylabel(r'$\mu$ (mm yr$^{-1}$ K$^{-1}$)')
    plt.tight_layout()
    plt.show()


def example_plot_bias_ts():
    ax = pdf.plot(figsize=(8, 4), secondary_y='bias')
    plt.hlines(0, 1800, 2015, linestyles='-')
    ax.set_ylabel(r'$\mu$ (mm yr$^{-1}$ K$^{-1}$)');
    ax.set_title(r'$\mu$ candidates HEF');
    plt.ylabel(r'bias (mm yr$^{-1}$)')
    yl = plt.gca().get_ylim()
    for ts in res['t_star']:
        plt.plot((ts, ts), (yl[0], 0), linestyle=':', color='grey')
    plt.ylim(yl)
    plt.tight_layout()
    plt.show()


def example_plot_massflux():
    fig, ax = plt.subplots(figsize=(8, 4))
    fdf.plot(ax=ax, secondary_y='Mass balance', style=['C1-', 'C0-'])
    plt.axhline(0., color='grey', linestyle=':')
    ax.set_ylabel('Flux [m$^3$ s$^{-1}$]')
    ax.right_ax.set_ylabel('MB [kg m$^{-2}$ yr$^{-1}$]')
    ax.set_xlabel('Distance along flowline (m)')
    plt.title('Mass flux and mass balance along flowline')
    plt.tight_layout()
    plt.show()
```

1.1.4 Ice dynamics

The glaciers in OGGM are represented by a depth integrated flowline model. The equations for the isothermal shallow ice are solved along the glacier centerline, computed to represent best the flow of ice along the glacier (see for example antarcticglaciers.org for a general introduction about the various type of glacier models).

Here we present the basic physics and numerics of the two models implemented currently in OGGM, the `FluxBasedModel` (homegrown model with a rather simple numerical solver) and the `MUSCLSuperBeeModel` (mass-conserving numerical scheme, see *[Jarosch_etal_2013]*).

Ice flow

Let S be the area of a cross-section perpendicular to the flowline. It has a width w and a thickness h and, in this example, a parabolic bed shape.

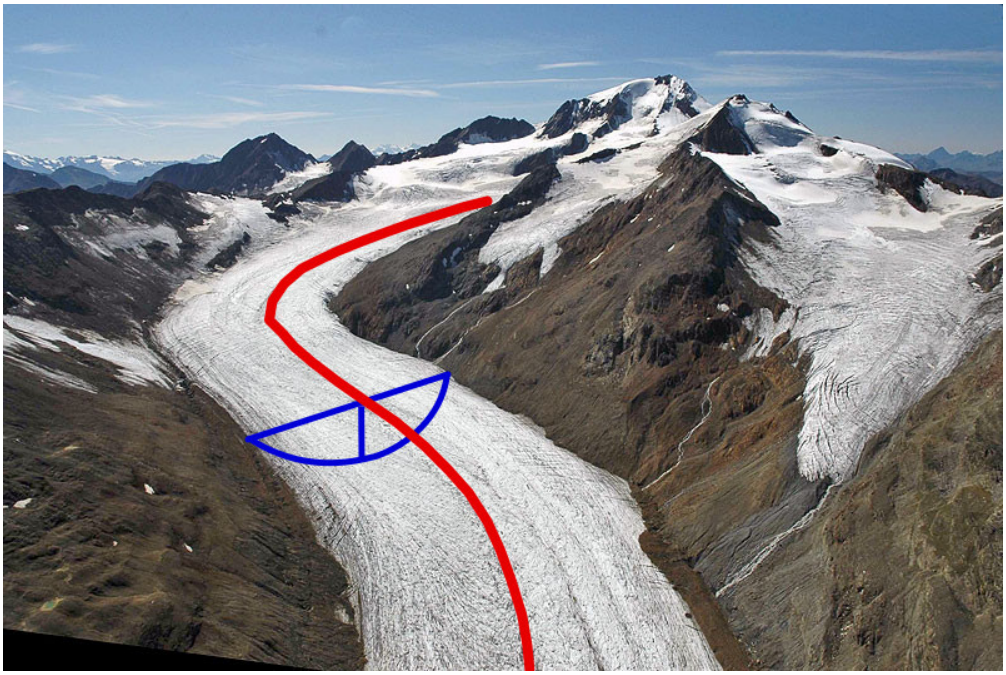


Fig. 1.2: Example of a cross-section along the glacier flowline. Background image from http://www.swisseduc.ch/glaciers/alps/hintereisferner/index-de.html

Volume conservation for this discrete element implies:

$$\frac{\partial S}{\partial t} = w\,\dot{m} - \nabla \cdot q$$

where $\dot{m}$ is the mass-balance, $q = uS$ the flux of ice, and u the depth-integrated ice velocity (*[Cuffey_Paterson_2010]*, p 310). This velocity can be computed from Glen's flow law as a function of the basal shear stress τ:

$$u = u_d + u_s = f_d h \tau^n + f_s \frac{\tau^n}{h}$$

The second term is to account for basal sliding, see e.g. *[Oerlemans_1997]* or *[Golledge_Levy_2011]*. It introduces an additional free parameter f_s and will therefore be ignored in a first approach. The deformation parameter f_d is better constrained and relates to Glen's temperaturedependent creep parameter A:

$$f_d = \frac{2A}{n+2}$$

The basal shear stress τ depends e.g. on the geometry of the bed *[Cuffey_Paterson_2010]*. Currently it is assumed to be equal to the driving stress τ_d:

$$\tau_d = \alpha \rho g h$$

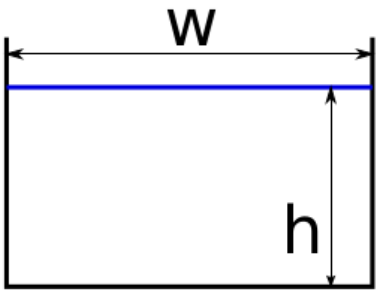
where α is the slope of the flowline and ρ the density of ice. Both the `FluxBasedModel` and the `MUSCLSuperBeeModel` solve for these equations, but with different numerical schemes.

Bed shapes

OGGM implements a number of possible bed-shapes. Currently the shape has no direct influence on the shear stress (i.e. Cuffey and Paterson's "shape factor" is not considered), but the shape will still have a considerable influence on glacier dynamics:

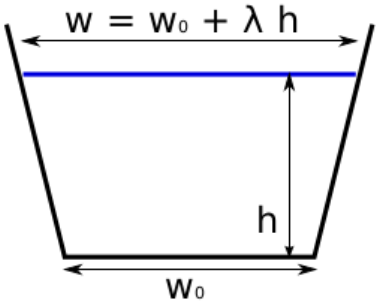
- the width change as a result of mass transport will be different for each shape, thus influencing the mass balance $w\,\dot{m}$
- with all other things held constant, a change in section area $\partial S/\partial t$ due to mass convergence/divergence will result in a different $\partial h/\partial t$ and thus in different shear stress computation at the next time step.

Rectangular



The simplest shape. The glacier width does not change with ice thickness.

Trapezoidal

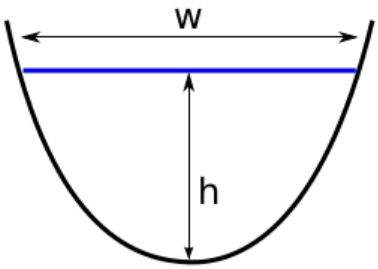


Trapezoidal shape with two degrees of freedom. The width change with thickness depends on λ. *[Golledge_Levy_2011]* uses $\lambda = 1$ (a 45° wall angle).

Parabolic

Parabolic shape with one degree of freedom, which makes it particularly useful for the bed inversion: if S and w are known:

$$h = \frac{3}{2}\frac{S}{w}$$



The parabola is defined by the bed-shape parameter $P_s = 4h/w^2$[1]. Very small values of this parameter imply very *flat* shapes, unrealistically sensitive to changes in h. For this reason, the default in OGGM is to use the mixed flowline model.

Mixed

A combination of rectangular, trapezoidal and parabolic shapes. The default is parabolic, but can be adapted in two cases:

- if the glacier section touches an ice-divide or a neighboring tributary catchment outline, the bed is considered to be rectangular;
- if the parabolic shape parameter P_s is below a certain threshold, a trapezoidal shape is used. Indeed, flat parabolas tend to be very sensitive to small changes in h, which is undesired.

Numerics

"Flux based" model

Most flowline models treat the volume conservation equation as a diffusion problem, taking advantage of the robust numerical solutions available for this type of equations. The problem with this approach is that it implies developing the $\partial S/\partial t$ term to solve for ice thickness h directly, thus implying different diffusion equations for different bed geometries (e.g. *[Oerlemans_1997]* with a trapezoidal bed).

The OGGM "flux based" model solves for the $\nabla \cdot q$ term (hence the name). The strong advantage of this method is that the numerical equations are the same for *any* bed shape, considerably simplifying the implementation. Similar to the "diffusion approach", the model looses mass-conservation in very steep slopes (*[Jarosch_etal_2013]*).

The numerical scheme implemented in OGGM is tested against A. Jarosch's MUSCLSuperBee Model (see below) and J. Oerleman's diffusion model for various idealized cases. For all cases but the steep slope one the model performs very well.

In order to increase the stability and speed of the computations, we solve the numerical equations on a forward staggered grid and we use an adaptive time stepping scheme.

MUSCLSuperBeeModel

A shallow ice model with improved numerics ensuring mass-conservation in very steep walls *[Jarosch_etal_2013]*. The model is currently in development to account for various bed shapes and tributaries and will likely become the default in OGGM.

[1] the local thickness y of the parabolic bed can be described by $y = hP_s x^2$. At $x = w/2$, $y = 0$ and therefore $P_s = 4h/w^2$.

Glacier tributaries

Glaciers in OGGM have a main centerline and, sometimes, one or more tributaries (which can themselves also have tributaries, see *Glacier flowlines*). The number of these tributaries depends on many factors, but most of the time the algorithm works well.

The main flowline and its tributaries are all modelled individually. At the end of a time step, the tributaries will transport mass to the branch they are flowing to. Numerically, this mass transport is handled by adding an element at the end of the flowline with the same properties (with, thickness...) as the last grid point, with the difference that the slope α is computed with respect to the altitude of the point they are flowing to. The ice flux is then computed as usual and transferred to the downlying branch.

The computation of the ice flux is always done first from the lowest order branches (without tributaries) to the highest ones, ensuring a correct mass-redistribution. The use of the slope between the tributary and main branch ensures that the former is not dynamical coupled to the latter. If the angle is positive or if no ice is present at the end of the tributary, no mass exchange occurs.

References

1.1.5 Bed inversion

To compute the initial ice thickness h_0, OGGM follows a methodology largely inspired from *[Farinotti_etal_2009]*, but fully automatised and relying on different methods for the mass-balance and the calibration.

Basics

The principle is simple. Let's assume for now that we know the flux of ice q flowing through a section of our glacier. The flowline physics and geometrical assumptions can be used to solve for the ice thickness h:

$$q = uS = \left(f_d h \tau^n + f_s \frac{\tau^n}{h} \right) S$$

With $n = 3$ and $S = hw$ (in the case of a rectangular section) or $S = 2/3hw$ (parabolic section), the equation reduces to solving a polynomial of degree 5 with one unique solution in $\mathbb{R}_+$. If we neglect sliding (the default in OGGM and in *[Farinotti_etal_2009]*), the solution is even simpler.

Ice flux

If we consider a point on the flowline and the catchment area Ω upstream of this point we have:

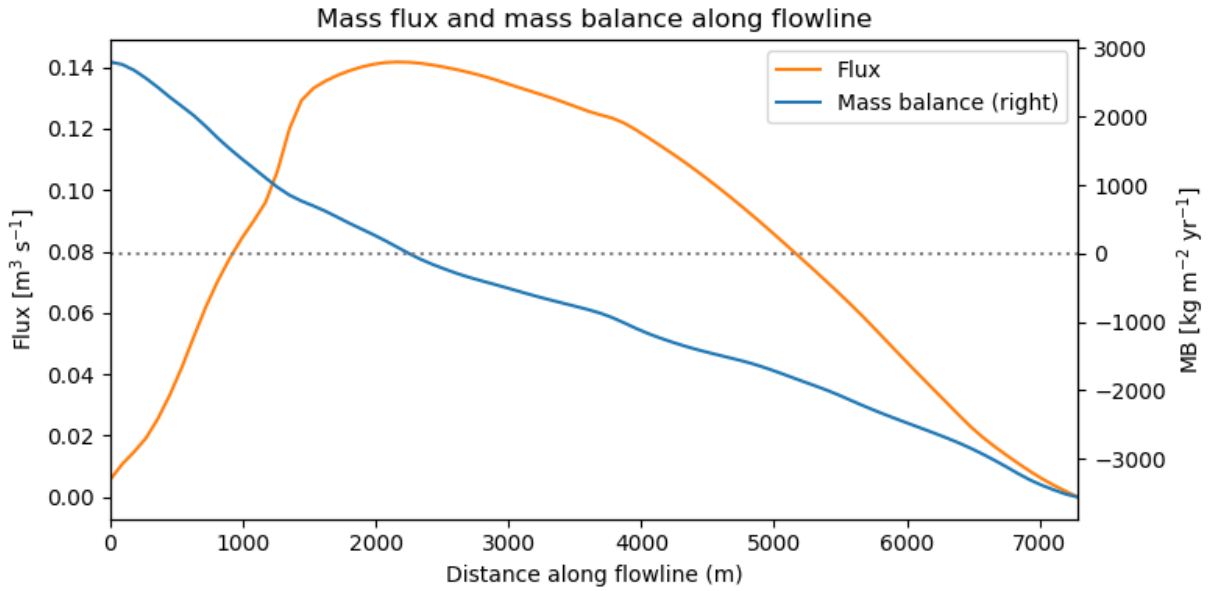
$$q = \int_\Omega (\dot{m} - \rho \frac{\partial h}{\partial t}) \, dA = \int_\Omega \widetilde{m} \, dA$$

with $\dot{m}$ the mass balance, and $\widetilde{m} = \dot{m} - \rho \partial h / \partial t$ the "apparent mass-balance" after *[Farinotti_etal_2009]*. If the glacier is in steady state, the apparent mass-balance is equivalent to the the actual (and observable) mass-balance. Unfortunately, $\partial h / \partial t$ is not known and there is no easy way to compute it. In order to continue, we have to make the assumption that our geometry is in equilibrium.

This however has a very useful consequence: indeed, for the calibration of our *Mass-balance* model it is required to find a date t^* at which the glacier would be in equilibrium with its average climate *while conserving its modern geometry*. Thus, we have $\widetilde{m} = \dot{m}_{t^*}$, where $\dot{m}_{t^*}$ is the 31-yr average mass-balance centered at t^* (which is known since the mass-balance model calibration).

The plot below shows the mass flux along the major flowline of Hintereisferner glacier. By construction, the flux is maximal at the equilibrium line and zero at the glacier tongue.

```
In [1]: example_plot_massflux()
```



Calibration

A number of climate and glacier related parameters are fixed prior to the inversion, leaving only one free parameter for the calibration of the bed inversion procedure: the inversion factor f_{inv}. It is defined such as:

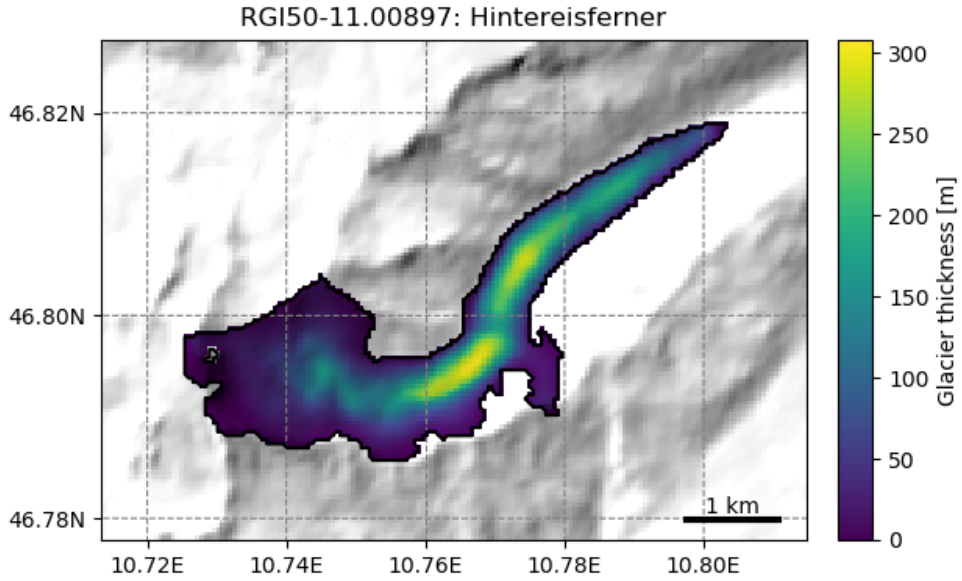
$$A = f_{inv}\, A_0$$

With A_0 the standard creep parameter (2.4e-24). Currently, f_{inv} is calibrated to minimize the volume RMSD of all glaciers with a volume estimation in the GlaThiDa database. It is therefore neither glacier nor temperature dependent and does not account for uncertainties in GlaThiDa's glacier-wide thickness estimations, two approximations which should be better handled in the future.

Distributed ice thickness

To obtain a 2D map of the glacier bed, the flowline thicknesses need to be interpolated to the glacier mask. The current implementation of this step in OGGM is currently very simple, but provides nice looking maps:

```
In [2]: tasks.catchment_area(gdir)

In [3]: graphics.plot_distributed_thickness(gdir)
```



References

1.2 Using OGGM

- *Installing OGGM*
- *Getting started*
- *Glacier working directories*
- *Input data*
- *Parallel computations*
- *Set-up an OGGM run*
- *Developer documentation*
- *Version history*

1.2.1 Installing OGGM

OGGM itself is a pure python package, but it has several dependencies wich are not trivial to install. The instructions below are self-explanatory and should work on any platform.

OGGM is fully tested with python version 3.6 on linux and partially tested on windows (windows should be used for development purposes only). OGGM doesn't work with python version 2.7.

Note: Complete beginners should get familiar with python and its packaging ecosystem before trying to install and run OGGM.

For most users we recommend to install python and the package dependencies withs the conda package manager: *Install with conda (all platforms)*. Linux users and people with experience with pip can follow the specific instructions *Install with virtualenv (linux/debian)*.

Dependencies

Standard SciPy track:

- numpy
- scipy
- scikit-image
- pillow
- matplotlib
- pandas
- xarray
- joblib

Configuration file parsing tool:

- configobj

I/O:

- netcdf4

GIS tools:

- gdal
- shapely
- pyproj
- rasterio
- geopandas

Testing:

- pytest

Other libraries:

- boto3
- salem
- motionless

Optional:

- progressbar2 (displays the download progress)

Install with conda (all platforms)

This is the recommended way to install OGGM.

Prerequisites

You should have a recent version of git and of the conda package manager. You can get conda by installing miniconda (the package manager alone - recommended) or anaconda (the full suite - with many packages you wont need).

Linux users should install a couple of linux packages (not all of them are required but it's good to have them anyway):

```
$ sudo apt-get install build-essential liblapack-dev gfortran libproj-dev git gdal-
↪bin libgdal-dev netcdf-bin ncview python-netcdf4 ttf-bitstream-vera
```

Conda environment

We recommend to create a specific environment for OGGM. In a terminal window, type:

```
conda create --name oggm_env python=3.5
```

You can of course use any other name for your environment.

Don't forget to activate it before going on:

```
source activate oggm_env
```

(on windows: `activate oggm_env`)

Packages

Install the packages from the conda-forge and oggm channels:

```
conda install -c oggm -c conda-forge oggm-deps
```

The oggm-deps package is a "meta package". It does not contain any code but will insall all the packages oggm needs automatically.

Warning: The conda-forge channel ensures that the complex package dependencies are handled correctly. Subsequent installations or upgrades from the default conda channel might brake the chain (see an example here). We strongly recommend to **always** use the the conda-forge channel for your installation.

You might consider setting conda-forge (and oggm) per default, as suggested on their documentation page:

```
conda config --add channels conda-forge
conda config --add channels oggm
conda install <package-name>
```

No scientific python installation is complete without installing ipython and jupyter:

```
conda install -c conda-forge ipython jupyter
```

OGGM

If you are using conda, you can install OGGM as a normal conda package:

```
conda install -c oggm -c conda-forge oggm
```

If you are using pip, you can install OGGM from PyPI:

```
pip install oggm
```

In this case you will be able to use the model but you cannot change its code. If you want to explore the code or participate to its development, we recommend to clone the git repository (or your own fork , see also *Contributing to OGGM*):

```
git clone https://github.com/OGGM/oggm.git
```

Then go to the project root directory:

```
cd oggm
```

And install OGGM in development mode (this is valid for **pip** or **conda** environments):

```
pip install -e .
```

Note: Installing OGGM in development mode means that subsequent changes to this code repository will be taken into account the next time you will `import oggm`. You can also update OGGM with a simple git pull from the root of the cloned repository.

Testing

You are almost there! The last step is to check if everything works as expected. From the oggm directory, type:

```
pytest .
```

The tests can run for a couple of minutes. If everything worked fine, you should see something like:

```
============================= test session starts ==============================
platform linux -- Python 3.5.2, pytest-3.3.1, py-1.5.2, pluggy-0.6.0
Matplotlib: 2.1.1
Freetype: 2.6.1
rootdir:
plugins: mpl-0.9
collected 164 items

oggm/tests/test_benchmarks.py ...                                        [  1%]
oggm/tests/test_graphics.py ...................                          [ 13%]
oggm/tests/test_models.py ................sss.ss.....sssssss              [ 34%]
oggm/tests/test_numerics.py .sssssssssssssss                             [ 44%]
oggm/tests/test_prepro.py .......s.......................s..s.......      [ 70%]
oggm/tests/test_utils.py ......................sss.s.sss.sssss..ss.       [ 95%]
oggm/tests/test_workflow.py sssssss                                      [100%]

=================== 112 passed, 52 skipped in 187.35 seconds ===================
```

You can safely ignore deprecation warnings and other DLL messages as long as the tests end without errors.

Congrats, you are now set-up for the *Getting started* section!

Install with virtualenv (linux/debian)

Note: The installation with pip requires a few more steps than with conda. Unless you have a good reason to be here, *Install with conda (all platforms)* is probably what you want do do.

The instructions below are for Debian / Ubuntu / Mint systems only!

Linux packages

For building stuffs:

```
$ sudo apt-get install build-essential python-pip liblapack-dev gfortran libproj-dev␣
↪python-setuptools
```

For matplolib:

```
$ sudo apt-get install tk-dev python3-tk python3-dev
```

For GDAL:

```
$ sudo apt-get install gdal-bin libgdal-dev python-gdal
```

For NETCDF:

```
$ sudo apt-get install netcdf-bin ncview python-netcdf4
```

Virtual environment

Install:

```
$ sudo pip install virtualenvwrapper
```

Create the directory where the virtual environments will be created:

```
$ mkdir ~/.pyvirtualenvs
```

Add these three lines to the files: ~/.profile and ~/.bashrc:

```
# Virtual environment options
export WORKON_HOME=$HOME/.pyvirtualenvs
source /usr/local/bin/virtualenvwrapper_lazy.sh
```

Reset your profile:

```
$ . ~/.profile
```

Make a new environment with **python 3**:

```
$ mkvirtualenv oggm_env -p /usr/bin/python3
```

(Details: http://simononsoftware.com/virtualenv-tutorial-part-2/)

Python Packages

Be sure to be on the working environment:

```
$ workon oggm_env
```

Update pip (important!):

```
$ pip install --upgrade pip
```

Install one by one the easy stuff:

```
$ pip install numpy scipy pandas shapely matplotlib
```

For **GDAL**, it's not as straight forward. First, check which version of GDAL is installed:

```
$ dpkg -s libgdal-dev  | grep '^Version:'
```

The major and minor package version (e.g. `1.10`, `1.11`, ...) should match that of the python package you want to install. For example, if the linux GDAL version is `1.11.3`, install the latest corresponding python version (in this case, `1.11.2`):

```
$ pip install gdal==1.11.2 --install-option="build_ext" --install-option="--include-
↪dirs=/usr/include/gdal"
```

Fiona also builds upon GDAL, so let's compile it the same way:

```
$ pip install fiona --install-option="build_ext" --install-option="--include-dirs=/
↪usr/include/gdal"
```

(Details: http://tylerickson.blogspot.co.at/2011/09/installing-gdal-in-python-virtual.html)

Install further stuffs:

```
$ pip install pyproj rasterio Pillow geopandas netcdf4 scikit-image configobj joblib␣
↪xarray boto3 progressbar2 pytest motionless
```

And the salem library:

```
$ pip install git+https://github.com/fmaussion/salem.git
```

OGGM and tests

Refer to *OGGM* above.

1.2.2 Getting started

The ultimate goal of OGGM will be to hide the python workflow behind the model entirely, and run it only using configuration files and scripts. We are not there yet, and if you want to use and participate to the development of

OGGM you'll have to get your hands dirty. We hope however that the workflow is structured enough so that it is possible to jump in without having to understand all of its internals.

The few examples below are meant to illustrate the general design of OGGM, without going into the details of the implementation.

Imports

The following imports are necessary for all of the examples:

```
In [1]: import geopandas as gpd

In [2]: import oggm

In [3]: from oggm import cfg, tasks, graphics

In [4]: from oggm.utils import get_demo_file
```

Initialisation and GlacierDirectories

The first thing to do when running OGGM is to initialise it. This function will read the default configuration file which contains all user defined parameters:

```
In [5]: cfg.initialize()
```

These parameters are now accessible to all OGGM routines. For example, the `cfg.PARAMS` dict contains some runtime parameters, while `cfg.PATHS` stores the paths to the input files and the working directory (where the model output will be written):

```
In [6]: cfg.PARAMS['topo_interp']
Out[6]: 'cubic'

In [7]: cfg.PARAMS['temp_default_gradient']
\\\\\\\\\\\\\\\Out[7]: -0.0065

In [8]: cfg.PATHS
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[8]:
PathOrderedDict([('dl_cache_dir', '/home/docs/OGGM/download_cache'),
                 ('tmp_dir', '/home/docs/OGGM/tmp'),
                 ('cru_dir', '/home/docs/OGGM/cru'),
                 ('rgi_dir', '/home/docs/OGGM/rgi'),
                 ('test_dir', '/home/docs/OGGM/tests'),
                 ('working_dir', ''),
                 ('dem_file', ''),
                 ('climate_file', '')])
```

We'll use some demo files (shipped with OGGM) for the basic input:

```
In [9]: cfg.PATHS['working_dir'] = os.path.expanduser('~/doc_wd')  # working directory

In [10]: cfg.PATHS['dem_file'] = get_demo_file('hef_srtm.tif')  # topography

In [11]: cfg.set_intersects_db(get_demo_file('rgi_intersect_oetztal.shp'))  #␣
↪intersects
```

The starting point of a run is always a valid RGI file. In this case we use a very small subset of the RGI, the outlines of the Hinereisferner (HEF) in the Austrian Alps:

```
In [12]: entity = gpd.GeoDataFrame.from_file(get_demo_file('HEF_MajDivide.shp')).
↪iloc[0]

In [13]: entity
Out[13]:
Area                                                    6.24736
Aspect                                                       71
BgnDate                                                20030799
CenLat                                                  46.8003
CenLon                                                  10.7584
EndDate                                                20030999
GLIMSId                                          G010758E46800N
GlacType                                                  0091
Lmax                                                      7178
Name                                            Hintereisferner
O1Region                                                    11
O2Region                                                     1
RGIFlag                                                   0909
RGIId                                           RGI50-11.00897
Slope                                                       16
Zmax                                                      3674
Zmed                                                      3050
Zmin                                                      2430
geometry    POLYGON ((10.74505084001018 46.80376064580748,...
Name: 0, dtype: object
```

This information is enough to define HEF's *GlacierDirectory*:

```
In [14]: gdir = oggm.GlacierDirectory(entity)
```

Glacier working directories have two major purposes:

- carry information about the glacier attributes
- handle I/O and filepaths operations

For example, it will tell OGGM where to write the data for this glacier or its terminus type:

```
In [15]: gdir.dir
Out[15]: '/home/docs/doc_wd/per_glacier/RGI50-11/RGI50-11.00/RGI50-11.00897'

In [16]: gdir.terminus_type
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[16]:
↪'Land-terminating'
```

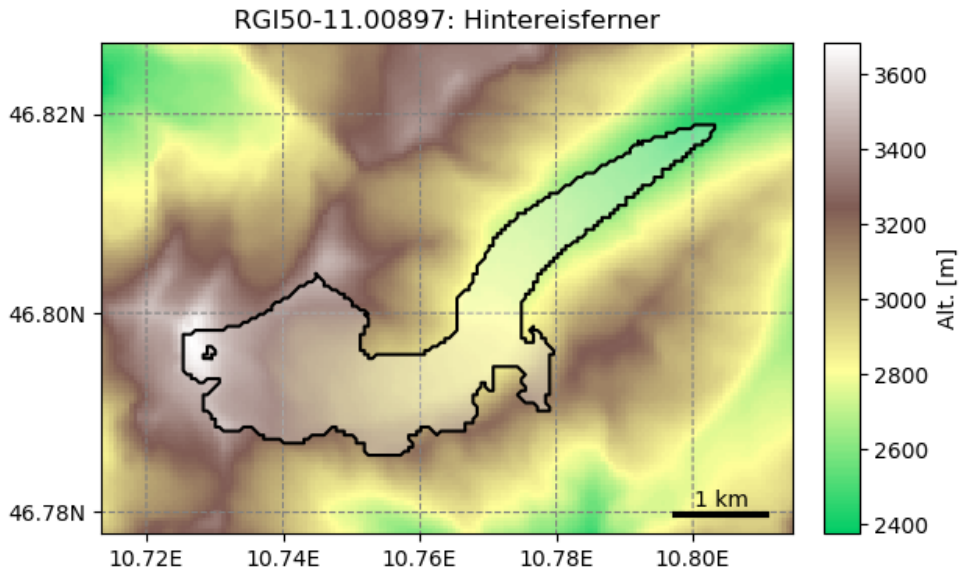
GlacierDirectories are the input to most OGGM functions. In fact, they are the only required input to all *Entity tasks*. These entity tasks are processes which can run on one glacier at a time (the vast majority of OGGM tasks are entity tasks). The first task to apply to an empty GlacierDirectory is *tasks.define_glacier_region()*, which sets the local glacier map and topography, and *tasks.glacier_masks()*, which prepares gridded topography data:

```
In [17]: tasks.define_glacier_region(gdir, entity=entity)

In [18]: tasks.glacier_masks(gdir)

In [19]: os.listdir(gdir.dir)
Out[19]:
```

```
['intersects.cpg',
 'outlines.dbf',
 'geometries.pkl',
 'outlines.cpg',
 'intersects.shx',
 'dem_source.pkl',
 'intersects.shp',
 'gridded_data.nc',
 'outlines.shx',
 'intersects.prj',
 'outlines.shp',
 'log.txt',
 'dem.tif',
 'glacier_grid.json',
 'intersects.dbf',
 'outlines.prj']
```

The directory is now filled with data. Other tasks can build upon these, for example the plotting functions:

```
In [20]: graphics.plot_domain(gdir)
```



What next?

This documentation is growing step by step. In the meantime, a good place to start is the `oggm/docs/notebooks` directory.

You will find two notebooks:

- `getting_started.ipynb`, which set-ups an entire OGGM run in the Ötztal region.
- `flowline_model.ipynb`, which describes the usage of the flowline model for idealized test cases.

1.2.3 Glacier working directories

See also: *GlacierDirectory*

The majority of OGGM tasks are so-called "entity tasks". They are standalone operations to be realized on one single glacier entity. These tasks are executed sequentially: they often need input generated by the previous task(s). In order to avoid complicated chains of arguments, each task will read the input data from a glacier-specific directory and writes its output into the same directory, making the new data available for further computations.

Initalising a glacier directory

If no directory has been created yet, a GlacierDirectory requires an RGI entity as input:

```
In [1]: base_dir = os.path.join(os.path.expanduser('~'), 'OGGM_docs', 'GlacierDir')

In [2]: entity = gpd.GeoDataFrame.from_file(get_demo_file('HEF_MajDivide.shp')).
↪iloc[0]

In [3]: gdir = oggm.GlacierDirectory(entity, base_dir=base_dir)

In [4]: gdir.dir
Out[4]: '/home/docs/OGGM_docs/GlacierDir/RGI50-11/RGI50-11.00/RGI50-11.00897'

In [5]: gdir.rgi_id, gdir.rgi_area_km2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[5]:␣
↪('RGI50-11.00897', 6.247362925353041)
```

Note that this directory has just been created and is empty. The *tasks.define_glacier_region()* will fill it with the first data files:

```
In [6]: tasks.define_glacier_region(gdir, entity=entity)

In [7]: os.listdir(gdir.dir)
Out[7]:
['intersects.cpg',
 'outlines.dbf',
 'outlines.cpg',
 'intersects.shx',
 'dem_source.pkl',
 'intersects.shp',
 'outlines.shx',
 'intersects.prj',
 'outlines.shp',
 'log.txt',
 'dem.tif',
 'glacier_grid.json',
 'intersects.dbf',
 'outlines.prj']
```

This persistence on disk allows for example to continue a workflow that has been previously interrupted. Initialising a GlacierDirectory from a non-empty folder won't erase its content (you'll have to set `reset=True` explicitly if you want that):

```
In [8]: gdir = oggm.GlacierDirectory(entity, base_dir=base_dir)

In [9]: os.listdir(gdir.dir)  # the directory still contains the data
Out[9]:
```

```
['intersects.cpg',
 'outlines.dbf',
 'outlines.cpg',
 'intersects.shx',
 'dem_source.pkl',
 'intersects.shp',
 'outlines.shx',
 'intersects.prj',
 'outlines.shp',
 'log.txt',
 'dem.tif',
 'glacier_grid.json',
 'intersects.dbf',
 'outlines.prj']
```

You can also initialise a non-empty GlacierDirectory with its RGI ID, thus sparing the reading of the shapefile every time:

```
In [10]: gdir = oggm.GlacierDirectory('RGI50-11.00897', base_dir=base_dir)
```

cfg.BASENAMES

This is a list of the files that can be found in the glacier directory or its divides. These data files and their names are standardized and listed in the `oggm.cfg` module. If you want to implement your own task you'll have to add an entry to this file too.

calving_output.pkl Calving output

catchment_indices.pkl A list of len n_centerlines, each element conaining a numpy array of the indices in the glacier grid which represent the centerline's catchment area.

catchments_intersects.shp The catchments intersections in the local projection.

centerlines.pkl A list of *Centerline* instances, sorted by flow order.

cesm_data.nc The monthly GCM climate timeseries for this glacier, stored in a netCDF file.

climate_info.pkl Some information (dictionary) about the climate data for this glacier, avoiding many useless accesses to the netCDF file.

climate_monthly.nc The monthly climate timeseries for this glacier, stored in a netCDF file.

dem.tif A geotiff file containing the DEM (reprojected into the local grid).

dem_source.pkl A string with the source of the topo file (ASTER, SRTM, ...).

downstream_line.pkl A `dict` containing the downsteam line geometry as well as the bedshape computed from a parabolic fit.

flowline_catchments.shp The flowline catchments in the local projection.

geometries.pkl A `dict` containing the shapely.Polygons of a glacier. The "polygon_hr" entry contains the geometry transformed to the local grid in (i, j) coordinates, while the "polygon_pix" entry contains the geometries transformed into the coarse grid (the i, j elements are integers). The "polygon_area" entry contains the area of the polygon as computed by Shapely.

glacier_grid.json A `salem.Grid` handling the georeferencing of the local grid.

gridded_data.nc A netcdf file containing several gridded data variables such as topography, the glacier masks and more (see the netCDF file metadata).

intersects.shp The glacier intersects in the local projection.

inversion_flowlines.pkl A "better" version of the Centerlines, now on a regular spacing i.e., not on the gridded (i, j) indices. The tails of the tributaries are cut out to make more realistic junctions. They are now "1.5D" i.e., with a width.

inversion_input.pkl List of dicts containing the data needed for the inversion.

inversion_output.pkl List of dicts containing the output data from the inversion.

inversion_params.pkl Dict of fs and fd as computed by the inversion optimisation.

linear_mb_params.pkl When using a linear mass-balance for the inversion, this dict stores the optimal ela_h and grad.

local_mustar.csv A csv with three values: the local scalars mu*, t*, bias

model_diagnostics.nc A netcdf file containing the model diagnostics (volume, mass-balance, length...).

model_flowlines.pkl List of flowlines ready to be run by the model.

model_run.nc A netcdf file containing enough information to reconstruct the entire flowline glacier along the run (can be data expensive).

mu_candidates.pkl A pandas.Series with the (year, mu) data.

outlines.shp The glacier outlines in the local projection.

prcp_fac_optim.pkl A Dataframe containing the bias scores as a function of the prcp factor. This is useful for testing mostly.

1.2.4 Input data

OGGM needs various data files to run. To date, we rely exclusively on open-access data that are all downloaded automatically for the user. This page explains the various ways OGGM uses to get the the data it needs.

Calibration data and testing: the `~/.oggm` directory

At the first import, OGGM will create a cached `.oggm` directory in your `$HOME` folder. This directory contains all data obtained from the oggm sample data repository. It contains various files needed only for testing, but also some important files needed for calibration and validation. For example:

- The CRU baseline climatology (CL v2.0, obtained from crudata.uea.ac.uk/ and prepared for OGGM),
- The reference mass-balance data from WGMS with links to the respective RGI polygons,
- The reference ice thickness data from WGMS (GlaThiDa database).

This directory should be updated automatically when new files are available on GitHub, but if you encounter any problems simply delete it, it will be re-downloaded automatically.

All other data: auto-downloads and the `~/.oggm_config` file

OGGM implements a bunch of logic to make access to the input data as painless as possible, including the automated download of all the required files.

Unlike runtime parameters (such as physical constants or working directories), the input data is shared accross runs and even accross computers. Therefore, the paths to previously downloaded data are stored in a simple configuration file that you'll find in your `$HOME` folder: the `~/.oggm_config` file.

The file should look like:

```
dl_cache_dir = /path/to/download_cache
dl_cache_readonly = False
tmp_dir = /path/to/tmp_dir
cru_dir = /path/to/cru_dir
rgi_dir = /path/to/rgi_dir
test_dir = /path/to/test_dir
has_internet = True
```

With:

- `dl_cache_dir` is a path to a directory where *all* the files you downloaded will be cached for later use. Most of the users won't need to explore this folder (it is organized as a list of urls) but you have to make sure to set this path to a folder with sufficient disk space available.
- `dl_cache_readonly` indicates if writing is allowed in this folder (this is the default). Setting this to `True` will prevent any further download in this directory (useful for cluster environments, where this data might be available on a readonly folder).
- `tmp_dir` is a path to OGGM's temporary directory. Most of the topography files used by OGGM are downloaded and cached in a compressed format. They will be extracted in `tmp_dir` before use. OGGM will never allow more than 100 `.tiff` files to exist in this directory by deleting the oldest ones following the rule of the Least Recently Used (LRU) item. Nevertheless, this directory might still grow to quite a large size. Simply delete it if you want to get this space back.
- `cru_dir` is the location where the CRU climate files are extracted. They are quite large! (approx. 6Gb)
- `rgi_dir` is the location where the RGI shapefiles are extracted.
- `test_dir` is the location where OGGM will write some of its output during tests. It can be set to `tmp_dir` if you want to, but it can also be another directory (for example a fast SSD disk – the data written for testing is much smaller than the input files).

Note: `tmp_dir`, `cru_dir` and `rgi_dir` can be overridden and set to a specific directory by defining an environment variable `OGGM_EXTRACT_DIR` to a directory path. Similarly the envorinment variables `OGGM_DOWNLOAD_CACHE` and `OGGM_DOWNLOAD_CACHE_RO` override the `dl_cache_dir` and `dl_cache_readonly` settings.

1.2.5 Parallel computations

OGGM is designed to use the available resources as well as possible. For single nodes machines but with more than one processor (frequent case for personal comuters) OGGM ships with a multiprocessing approach which is fairly simple to use. For cluster environments, use *MPI*.

Multiprocessing

Most OGGM computations are embarrassingly parallel: they are standalone operations to be realized on one single glacier entity and therefore independent from each other (they are called **entity tasks**, as opposed to the non-parallelizable **global tasks**).

When given a list of *Glacier working directories* on which to apply a given task, the *`workflow.execute_entity_task()`* will distribute the operations on the available processors using Python's multiprocessing module. You can control this behavior with the `use_multiprocessing` config parameter and the number of processors with `mp_processes`. The default in OGGM is:

```
In [1]: from oggm import cfg

In [2]: cfg.initialize()

In [3]: cfg.PARAMS['use_multiprocessing']  # whether to use multiprocessing
Out[3]: True

In [4]: cfg.PARAMS['mp_processes']  # number of processors to use
\\\\\\\\\\\\\Out[4]: -1
```

`-1` means that all available processors will be used.

MPI

OGGM can be run in a clustered environment, using standard mpi features. OGGM depends on mpi4py in that case, which can be installed via either conda:

```
conda install -c conda-forge mpi4py
```

or pip:

```
pip install mpi4py
```

mpi4py itself depends on a working mpi environment, which is usually supplied by the maintainers of your cluster. On conda, it comes with its own copy of mpich, which is nice and easy for quick testing, but likely undesireable for the performance of actual runs.

For an actual run, invoke any script using oggm via mpiexec, and pass the `--mpi` parameter to the script itself:

```
mpiexec -n 10 python ./run_rgi_region.py --mpi
```

Be aware that the first process with rank 0 is the manager process, that by itself does not do any calculations and is only used to distribute tasks. So the actual number of working processes is one lower than the number passed to mpiexec/your clusters scheduler.

1.2.6 Set-up an OGGM run

These examples will show you some example scripts to realise regional or global runs. The first example can be run on a laptop within a few minutes. The second example (mass balance calibration) requires more resources but with a reasonably powerful personal computer you should be OK.

For region-wide or global simulations, we recommend to use a cluster environment. OGGM should work well on cloud computing services like Amazon or Google Cloud, too.

Before you start, make sure you had a look at the *Input data* section.

1. Set-up a default run for a list of glaciers

This example shows how to run the OGGM model for a list of selected glaciers (here, four). For this example we download the list of glaciers from Fabien's dropbox, but you can use any list of glaciers for this. See the prepare_glacier_list.ipynb notebook in the `oggm/docs/notebooks` directory for an example on how to prepare such a file.

Note that the default in OGGM is to use a previously calibrated list of t^* for the run, which means that we don't have to calibrate the mass balance model ourselves (thankfully, otherwise you would have to add all the calibration glaciers to your list too).

Note that to be exact, this procedure can only be applied if the model parameters don't change between the calibration and the run. After testing, it appears that changing the 'border' parameter won't affect the results much (as expected), so it's ok to change this parameter. Some other parameters (e.g. topo smoothing, dx, precip factor, alternative climate data...) will probably need a re-calibration (see the OGGM calibration recipe for this).

Script

```
# Python imports
from os import path
import shutil
import zipfile
import oggm

# Module logger
import logging
log = logging.getLogger(__name__)

# Libs
import salem

# Locals
import oggm.cfg as cfg
from oggm import tasks, utils, workflow
from oggm.workflow import execute_entity_task

# For timing the run
import time
start = time.time()

# Initialize OGGM and set up the default run parameters
cfg.initialize()

# Local working directory (where OGGM will write its output)
WORKING_DIR = path.join(path.expanduser('~'), 'tmp', 'OGGM_precalibrated_run')
utils.mkdir(WORKING_DIR, reset=True)
cfg.PATHS['working_dir'] = WORKING_DIR

# Use multiprocessing?
cfg.PARAMS['use_multiprocessing'] = True

# Here we override some of the default parameters
# How many grid points around the glacier?
# Make it large if you expect your glaciers to grow large
cfg.PARAMS['border'] = 100

# Set to True for operational runs
cfg.PARAMS['continue_on_error'] = False

# We use intersects
# (this is slow, it could be replaced with a subset of the global file)
rgi_dir = utils.get_rgi_intersects_dir(version='5')
cfg.set_intersects_db(path.join(rgi_dir, '00_rgi50_AllRegs',
                                'intersects_rgi50_AllRegs.shp'))
```

```
# Pre-download other files which will be needed later
utils.get_cru_cl_file()
utils.get_cru_file(var='tmp')
utils.get_cru_file(var='pre')

# Download the RGI file for the run
# We us a set of four glaciers here but this could be an entire RGI region,
# or any glacier list you'd like to model
dl = 'https://www.dropbox.com/s/6cwi7b4q4zqgh4a/RGI_example_glaciers.zip?dl=1'
with zipfile.ZipFile(utils.file_downloader(dl)) as zf:
    zf.extractall(WORKING_DIR)
rgidf = salem.read_shapefile(path.join(WORKING_DIR, 'RGI_example_glaciers',
                                       'RGI_example_glaciers.shp'))

# Sort for more efficient parallel computing
rgidf = rgidf.sort_values('Area', ascending=False)

log.info('Starting OGGM run')
log.info('Number of glaciers: {}'.format(len(rgidf)))

# Go - initialize working directories
gdirs = workflow.init_glacier_regions(rgidf)

# Preprocessing tasks
task_list = [
    tasks.glacier_masks,
    tasks.compute_centerlines,
    tasks.initialize_flowlines,
    tasks.compute_downstream_line,
    tasks.compute_downstream_bedshape,
    tasks.catchment_area,
    tasks.catchment_intersections,
    tasks.catchment_width_geom,
    tasks.catchment_width_correction,
]
for task in task_list:
    execute_entity_task(task, gdirs)

# Climate tasks -- only data IO and tstar interpolation!
execute_entity_task(tasks.process_cru_data, gdirs)
tasks.distribute_t_stars(gdirs)
execute_entity_task(tasks.apparent_mb, gdirs)

# Inversion tasks
execute_entity_task(tasks.prepare_for_inversion, gdirs)
# We use the default parameters for this run
execute_entity_task(tasks.volume_inversion, gdirs, glen_a=cfg.A, fs=0)
execute_entity_task(tasks.filter_inversion_output, gdirs)

# Final preparation for the run
execute_entity_task(tasks.init_present_time_glacier, gdirs)

# Random climate representative for the tstar climate, without bias
# In an ideal world this would imply that the glaciers remain stable,
# but it doesn't have to be so
execute_entity_task(tasks.random_glacier_evolution, gdirs,
                    nyears=200, bias=0, seed=1,
```

```
                    filesuffix='_tstar')

# Compile output
log.info('Compiling output')
utils.glacier_characteristics(gdirs)
utils.compile_run_output(gdirs, filesuffix='_tstar')

# Log
m, s = divmod(time.time() - start, 60)
h, m = divmod(m, 60)
log.info('OGGM is done! Time needed: %d:%02d:%02d' % (h, m, s))
```

If everything went well, you should see an output similar to:

```
2017-10-21 00:07:17: oggm.cfg: Parameter file: /home/mowglie/Documents/git/oggm-fork/
↪oggm/params.cfg
2017-10-21 00:07:27: __main__: Starting OGGM run
2017-10-21 00:07:27: __main__: Number of glaciers: 4
2017-10-21 00:07:27: oggm.workflow: Multiprocessing: using all available processors␣
↪(N=4)
2017-10-21 00:07:27: oggm.core.gis: (RGI50-01.10299) define_glacier_region
2017-10-21 00:07:27: oggm.core.gis: (RGI50-18.02342) define_glacier_region
(...)
2017-10-21 00:09:30: oggm.core.flowline: (RGI50-01.10299) default time stepping was␣
↪successful!
2017-10-21 00:09:39: oggm.core.flowline: (RGI50-18.02342) default time stepping was␣
↪successful!
2017-10-21 00:09:39: __main__: Compiling output
2017-10-21 00:09:39: __main__: OGGM is done! Time needed: 0:02:22
```

Note: During the `random_glacier_evolution` task some numerical warnings might occur. These are expected to happen and are caught by the solver, which then tries a more conservative time stepping scheme.

Note: The `random_glacier_evolution` task can be replaced by any climate scenario built by the user. For this you'll have to develop your own task, which will be the topic of another example script.

Starting from a preprocessed state

Now that we've gone through all the preprocessing steps once and that their output is stored on disk, it isn't necessary to re-run everything to make a new experiment. The code can be simplified to:

```
# Python imports
from os import path
import oggm

# Module logger
import logging
log = logging.getLogger(__name__)

# Libs
import salem
```

```
# Locals
import oggm.cfg as cfg
from oggm import tasks, utils, workflow
from oggm.workflow import execute_entity_task

# For timing the run
import time
start = time.time()

# Initialize OGGM and set up the run parameters
cfg.initialize()

# Local working directory (where OGGM will write its output)
WORKING_DIR = path.join(path.expanduser('~'), 'tmp', 'OGGM_precalibrated_run')
cfg.PATHS['working_dir'] = WORKING_DIR

# Use multiprocessing?
cfg.PARAMS['use_multiprocessing'] = True

# Read RGI
rgidf = salem.read_shapefile(path.join(WORKING_DIR, 'RGI_example_glaciers',
                                       'RGI_example_glaciers.shp'))
# Sort for more efficient parallel computing
rgidf = rgidf.sort_values('Area', ascending=False)

log.info('Starting OGGM run')
log.info('Number of glaciers: {}'.format(len(rgidf)))

# Initialize from existing directories
gdirs = workflow.init_glacier_regions(rgidf)

# We can step directly to a new experiment!
# Random climate representative for the recent climate (1985-2015)
# This is a kinf of "commitment" run
execute_entity_task(tasks.random_glacier_evolution, gdirs,
                    nyears=200, y0=2000, seed=1,
                    filesuffix='_commitment')

# Compile output
log.info('Compiling output')
utils.compile_run_output(gdirs, filesuffix='_commitment')

# Log
m, s = divmod(time.time() - start, 60)
h, m = divmod(m, 60)
log.info('OGGM is done! Time needed: %d:%02d:%02d' % (h, m, s))
```

Note: Note the use of the `filesuffix` keyword argument. This allows to store the output of different runs in different files, useful for later analyses.

Some analyses

The output directory contains the compiled output files from the run. The `glacier_characteristics.csv` file contains various information about each glacier obation after the preprocessing, either from the RGI directly (location,

name, glacier type...) or from the model itself (hypsometry, inversion model output...).

Here is an example of how to read the file:

```
from os import path
import pandas as pd
WORKING_DIR = path.join(path.expanduser('~'), 'tmp', 'OGGM_precalibrated_run')
df = pd.read_csv(path.join(WORKING_DIR, 'glacier_characteristics.csv'))
print(df)
```

Output (reduced for clarity):

rgi_id	name	cen-lon	cen-lat	rgi_area_km2	terminus_type	dem_max_elev	dem_min_elev	dem_mean_elev	longest_centerline_km	inv_thickness_m	vas_thickness_m
RGI50-18.02342	Tasman Glacier	170.238	-43.5653	95.216	Land-terminating	3662	715	7	27.2195	186.38	187.713
RGI50-01.10299	Coxe Glacier	-148.037	61.14	419.282	Marine-terminating	1840	6	8	11.6243	137.153	103.136
RGI50-11.00897	Hintereisferner	10.7584	46.8008	8.036	Land-terminating	3684	2447	3	8.99975	107.642	74.2795
RGI50-08.02637	Storglaciaeren	18.5606	67.9043	3.163	Land-terminating	1909	1176	2	3.54495	63.2469	52.362

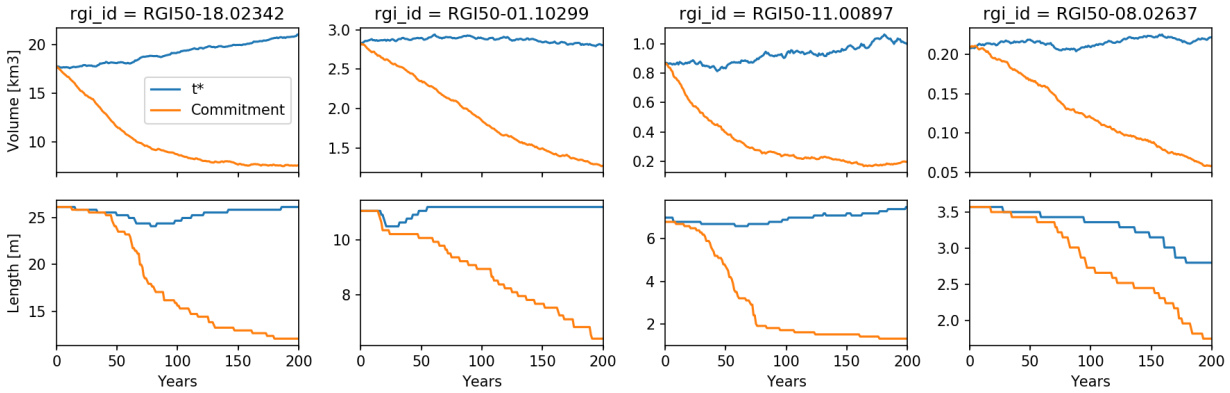
The run outpout is stored in netCDF files, and it can therefore be read with any tool able to read those (Matlab, R, ...).

I myself am familiar with python, and I use the xarray package to read the data:

```
import xarray as xr
import matplotlib.pyplot as plt
from os import path

WORKING_DIR = path.join(path.expanduser('~'), 'tmp', 'OGGM_precalibrated_run')
ds1 = xr.open_dataset(path.join(WORKING_DIR, 'run_output_tstar.nc'))
ds2 = xr.open_dataset(path.join(WORKING_DIR, 'run_output_commitment.nc'))

v1_km3 = ds1.volume * 1e-9
v2_km3 = ds2.volume * 1e-9
l1_km = ds1.length * 1e-3
l2_km = ds2.length * 1e-3

f, axs = plt.subplots(2, 4, figsize=(12, 4), sharex=True)

for i in range(4):
    ax = axs[0, i]
    v1_km3.isel(rgi_id=i).plot(ax=ax, label='t*')
    v2_km3.isel(rgi_id=i).plot(ax=ax, label='Commitment')
    if i == 0:
        ax.set_ylabel('Volume [km3]')
        ax.legend(loc='best')
    else:
        ax.set_ylabel('')
    ax.set_xlabel('')

    ax = axs[1, i]
```

```
    # Length can need a bit of postprocessing because of some cold years
    # Where seasonal snow is thought to be a glacier...
    for l in [l1_km, l2_km]:
        roll_yrs = 5
        sel = l.isel(rgi_id=i).to_series()
        # Take the minimum out of 5 years
        sel = sel.rolling(roll_yrs).min()
        sel.iloc[0:roll_yrs] = sel.iloc[roll_yrs]
        sel.plot(ax=ax)
    if i == 0:
        ax.set_ylabel('Length [m]')
    else:
        ax.set_ylabel('')
    ax.set_xlabel('Years')
    ax.set_title('')

plt.tight_layout()
plt.show()
```

This code snippet should produce the following plot:



Warning: In the script above we have to "smooth" our length data for nicer plots. This is necessary because of annual snow cover: indeed, OGGM cannot differentiate between snow and ice. At the end of a cold mass-balance year, it can happen that some snow remains at the tongue and below: for the model, this looks like a longer glacier... (this cover is very thin, so that it doesn't influence the volume much).

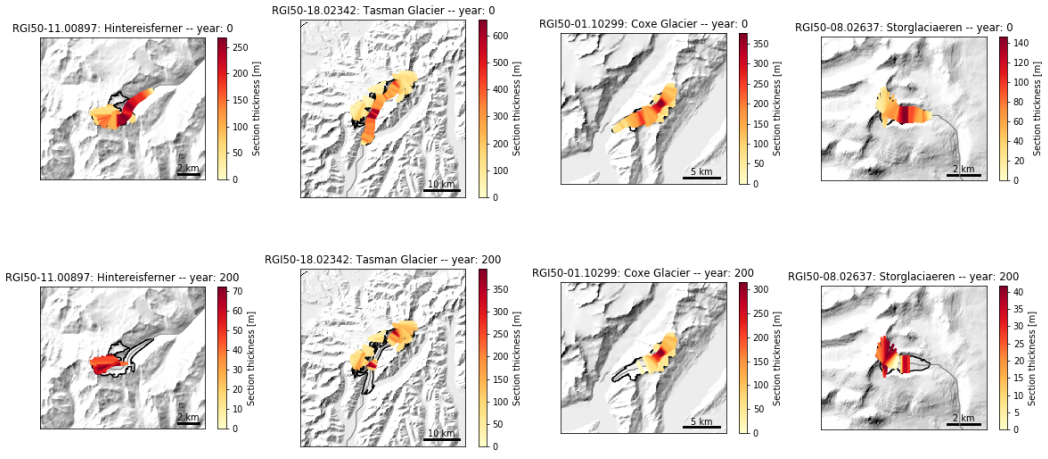
More analyses

Here is a more complex example to demonstrate how to plot the glacier geometries after the run:

```
# Python imports
import logging
from os import path

# External modules
import matplotlib.pyplot as plt

# Locals
import oggm.cfg as cfg
```

```
from oggm import workflow, graphics
from oggm.core import flowline

# Module logger
log = logging.getLogger(__name__)

# Initialize OGGM
cfg.initialize()

# Local working directory (where OGGM will write its output)
WORKING_DIR = path.join(path.expanduser('~'), 'tmp', 'OGGM_precalibrated_run')
cfg.PATHS['working_dir'] = WORKING_DIR

# Initialize from existing directories
# (note that we don't need the RGI file: this is going to be slow sometimes
# but it works)
gdirs = workflow.init_glacier_regions()

# Plot: we will show the state of all four glaciers at the beginning and at
# the end of the commitment simulation
f, axs = plt.subplots(2, 4, figsize=(20, 8))

for i in range(4):
    ax = axs[0, i]
    gdir = gdirs[i]
    # Use the model ouptut file to simulate the glacier evolution
    model = flowline.FileModel(gdir.get_filepath('model_run',
                                                 filesuffix='_commitment'))
    graphics.plot_modeloutput_map(gdirs[i], model=model, ax=ax,
                                  lonlat_contours_kwargs={'interval': 0})
    ax = axs[1, i]
    model.run_until(200)
    graphics.plot_modeloutput_map(gdirs[i], model=model, ax=ax,
                                  lonlat_contours_kwargs={'interval': 0})

plt.subplots_adjust(wspace=0.4, hspace=0.08, top=0.98, bottom=0.02)
plt.show()
```

Which should produce the following plot:



2. Run the mass-balance calibration

Sometimes you will need to do the mass-balance calibration yourself. For example if you use alternate climate data, or change the parameters of the model. Here we show how to run the calibration for all available reference glaciers, but you can also doit for any regional subset of course.

The output of this script are the `ref_tstars.csv` and `crossval_tstars.csv` files, both found in the working directory. The `ref_tstars.csv` file can then be used for further runs, simply by copying it in the corresponding working directory.

Script

```
# Python imports
from os import path
from glob import glob

# Libs
import numpy as np
import pandas as pd
import geopandas as gpd

# Locals
import oggm
from oggm import cfg, utils, tasks, workflow
from oggm.workflow import execute_entity_task

# Module logger
import logging
log = logging.getLogger(__name__)

# RGI Version
rgi_version = '6'

# Initialize OGGM and set up the run parameters
cfg.initialize()

# Local paths (where to write the OGGM run output)
WORKING_DIR = path.join(path.expanduser('~'), 'tmp',
                        'OGGM_ref_mb_RGIV{}'.format(rgi_version))
utils.mkdir(WORKING_DIR, reset=True)
cfg.PATHS['working_dir'] = WORKING_DIR

# We are running the calibration ourselves
cfg.PARAMS['run_mb_calibration'] = True

# No need for intersects since this has an effect on the inversion only
cfg.PARAMS['use_intersects'] = False

# Use multiprocessing?
cfg.PARAMS['use_multiprocessing'] = True

# Set to True for operational runs
cfg.PARAMS['continue_on_error'] = False

# Pre-download other files which will be needed later
_ = utils.get_cru_file(var='tmp')
_ = utils.get_cru_file(var='pre')
```

```
rgi_dir = utils.get_rgi_dir(version=rgi_version)

# Get the reference glacier ids (they are different for each RGI version)
df, _ = utils.get_wgms_files(version=rgi_version)
rids = df['RGI{}0_ID'.format(rgi_version)]

# Make a new dataframe with those (this takes a while)
log.info('Reading the RGI shapefiles...')
rgidf = []
for reg in df['RGI_REG'].unique():
    if reg == '19':
        continue  # we have no climate data in Antarctica
    fn = '*' + reg + '_rgi{}0_*.shp'.format(rgi_version)
    fs = list(sorted(glob(path.join(rgi_dir, '*', fn))))[0]
    sh = gpd.read_file(fs)
    rgidf.append(sh.loc[sh.RGIId.isin(rids)])
rgidf = pd.concat(rgidf)
rgidf.crs = sh.crs  # for geolocalisation

# We have to check which of them actually have enough mb data.
# Let OGGM do it:
gdirs = workflow.init_glacier_regions(rgidf)
# We need to know which period we have data for
log.info('Process the climate data...')
execute_entity_task(tasks.process_cru_data, gdirs, print_log=False)
gdirs = utils.get_ref_mb_glaciers(gdirs)
# Keep only these
rgidf = rgidf.loc[rgidf.RGIId.isin([g.rgi_id for g in gdirs])]

# Save
log.info('For RGIV{} we have {} reference glaciers.'.format(rgi_version,
                                                            len(rgidf)))
rgidf.to_file(path.join(WORKING_DIR, 'mb_ref_glaciers.shp'))

# Sort for more efficient parallel computing
rgidf = rgidf.sort_values('Area', ascending=False)

# Go - initialize working directories
gdirs = workflow.init_glacier_regions(rgidf)

# Prepro tasks
task_list = [
    tasks.glacier_masks,
    tasks.compute_centerlines,
    tasks.initialize_flowlines,
    tasks.catchment_area,
    tasks.catchment_intersections,
    tasks.catchment_width_geom,
    tasks.catchment_width_correction,
]
for task in task_list:
    execute_entity_task(task, gdirs)

# Climate tasks
execute_entity_task(tasks.process_cru_data, gdirs)
tasks.compute_ref_t_stars(gdirs)
tasks.distribute_t_stars(gdirs)
execute_entity_task(tasks.apparent_mb, gdirs)
```

```
# Recompute after the first round - this is being picky but this is
# Because geometries may change after apparent_mb's filtering
tasks.compute_ref_t_stars(gdirs)
tasks.distribute_t_stars(gdirs)
execute_entity_task(tasks.apparent_mb, gdirs)

# Model validation
tasks.quick_crossval_t_stars(gdirs)  # for later
tasks.distribute_t_stars(gdirs)  # To restore after cross-val

# Tests: for all glaciers, the mass-balance around tstar and the
# bias with observation should be approx 0
from oggm.core.massbalance import (ConstantMassBalance, PastMassBalance)
for gd in gdirs:
    heights, widths = gd.get_inversion_flowline_hw()

    mb_mod = ConstantMassBalance(gd, bias=0)  # bias=0 because of calib!
    mb = mb_mod.get_specific_mb(heights, widths)
    np.testing.assert_allclose(mb, 0, atol=10)  # numerical errors

    mb_mod = PastMassBalance(gd)  # Here we need the computed bias
    refmb = gd.get_ref_mb_data().copy()
    refmb['OGGM'] = mb_mod.get_specific_mb(heights, widths, year=refmb.index)
    np.testing.assert_allclose(refmb.OGGM.mean(), refmb.ANNUAL_BALANCE.mean(),
                               atol=10)

# Log
log.info('Calibration is done!')
```

Cross-validation

The results of the cross-validation are found in the `crossval_tstars.csv` file. Let's replicate Figure 3 in Marzeion et al., (2012) :

```
# Python imports
from os import path

# Libs
import numpy as np
import pandas as pd
import geopandas as gpd

# Locals
import oggm
from oggm import cfg, workflow
from oggm.core.massbalance import PastMassBalance
import matplotlib.pyplot as plt

# RGI Version
rgi_version = '6'

# Initialize OGGM and set up the run parameters
cfg.initialize()

# Local paths (where to find the OGGM run output)
WORKING_DIR = path.join(path.expanduser('~'), 'tmp',
```

```
                          'OGGM_ref_mb_RGIV{}'.format(rgi_version))
cfg.PATHS['working_dir'] = WORKING_DIR

# Read the rgi file
rgidf = gpd.read_file(path.join(WORKING_DIR, 'mb_ref_glaciers.shp'))

# Go - initialize working directories
gdirs = workflow.init_glacier_regions(rgidf)

# Cross-validation
file = path.join(cfg.PATHS['working_dir'], 'crossval_tstars.csv')
cvdf = pd.read_csv(file, index_col=0)
for gd in gdirs:
    t_cvdf = cvdf.loc[gd.rgi_id]
    heights, widths = gd.get_inversion_flowline_hw()
    # Mass-balance model with cross-validated parameters instead
    mb_mod = PastMassBalance(gd, mu_star=t_cvdf.cv_mustar,
                             bias=t_cvdf.cv_bias,
                             prcp_fac=t_cvdf.cv_prcp_fac)
    # Mass-blaance timeseries, observed and simulated
    refmb = gd.get_ref_mb_data().copy()
    refmb['OGGM'] = mb_mod.get_specific_mb(heights, widths,
                                           year=refmb.index)
    # Compare their standard deviation
    std_ref = refmb.ANNUAL_BALANCE.std()
    rcor = np.corrcoef(refmb.OGGM, refmb.ANNUAL_BALANCE)[0, 1]
    if std_ref == 0:
        # I think that such a thing happens with some geodetic values
        std_ref = refmb.OGGM.std()
        rcor = 1
    # Store the scores
    cvdf.loc[gd.rgi_id, 'CV_MB_BIAS'] = (refmb.OGGM.mean() -
                                         refmb.ANNUAL_BALANCE.mean())
    cvdf.loc[gd.rgi_id, 'CV_MB_SIGMA_BIAS'] = (refmb.OGGM.std() /
                                               std_ref)
    cvdf.loc[gd.rgi_id, 'CV_MB_COR'] = rcor
    mb_mod = PastMassBalance(gd, mu_star=t_cvdf.interp_mustar,
                             bias=t_cvdf.cv_bias,
                             prcp_fac=t_cvdf.cv_prcp_fac)
    refmb['OGGM'] = mb_mod.get_specific_mb(heights, widths, year=refmb.index)
    cvdf.loc[gd.rgi_id, 'INTERP_MB_BIAS'] = (refmb.OGGM.mean() -
                                             refmb.ANNUAL_BALANCE.mean())

# Marzeion et al Figure 3
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6), sharey=True)
bins = np.arange(20) * 400 - 3800
cvdf['CV_MB_BIAS'].plot(ax=ax1, kind='hist', bins=bins, color='C3', label='')
ax1.vlines(cvdf['CV_MB_BIAS'].mean(), 0, 120, linestyles='--', label='Mean')
ax1.vlines(cvdf['CV_MB_BIAS'].quantile(), 0, 120, label='Median')
ax1.vlines(cvdf['CV_MB_BIAS'].quantile([0.05, 0.95]), 0, 120, color='grey',
                                       label='5% and 95%\npercentiles')
ax1.text(0.01, 0.99, 'N = {}'.format(len(gdirs)),
         horizontalalignment='left',
         verticalalignment='top',
         transform=ax1.transAxes)

ax1.set_ylim(0, 120)
ax1.set_ylabel('N Glaciers')
```

```
ax1.set_xlabel('Mass-balance error (mm w.e. yr$^{-1}$)')
ax1.legend(loc='best')
cvdf['INTERP_MB_BIAS'].plot(ax=ax2, kind='hist', bins=bins, color='C0')
ax2.vlines(cvdf['INTERP_MB_BIAS'].mean(), 0, 120, linestyles='--')
ax2.vlines(cvdf['INTERP_MB_BIAS'].quantile(), 0, 120)
ax2.vlines(cvdf['INTERP_MB_BIAS'].quantile([0.05, 0.95]), 0, 120, color='grey')
ax2.set_xlabel('Mass-balance error (mm w.e. yr$^{-1}$)')
plt.tight_layout()
fn = path.join(WORKING_DIR, 'mb_crossval_rgi{}.pdf'.format(rgi_version))
plt.savefig(fn)

print('Median bias: {:.2f}'.format(cvdf['CV_MB_BIAS'].median()))
print('Mean bias: {:.2f}'.format(cvdf['CV_MB_BIAS'].mean()))
print('RMS: {:.2f}'.format(np.sqrt(np.mean(cvdf['CV_MB_BIAS']**2))))
print('Sigma bias: {:.2f}'.format(np.mean(cvdf['CV_MB_SIGMA_BIAS'])))
```

This should generate a figure similar to:

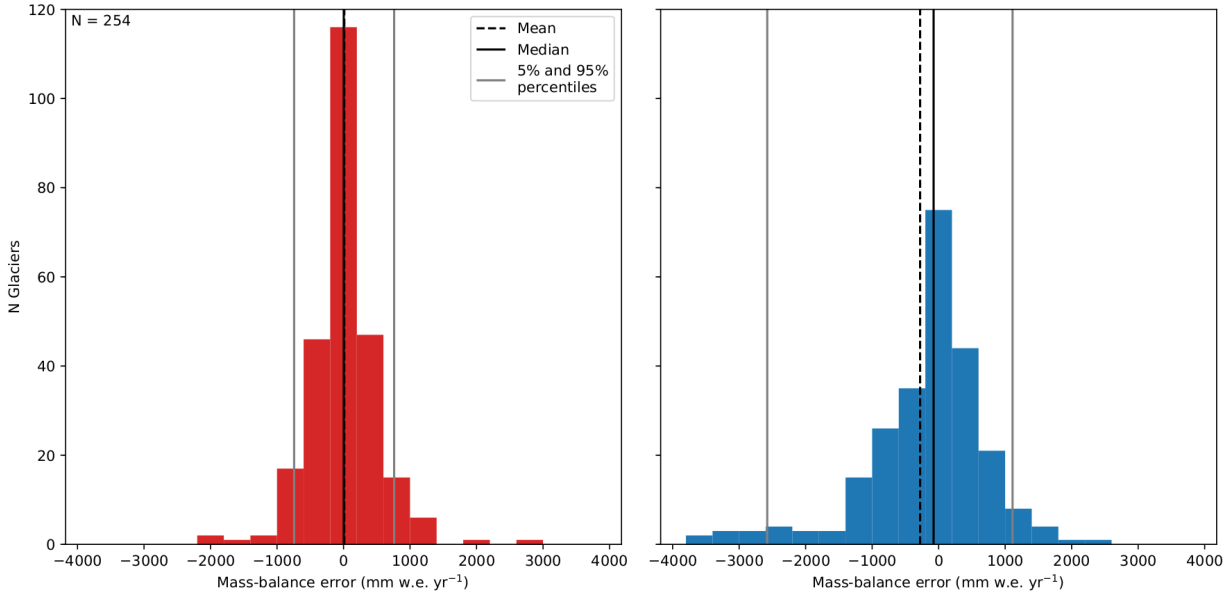


Fig. 1.3: Benefit of spatially interpolating t^* instead of μ^* as shown by leave-one-glacier-out cross-validation (N = 255). **Left**: error distribution of the computed mass-balance if determined by the interpolated t^*. **Right**: error distribution of the mass-balance if determined by interpolation of μ^*.

1.2.7 Developer documentation

Workflow

Tools to set-up and run OGGM.

workflow.init_glacier_regions	Very first task to do (always).
workflow.execute_entity_task	Execute a task on gdirs.
workflow.gis_prepro_tasks	Helper function: run all flowlines tasks.

Continued on next page

Table 1.1 – continued from previous page

workflow.climate_tasks	Helper function: run all climate tasks.
workflow.inversion_tasks	Helper function: run all bed inversion tasks.

oggm.workflow.init_glacier_regions

oggm.workflow.**init_glacier_regions**(*rgidf=None*, *reset=False*, *force=False*)
Very first task to do (always).

Set reset=True in order to delete the content of the directories.

oggm.workflow.execute_entity_task

oggm.workflow.**execute_entity_task**(*task*, *gdirs*, ***kwargs*)
Execute a task on gdirs.

If you asked for multiprocessing, it will do it.

Parameters **task** : function

the entity task to apply

gdirs : list

the list of oggm.GlacierDirectory to process.

oggm.workflow.gis_prepro_tasks

oggm.workflow.**gis_prepro_tasks**(*gdirs*)
Helper function: run all flowlines tasks.

oggm.workflow.climate_tasks

oggm.workflow.**climate_tasks**(*gdirs*)
Helper function: run all climate tasks.

oggm.workflow.inversion_tasks

oggm.workflow.**inversion_tasks**(*gdirs*)
Helper function: run all bed inversion tasks.

Entity tasks

Entity tasks are tasks which are applied on single glaciers individually and do not require information from other glaciers (this encompasses the majority of OGGM's tasks). They are parallelizable.

tasks.define_glacier_region	Very first task: define the glacier's local grid.
tasks.glacier_masks	Makes a gridded mask of the glacier outlines.
tasks.compute_centerlines	Compute the centerlines following Kienholz et al., (2014).

Continued on next page

Table 1.2 – continued from previous page

tasks.initialize_flowlines	Transforms the geometrical Centerlines in the more "physical" "Inversion Flowlines".
tasks.compute_downstream_line	Compute the line continuing the glacier.
tasks.compute_downstream_bedshape	The bedshape obtained by fitting a parabola to the line's normals.
tasks.catchment_area	Compute the catchment areas of each tributary line.
tasks.catchment_intersections	Computes the intersections between the catchments.
tasks.catchment_width_geom	Compute geometrical catchment widths for each point of the flowlines.
tasks.catchment_width_correction	Corrects for NaNs and inconsistencies in the geometrical widths.
tasks.process_cru_data	Processes and writes the climate data for this glacier.
tasks.process_custom_climate_data	Processes and writes the climate data from a user-defined climate file.
tasks.process_cesm_data	Processes and writes the climate data for this glacier.
tasks.local_mustar	Compute the local mustar from interpolated tstars.
tasks.apparent_mb	Compute the apparent mb from the calibrated mustar.
tasks.apparent_mb_from_linear_mb	Compute apparent mb from a linear mass-balance assumption (for testing).
tasks.mu_candidates	Computes the mu candidates.
tasks.prepare_for_inversion	Prepares the data needed for the inversion.
tasks.volume_inversion	Computes the inversion the glacier.
tasks.distribute_thickness	Compute a thickness map of the glacier using the nearest centerlines.
tasks.init_present_time_glacier	First task after inversion.
tasks.random_glacier_evolution	Random glacier dynamics for benchmarking purposes.
tasks.iterative_initial_glacier_search	Iterative search for the glacier in year y0.
tasks.run_from_climate_data	Runs glacier with climate input from a general circulation model.
tasks.run_constant_climate	Run a glacier under a constant climate for a given climate period.

oggm.tasks.define_glacier_region

oggm.tasks.**define_glacier_region**(*gdir*, *entity=None*)

Very first task: define the glacier's local grid.

Defines the local projection (Transverse Mercator), centered on the glacier. There is some options to set the resolution of the local grid. It can be adapted depending on the size of the glacier with:

```
dx (m) = d1 * AREA (km) + d2 ; clipped to dmax
```

or be set to a fixed value. See `params.cfg` for setting these options. Default values of the adapted mode lead to a resolution of 50 m for Hintereisferner, which is approx. 8 km2 large. After defining the grid, the topography and the outlines of the glacier are transformed into the local projection. The default interpolation for the topography is *cubic*.

Parameters **gdir** : *oggm.GlacierDirectory*

where to write the data

entity : geopandas GeoSeries

the glacier geometry to process

Returns Files writen to the glacier directory:

dem.tif A geotiff file containing the DEM (reprojected into the local grid).

glacier_grid.json A `salem.Grid` handling the georeferencing of the local grid.

outlines.shp The glacier outlines in the local projection.

oggm.tasks.glacier_masks

`oggm.tasks.`**`glacier_masks`**(*gdir*)

Makes a gridded mask of the glacier outlines.

Parameters **gdir** : *oggm.GlacierDirectory*

where to write the data

Returns Files writen to the glacier directory:

geometries.pkl A `dict` containing the shapely.Polygons of a glacier. The "polygon_hr" entry contains the geometry transformed to the local grid in (i, j) coordinates, while the "polygon_pix" entry contains the geometries transformed into the coarse grid (the i, j elements are integers). The "polygon_area" entry contains the area of the polygon as computed by Shapely.

gridded_data.nc A netcdf file containing several gridded data variables such as topography, the glacier masks and more (see the netCDF file metadata).

oggm.tasks.compute_centerlines

`oggm.tasks.`**`compute_centerlines`**(*gdir*, *heads=None*)

Compute the centerlines following Kienholz et al., (2014).

They are then sorted according to the modified Strahler number: http://en.wikipedia.org/wiki/Strahler_number

Parameters **heads** : list, optional

list of shpg.Points to use as line heads (default is to compute them like Kienholz did)

Returns Files writen to the glacier directory:

centerlines.pkl A list of `Centerline` instances, sorted by flow order.

gridded_data.nc A netcdf file containing several gridded data variables such as topography, the glacier masks and more (see the netCDF file metadata).

oggm.tasks.initialize_flowlines

`oggm.tasks.`**`initialize_flowlines`**(*gdir*)

Transforms the geometrical Centerlines in the more "physical" "Inversion Flowlines".

This interpolates the centerlines on a regular spacing (i.e. not the grid's (i, j) indices. Cuts out the tail of the tributaries to make more realistic junctions. Also checks for low and negative slopes and corrects them by interpolation.

Parameters **gdir** : oggm.GlacierDirectory

Returns Files writen to the glacier directory:

inversion_flowlines.pkl A "better" version of the Centerlines, now on a regular spacing i.e., not on the gridded (i, j) indices. The tails of the tributaries are cut out to make more realistic junctions. They are now "1.5D" i.e., with a width.

oggm.tasks.compute_downstream_line

oggm.tasks.**compute_downstream_line**(*gdir*)
Compute the line continuing the glacier.

The idea is simple: starting from the glacier tail, compute all the routes to all local minimas found at the domain edge. The cheapest is "The One".

The rest of the job (merging centerlines + downstream into one single glacier is realized by *init_present_time_glacier()*).

Parameters **gdir** : oggm.GlacierDirectory

Returns Files writen to the glacier directory:

downstream_line.pkl A `dict` containing the downsteam line geometry as well as the bedshape computed from a parabolic fit.

oggm.tasks.compute_downstream_bedshape

oggm.tasks.**compute_downstream_bedshape**(*gdir*)
The bedshape obtained by fitting a parabola to the line's normals. Also computes the downstream's altitude.

Parameters **gdir** : oggm.GlacierDirectory

Returns Files writen to the glacier directory:

downstream_line.pkl A `dict` containing the downsteam line geometry as well as the bedshape computed from a parabolic fit.

oggm.tasks.catchment_area

oggm.tasks.**catchment_area**(*gdir*)
Compute the catchment areas of each tributary line.

The idea is to compute the route of lowest cost for any point on the glacier to rejoin a centerline. These routes are then put together if they belong to the same centerline, thus creating "catchment areas" for each centerline.

Parameters **gdir** : oggm.GlacierDirectory

Returns Files writen to the glacier directory:

catchment_indices.pkl A list of len n_centerlines, each element conaining a numpy array of the indices in the glacier grid which represent the centerline's catchment area.

oggm.tasks.catchment_intersections

oggm.tasks.**catchment_intersections**(*gdir*)
Computes the intersections between the catchments.

Parameters **gdir** : oggm.GlacierDirectory

Returns Files writen to the glacier directory:

catchments_intersects.shp The catchments intersections in the local projection.

flowline_catchments.shp The flowline catchments in the local projection.

oggm.tasks.catchment_width_geom

oggm.tasks.**catchment_width_geom**(*gdir*)
Compute geometrical catchment widths for each point of the flowlines.

Updates the 'inversion_flowlines' save file.

Parameters **gdir** : oggm.GlacierDirectory

Returns Files writen to the glacier directory:

inversion_flowlines.pkl A "better" version of the Centerlines, now on a regular spacing i.e., not on the gridded (i, j) indices. The tails of the tributaries are cut out to make more realistic junctions. They are now "1.5D" i.e., with a width.

oggm.tasks.catchment_width_correction

oggm.tasks.**catchment_width_correction**(*gdir*)
Corrects for NaNs and inconsistencies in the geometrical widths.

Interpolates missing values, ensures consistency of the surface-area distribution AND with the geometrical area of the glacier polygon, avoiding errors due to gridded representation.

Updates the 'inversion_flowlines' save file.

Parameters **gdir** : oggm.GlacierDirectory

Returns Files writen to the glacier directory:

inversion_flowlines.pkl A "better" version of the Centerlines, now on a regular spacing i.e., not on the gridded (i, j) indices. The tails of the tributaries are cut out to make more realistic junctions. They are now "1.5D" i.e., with a width.

oggm.tasks.process_cru_data

oggm.tasks.**process_cru_data**(*gdir*)
Processes and writes the climate data for this glacier.

Interpolates the CRU TS data to the high-resolution CL2 climatologies (provided with OGGM) and writes everything to a NetCDF file.

Returns Files writen to the glacier directory:

climate_monthly.nc The monthly climate timeseries for this glacier, stored in a netCDF file.

oggm.tasks.process_custom_climate_data

oggm.tasks.**process_custom_climate_data**(*gdir*)
Processes and writes the climate data from a user-defined climate file.

The input file must have a specific format (see oggm-sample-data/test-files/histalp_merged_hef.nc for an example).

Uses caching for faster retrieval.

This is the way OGGM does it for the Alps (HISTALP).

Returns Files writen to the glacier directory:

climate_monthly.nc The monthly climate timeseries for this glacier, stored in a netCDF file.

oggm.tasks.process_cesm_data

oggm.tasks.**process_cesm_data**(*gdir*, *filesuffix=''*, *fpath_temp=None*, *fpath_precc=None*, *fpath_precl=None*)

Processes and writes the climate data for this glacier.

This function is made for interpolating the Community Earth System Model Last Millenial Ensemble (CESM-LME) climate simulations, from Otto-Bliesner et al. (2016), to the high-resolution CL2 climatologies (provided with OGGM) and writes everything to a NetCDF file.

Parameters **filesuffix** : str

append a suffix to the filename (useful for ensemble experiments).

fpath_temp : str

path to the temp file (default: cfg.PATHS['gcm_temp_file'])

fpath_precc : str

path to the precc file (default: cfg.PATHS['gcm_precc_file'])

fpath_precl : str

path to the precl file (default: cfg.PATHS['gcm_precl_file'])

Returns Files writen to the glacier directory:

cesm_data.nc The monthly GCM climate timeseries for this glacier, stored in a netCDF file.

oggm.tasks.local_mustar

oggm.tasks.**local_mustar**(*gdir*, *tstar=None*, *bias=None*, *prcp_fac=None*, *minimum_mustar=0.0*)

Compute the local mustar from interpolated tstars.

Parameters **gdir** : oggm.GlacierDirectory

tstar: int

the year where the glacier should be equilibrium

bias: int

the associated reference bias

prcp_fac: int

the associated precipitation factor

minimum_mustar: float

if mustar goes below this threshold, clip it to that value. If you want this to happen with *minimum_mustar=0.* you will have to set *cfg.PARAMS['allow_negative_mustar']=True* first.

Returns Files writen to the glacier directory:

local_mustar.csv A csv with three values: the local scalars mu*, t*, bias

oggm.tasks.apparent_mb

oggm.tasks.**apparent_mb**(*gdir*)

Compute the apparent mb from the calibrated mustar.

Returns Files writen to the glacier directory:

inversion_flowlines.pkl A "better" version of the Centerlines, now on a regular spacing i.e., not on the gridded (i, j) indices. The tails of the tributaries are cut out to make more realistic junctions. They are now "1.5D" i.e., with a width.

oggm.tasks.apparent_mb_from_linear_mb

oggm.tasks.**apparent_mb_from_linear_mb**(*gdir*, *mb_gradient=3.0*)

Compute apparent mb from a linear mass-balance assumption (for testing).

This is for testing currently, but could be used as alternative method for the inversion quite easily.

Parameters **gdir** : oggm.GlacierDirectory

Returns Files writen to the glacier directory:

inversion_flowlines.pkl A "better" version of the Centerlines, now on a regular spacing i.e., not on the gridded (i, j) indices. The tails of the tributaries are cut out to make more realistic junctions. They are now "1.5D" i.e., with a width.

linear_mb_params.pkl When using a linear mass-balance for the inversion, this dict stores the optimal ela_h and grad.

oggm.tasks.mu_candidates

oggm.tasks.**mu_candidates**(*gdir*, *prcp_sf=None*)

Computes the mu candidates.

For each 31 year-period centered on the year of interest, mu is is the temperature sensitivity necessary for the glacier with its current shape to be in equilibrium with its climate.

For glaciers with MB data only!

Parameters **gdir** : oggm.GlacierDirectory

prcp_sf : float (optional)

force to a certain prcp scaling factor

Returns Files writen to the glacier directory:

mu_candidates.pkl A pandas.Series with the (year, mu) data.

oggm.tasks.prepare_for_inversion

oggm.tasks.**prepare_for_inversion**(*gdir*, *add_debug_var=False*, *invert_with_rectangular=True*, *invert_all_rectangular=False*)

Prepares the data needed for the inversion.

Mostly the mass flux and slope angle, the rest (width, height) was already computed. It is then stored in a list of dicts in order to be faster.

Parameters **gdir** : oggm.GlacierDirectory

Returns Files writen to the glacier directory:

inversion_input.pkl List of dicts containing the data needed for the inversion.

oggm.tasks.volume_inversion

oggm.tasks.**volume_inversion**(*gdir*, *glen_a=None*, *fs=None*, *filesuffix=''*)

Computes the inversion the glacier.

If glen_a and fs are not given, it will use the optimized params.

Parameters **gdir** : oggm.GlacierDirectory

glen_a : float, optional

the ice creep parameter (defaults to cfg.PARAMS['inversion_glen_a'])

fs : float, optional

the sliding parameter (defaults to cfg.PARAMS['inversion_fs'])

fs : float, optional

the sliding parameter (defaults to cfg.PARAMS['inversion_fs'])

filesuffix : str

add a suffix to the output file

Returns Files writen to the glacier directory:

inversion_output.pkl List of dicts containing the output data from the inversion.

oggm.tasks.distribute_thickness

oggm.tasks.**distribute_thickness**(*gdir*, *how=''*, *add_slope=True*, *smooth=True*, *add_nc_name=False*)

Compute a thickness map of the glacier using the nearest centerlines.

This is a rather cosmetic task, not relevant for OGGM but for ITMIX. Here we take the nearest neighbors in a certain altitude range.

Returns Files writen to the glacier directory:

gridded_data.nc A netcdf file containing several gridded data variables such as topography, the glacier masks and more (see the netCDF file metadata).

oggm.tasks.init_present_time_glacier

oggm.tasks.**init_present_time_glacier**(*gdir*)

First task after inversion. Merges the data from the various preprocessing tasks into a stand-alone numerical glacier ready for run.

Parameters **gdir** : oggm.GlacierDirectory

Returns Files writen to the glacier directory:

model_flowlines.pkl List of flowlines ready to be run by the model.

oggm.tasks.random_glacier_evolution

oggm.tasks.**random_glacier_evolution**(*gdir*, *nyears=1000*, *y0=None*, *halfsize=15*, *bias=None*, *seed=None*, *temperature_bias=None*, *filename='climate_monthly'*, *input_filesuffix=''*, *filesuffix=''*, *init_model_fls=None*, *zero_initial_glacier=False*, ***kwargs*)

Random glacier dynamics for benchmarking purposes.

This runs the random mass-balance model for a certain number of years.

Parameters **nyears** : int

length of the simulation

y0 [int, optional] central year of the random climate period. The default is to be centred on t*.

halfsize [int, optional] the half-size of the time window (window size = 2 * halfsize + 1)

bias [float] bias of the mb model. Default is to use the calibrated one, which is often a better idea. For t* experiments it can be useful to set it to zero

seed [int] seed for the random generator. If you ignore this, the runs will be different each time. Setting it to a fixed seed accross glaciers can be usefull if you want to have the same climate years for all of them

temperature_bias [float] add a bias to the temperature timeseries

filename [str] name of the climate file, e.g. 'climate_monthly' (default) or 'cesm_data'

input_filesuffix: str filesuffix for the input climate file

filesuffix [str] this add a suffix to the output file (useful to avoid overwriting previous experiments)

init_model_fls [[]] list of flowlines to use to initialise the model (the default is the present_time_glacier file from the glacier directory)

zero_initial_glacier [bool] if true, the ice thickness is set to zero before the simulation

kwargs [dict] kwargs to pass to the FluxBasedModel instance

Returns Files writen to the glacier directory:

oggm.tasks.iterative_initial_glacier_search

oggm.tasks.**iterative_initial_glacier_search**(*gdir*, *y0=None*, *init_bias=0.0*, *rtol=0.005*, *write_steps=True*)

Iterative search for the glacier in year y0.

this is outdated and doesn't really work.

Returns Files writen to the glacier directory:

model_run.nc A netcdf file containing enough information to reconstruct the entire flowline glacier along the run (can be data expensive).

oggm.tasks.run_from_climate_data

oggm.tasks.**run_from_climate_data**(*gdir*, *ys=None*, *ye=None*, *filename='climate_monthly'*, *input_filesuffix=''*, *filesuffix=''*, *init_model_fls=None*, ***kwargs*)

Runs glacier with climate input from a general circulation model.

Parameters **ys** : int

start year of the model run (default: from the config file)

y1 [int] end year of the model run (default: from the config file)

filename [str] name of the climate file, e.g. 'climate_monthly' (default) or 'cesm_data'

input_filesuffix: str filesuffix for the input climate file

filesuffix [str] for the output file

init_model_fls [[]] list of flowlines to use to initialise the model (the default is the present_time_glacier file from the glacier directory)

kwargs [dict] kwargs to pass to the FluxBasedModel instance

Returns Files writen to the glacier directory:

oggm.tasks.run_constant_climate

oggm.tasks.**run_constant_climate**(*gdir*, *nyears=1000*, *y0=None*, *bias=None*, *temperature_bias=None*, *filesuffix=''*, *filename='climate_monthly'*, *input_filesuffix=''*, *init_model_fls=None*, *zero_initial_glacier=False*, ***kwargs*)

Run a glacier under a constant climate for a given climate period.

Parameters **nyears** : int

length of the simulation (default: as long as needed for reaching equilibrium)

y0 [int] central year of the requested climate period. The default is to be centred on t*.

bias [float] bias of the mb model. Default is to use the calibrated one, which is often a better idea. For t* experiments it can be useful to set it to zero

temperature_bias [float] add a bias to the temperature timeseries

filename [str] name of the climate file, e.g. 'climate_monthly' (default) or 'cesm_data'

input_filesuffix: str filesuffix for the input climate file

filesuffix [str] this add a suffix to the output file (useful to avoid overwriting previous experiments)

zero_initial_glacier [bool] if true, the ice thickness is set to zero before the simulation

init_model_fls [[]] list of flowlines to use to initialise the model (the default is the present_time_glacier file from the glacier directory)

kwargs [dict] kwargs to pass to the FluxBasedModel instance

Returns Files writen to the glacier directory:

Global tasks

Global tasks are tasks which are run on a set of glaciers (most often: all glaciers in the current run). They are not parallelizable.

tasks.compute_ref_t_stars	Detects the best t* for the reference glaciers.
tasks.distribute_t_stars	After the computation of the reference tstars, apply the interpolation to each individual glacier.
tasks.crossval_t_stars	Cross-validate the interpolation of tstar to each individual glacier.
tasks.optimize_inversion_params	Optimizes fs and fd based on GlaThiDa thicknesses.

oggm.tasks.compute_ref_t_stars

oggm.tasks.**compute_ref_t_stars**(*gdirs*)

Detects the best t* for the reference glaciers.

Parameters **gdirs: list of oggm.GlacierDirectory objects**

oggm.tasks.distribute_t_stars

oggm.tasks.**distribute_t_stars**(*gdirs*, *ref_df=None*, *minimum_mustar=0.0*)

After the computation of the reference tstars, apply the interpolation to each individual glacier.

Parameters **gdirs** : []

list of oggm.GlacierDirectory objects

ref_df : pd.Dataframe

replace the default calibration list

minimum_mustar: float

if mustar goes below this threshold, clip it to that value. If you want this to happen with *minimum_mustar=0.* you will have to set *cfg.PARAMS['allow_negative_mustar']=True* first.

oggm.tasks.crossval_t_stars

oggm.tasks.**crossval_t_stars**(*gdirs*)
Cross-validate the interpolation of tstar to each individual glacier.

This is a thorough check (redoes many calculations) because it recomputes the chosen tstars at each time. You should use quick_crossval_t_stars instead.

Parameters **gdirs: list of oggm.GlacierDirectory objects**

oggm.tasks.optimize_inversion_params

oggm.tasks.**optimize_inversion_params**(*gdirs*)
Optimizes fs and fd based on GlaThiDa thicknesses.

We use the glacier averaged thicknesses provided by GlaThiDa and correct them for differences in area with RGI, using a glacier specific volume-area scaling formula.

Parameters **gdirs: list of oggm.GlacierDirectory objects**

Classes

Listed here are the classes which are relevant at the API level (i.e. classes which are used and re-used across modules and tasks).

TODO: add the model classes, etc.

GlacierDirectory	Organizes read and write access to the glacier's files.
Centerline	A Centerline has geometrical and flow rooting properties.
Flowline	The is the input flowline for the model.

oggm.GlacierDirectory

class oggm.**GlacierDirectory**(*rgi_entity*, *base_dir=None*, *reset=False*)
Organizes read and write access to the glacier's files.

It handles a glacier directory created in a base directory (default is the "per_glacier" folder in the working directory). The role of a GlacierDirectory is to give access to file paths and to I/O operations. The user should not care about *where* the files are located, but should know their name (see *cfg.BASENAMES*).

If the directory does not exist, it will be created.

See *Glacier working directories* for more information.

Attributes

dir	(str) path to the directory
rgi_id	(str) The glacier's RGI identifier
glims_id	(str) The glacier's GLIMS identifier (when available)
rgi_area_km2	(float) The glacier's RGI area (km2)
cenlon, cenlat	(float) The glacier centerpoint's lon/lat
rgi_date	(datetime) The RGI's BGNDATE attribute if available. Otherwise, defaults to 2003-01-01
rgi_region	(str) The RGI region name
name	(str) The RGI glacier name (if Available)
glacier_type	(str) The RGI glacier type ('Glacier', 'Ice cap', 'Perennial snowfield', 'Seasonal snowfield')
terminus_type	(str) The RGI terminus type ('Land-terminating', 'Marine-terminating', 'Lake-terminating', 'Dry calving', 'Regenerated', 'Shelf-terminating')
is_tidewater	(bool) Is the glacier a caving glacier?
inversion_calving_rate	(float) Calving rate used for the inversion

__init__(*rgi_entity*, *base_dir=None*, *reset=False*)

Creates a new directory or opens an existing one.

Parameters **rgi_entity** : a GeoSeries or str

glacier entity read from the shapefile (or a valid RGI ID if the directory exists)

base_dir : str

path to the directory where to open the directory. Defaults to *cfg.PATHS['working_dir'] + /per_glacier/*

reset : bool, default=False

empties the directory at construction (careful!)

Methods

__init__(rgi_entity[, base_dir, reset])	Creates a new directory or opens an existing one.
copy_to_basedir(base_dir[, setup])	Copies the glacier directory and its content to a new location.
create_gridded_ncdf_file(fname)	Makes a gridded netcdf file template.
get_filepath(filename[, delete, filesuffix])	Absolute path to a specific file.
get_inversion_flowline_hw()	Shortcut function to read the heights and widths of the glacier.
get_ref_length_data()	Get the glacier lenght data from P.
get_ref_mb_data()	Get the reference mb data from WGMS (for some glaciers only!).
get_task_status(task_name)	Opens this directory's log file to see if a task was already run.
has_file(filename)	Checks if a file exists.
log(task_name[, err])	Logs a message to the glacier directory.
read_pickle(filename[, use_compression, ...])	Reads a pickle located in the directory.

Continued on next page

Table 1.5 – continued from previous page

write_monthly_climate_file(time, prcp, temp, ...)	Creates a netCDF4 file with climate data.
write_pickle(var, filename[, ...])	Writes a variable to a pickle on disk.

oggm.Centerline

class oggm.**Centerline**(*line*, *dx=None*, *surface_h=None*, *orig_head=None*)

A Centerline has geometrical and flow rooting properties.

It is instanciated and updated by _join_lines() exclusively

__init__(*line*, *dx=None*, *surface_h=None*, *orig_head=None*)

Instanciate.

Parameters **line: Shapely LineString**

Methods

__init__(line[, dx, surface_h, orig_head])	Instanciate.
set_apparent_mb(mb)	Set the apparent mb and flux for the flowline.
set_flows_to(other[, check_tail, last_point])	Find the closest point in "other" and sets all the corresponding attributes.
set_line(line)	Update the Shapely LineString coordinate.

oggm.Flowline

class oggm.**Flowline**(*line=None*, *dx=1*, *map_dx=None*, *surface_h=None*, *bed_h=None*)

The is the input flowline for the model.

__init__(*line=None*, *dx=1*, *map_dx=None*, *surface_h=None*, *bed_h=None*)

Instanciate.

#TODO: documentation

Methods

__init__([line, dx, map_dx, surface_h, bed_h])	Instanciate.
set_apparent_mb(mb)	Set the apparent mb and flux for the flowline.
set_flows_to(other[, check_tail, last_point])	Find the closest point in "other" and sets all the corresponding attributes.
set_line(line)	Update the Shapely LineString coordinate.
to_dataset()	Makes an xarray Dataset out of the flowline.

Mass-balance

Mass-balance models found in the core.massbalance module.

They follow the *MassBalanceModel()* interface. Here is a quick summary of the units and conventions used by all models:

Units

The computed mass-balance is in units of [m ice s-1] ("meters of ice per second"), unless otherwise specified (e.g. for the utility function `get_specific_mb`). The conversion from the climatic mass-balance ([kg m-2 s-1]) therefore assumes an ice density given by `cfg.RHO` (currently: 900 kg m-3).

Time

The time system used by OGGM is a simple "fraction of year" system, where the floating year can be used for conversion to months and years:

```
In [1]: from oggm.utils import floatyear_to_date, date_to_floatyear

In [2]: date_to_floatyear(1982, 12)
Out[2]: 1982.9166666666667

In [3]: floatyear_to_date(1.2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[3]: (1, 3)
```

Interface

MassBalanceModel	Common logic for the mass balance models.
MassBalanceModel.get_monthly_mb	Monthly mass-balance at given altitude(s) for a moment in time.
MassBalanceModel.get_annual_mb	Like *self.get_monthly_mb()*, but for annual MB.
MassBalanceModel.get_specific_mb	Specific mb for this year and a specific glacier geometry.
MassBalanceModel.temp_bias	Temperature bias to add to the original series.

oggm.core.massbalance.MassBalanceModel

class oggm.core.massbalance.**MassBalanceModel**

Common logic for the mass balance models.

All mass-balance models should implement this interface.

__init__()

Initialize.

Methods

__init__()	Initialize.
get_annual_mb(heights[, year])	Like *self.get_monthly_mb()*, but for annual MB.
get_ela([year])	Compute the equilibrium line altitude for this year
get_monthly_mb(heights[, year])	Monthly mass-balance at given altitude(s) for a moment in time.
get_specific_mb(heights, widths[, year])	Specific mb for this year and a specific glacier geometry.

oggm.core.massbalance.MassBalanceModel.get_monthly_mb

MassBalanceModel.**get_monthly_mb**(*heights*, *year=None*)

Monthly mass-balance at given altitude(s) for a moment in time.

Units: [m s-1], or meters of ice per second

Note: *year* is optional because some simpler models have no time component.

Parameters **heights: ndarray**

the atitudes at which the mass-balance will be computed

year: float, optional

the time (in the "hydrological floating year" convention)

Returns the mass-balance (same dim as *heights*) (units: [m s-1])

oggm.core.massbalance.MassBalanceModel.get_annual_mb

MassBalanceModel.**get_annual_mb**(*heights*, *year=None*)

Like *self.get_monthly_mb()*, but for annual MB.

For some simpler mass-balance models `get_monthly_mb()` and `get_annual_mb()` can be equivalent.

Units: [m s-1], or meters of ice per second

Note: *year* is optional because some simpler models have no time component.

Parameters **heights: ndarray**

the atitudes at which the mass-balance will be computed

year: float, optional

the time (in the "floating year" convention)

Returns the mass-balance (same dim as *heights*) (units: [m s-1])

oggm.core.massbalance.MassBalanceModel.get_specific_mb

MassBalanceModel.**get_specific_mb**(*heights*, *widths*, *year=None*)

Specific mb for this year and a specific glacier geometry.

Units: [mm w.e. yr-1], or millimeter water equivalent per year

Parameters **heights: ndarray**

the atitudes at which the mass-balance will be computed

widths: ndarray

the widths of the flowline (necessary for the weighted average)

year: float, optional

the time (in the "hydrological floating year" convention)

Returns the specific mass-balance (units: mm w.e. yr-1)

oggm.core.massbalance.MassBalanceModel.temp_bias

MassBalanceModel.**temp_bias**
Temperature bias to add to the original series.

Models

LinearMassBalance	Constant mass-balance as a linear function of altitude.
PastMassBalance	Mass balance during the climate data period.
ConstantMassBalance	Constant mass-balance during a chosen period.
RandomMassBalance	Random shuffle of all MB years within a given time period.

oggm.core.massbalance.LinearMassBalance

class oggm.core.massbalance.**LinearMassBalance**(*ela_h*, *grad=3.0*, *max_mb=None*)
Constant mass-balance as a linear function of altitude.

The "temperature bias" doesn't makes much sense in this context, but we implemented a simple empirical rule: + 1K -> ELA + 150 m

__init__(*ela_h*, *grad=3.0*, *max_mb=None*)
Initialize.

Parameters **ela_h: float**

Equilibrium line altitude (units: [m])

grad: float

Mass-balance gradient (unit: [mm w.e. yr-1 m-1])

max_mb: float

Cap the mass balance to a certain value (unit: [mm w.e. yr-1])

Methods

__init__(ela_h[, grad, max_mb])	Initialize.
get_annual_mb(heights[, year])	Like *self.get_monthly_mb()*, but for annual MB.
get_ela([year])	Compute the equilibrium line altitude for this year
get_monthly_mb(heights[, year])	Monthly mass-balance at given altitude(s) for a moment in time.
get_specific_mb(heights, widths[, year])	Specific mb for this year and a specific glacier geometry.

oggm.core.massbalance.PastMassBalance

class oggm.core.massbalance.**PastMassBalance**(*gdir*, *mu_star=None*, *bias=None*, *prcp_fac=None*, *filename='climate_monthly'*, *input_filesuffix=''*)
Mass balance during the climate data period.

__init__(*gdir*, *mu_star=None*, *bias=None*, *prcp_fac=None*, *filename='climate_monthly'*, *input_filesuffix=''*)

Initialize.

Parameters **gdir** : GlacierDirectory

the glacier directory

mu_star : float, optional

set to the alternative value of mustar you want to use (the default is to use the calibrated value)

bias : float, optional

set to the alternative value of the calibration bias [mm we yr-1] you want to use (the default is to use the calibrated value) Note that this bias is *substracted* from the computed MB. Indeed: BIAS = MODEL_MB - REFERENCE_MB.

prcp_fac : float, optional

set to the alternative value of the precipitation factor you want to use (the default is to use the calibrated value)

filename : str, optional

set to a different BASENAME if you want to use alternative climate data.

input_filesuffix : str

the file suffix of the input climate file

Methods

__init__(gdir[, mu_star, bias, prcp_fac, ...])	Initialize.
get_annual_mb(heights[, year])	Like *self.get_monthly_mb()*, but for annual MB.
get_ela([year])	Compute the equilibrium line altitude for this year
get_monthly_climate(heights[, year])	Monthly climate information at given heights.
get_monthly_mb(heights[, year])	Monthly mass-balance at given altitude(s) for a moment in time.
get_specific_mb(heights, widths[, year])	Specific mb for this year and a specific glacier geometry.

oggm.core.massbalance.ConstantMassBalance

class oggm.core.massbalance.**ConstantMassBalance**(*gdir*, *mu_star=None*, *bias=None*, *prcp_fac=None*, *y0=None*, *halfsize=15*, *filename='climate_monthly'*, *input_filesuffix=''*)

Constant mass-balance during a chosen period.

This is useful for equilibrium experiments.

__init__(*gdir*, *mu_star=None*, *bias=None*, *prcp_fac=None*, *y0=None*, *halfsize=15*, *filename='climate_monthly'*, *input_filesuffix=''*)

Initialize

Parameters **gdir** : GlacierDirectory

the glacier directory

mu_star : float, optional

set to the alternative value of mustar you want to use (the default is to use the calibrated value)

bias : float, optional

set to the alternative value of the annual bias [mm we yr-1] you want to use (the default is to use the calibrated value)

prcp_fac : float, optional

set to the alternative value of the precipitation factor you want to use (the default is to use the calibrated value)

y0 : int, optional, default: tstar

the year at the center of the period of interest. The default is to use tstar as center.

halfsize : int, optional

the half-size of the time window (window size = 2 * halfsize + 1)

filename : str, optional

set to a different BASENAME if you want to use alternative climate data.

input_filesuffix : str

the file suffix of the input climate file

Methods

`__init__`(gdir[, mu_star, bias, prcp_fac, ...])	Initialize
`get_annual_mb`(heights[, year])	Like *self.get_monthly_mb()*, but for annual MB.
`get_climate`(heights[, year])	Average climate information at given heights.
`get_ela`([year])	Compute the equilibrium line altitude for this year
`get_monthly_mb`(heights[, year])	Monthly mass-balance at given altitude(s) for a moment in time.
`get_specific_mb`(heights, widths[, year])	Specific mb for this year and a specific glacier geometry.

oggm.core.massbalance.RandomMassBalance

class oggm.core.massbalance.**RandomMassBalance**(*gdir*, *mu_star=None*, *bias=None*, *prcp_fac=None*, *y0=None*, *halfsize=15*, *seed=None*, *filename='climate_monthly'*, *input_filesuffix=''*)

Random shuffle of all MB years within a given time period.

This is useful for finding a possible past glacier state or for sensitivity experiments.

Note that this is going to be sensitive to extreme years in certain periods, but it is by far more physically reasonable than other approaches based on gaussian assumptions.

__init__(*gdir*, *mu_star=None*, *bias=None*, *prcp_fac=None*, *y0=None*, *halfsize=15*, *seed=None*, *filename='climate_monthly'*, *input_filesuffix=''*)

Initialize.

Parameters **gdir** : GlacierDirectory

the glacier directory

mu_star : float, optional

set to the alternative value of mustar you want to use (the default is to use the calibrated value)

bias : float, optional

set to the alternative value of the calibration bias [mm we yr-1] you want to use (the default is to use the calibrated value) Note that this bias is *substracted* from the computed MB. Indeed: BIAS = MODEL_MB - REFERENCE_MB.

prcp_fac : float, optional

set to the alternative value of the precipitation factor you want to use (the default is to use the calibrated value)

y0 : int, optional, default: tstar

the year at the center of the period of interest. The default is to use tstar as center.

halfsize : int, optional

the half-size of the time window (window size = 2 * halfsize + 1)

seed : int, optional

Random seed used to initialize the pseudo-random number generator.

filename : str, optional

set to a different BASENAME if you want to use alternative climate data.

input_filesuffix : str

the file suffix of the input climate file

Methods

`__init__`(gdir[, mu_star, bias, prcp_fac, ...])	Initialize.
`get_annual_mb`(heights[, year])	Like *self.get_monthly_mb()*, but for annual MB.
`get_ela`([year])	Compute the equilibrium line altitude for this year
`get_monthly_mb`(heights[, year])	Monthly mass-balance at given altitude(s) for a moment in time.
`get_specific_mb`(heights, widths[, year])	Specific mb for this year and a specific glacier geometry.
`get_state_yr`([year])	For a given year, get the random year associated to it.

1.2.8 Version history

v1.0 (16 January 2018)

This is the first official major release of OGGM. It is concomitant to the submission of a manuscript to Geoscientific Model Development (GMD).

This marks the stabilization of the codebase (hopefully) and implies that future changes will have to be documented carefully to ensure traceability.

New contributors to the project:

- **Anouk Vlug** (PhD student, University of Bremen), added the CESM climate data tools.
- **Anton Butenko** (PhD student, University of Bremen), developed the downstream bedshape algorithm

- **Beatriz Recinos** (PhD student, University of Bremen), participated to the development of the calving parametrization
- **Julia Eis** (PhD student, University of Bremen), developed the glacier partitioning algorithm
- **Schmitty Smith** (PhD student, Northand College, Wisconsin US), added optional parameters to the mass-balance models

v0.1.1 (16 February 2017)

Minor release: changes in ITMIX to handle the synthetic glacier cases.

It was tagged only recently for long term documentation purposes and storage on Zenodo.

v0.1 (29 March 2016)

Initial release, used to prepare the data submitted to ITMIX (see here).

This release is the result of several months of development (outside of GitHub for a large part). Several people have contributed to this release:

- **Michael Adamer** (intern, UIBK), participated to the development of the centerline determination algorithm (2014)
- **Kévin Fourteau** (intern, UIBK, ENS Cachan), participated to the development of the inversion and the flowline modelling algorithms (2014-2015)
- **Alexander H. Jarosch** (Associate Professor, University of Iceland), developed the MUSCL-SuperBee model (PR23)
- **Johannes Landmann** (intern, UIBK), participated to the links between databases project (2015)
- **Ben Marzeion** (project leader, University of Bremen)
- **Fabien Maussion** (project leader, UIBK)
- **Felix Oesterle** (Post-Doc, UIBK), develops OGGR and provided the AWS deployment script (PR25)
- **Timo Rothenpieler** (programmer, University of Bremen), participated to the OGGM deployment script (e.g. PR34, PR48), and developed OGGM installation tools
- **Christian Wild** (master student, UIBK), participated to the development of the centerline determination algorithm (2014)

1.3 Contributing

- *Contributing to OGGM*

1.3.1 Contributing to OGGM

All contributions, bug reports, bug fixes, documentation improvements, enhancements and ideas are welcome! OGGM is still in an early development phase, so most things are not written in stone and can probably be enhanced/corrected/meliorated by anyone!

You can report issues or discuss OGGM on the issue tracker.

Copyright note: this page is a shorter version of the excellent pandas documentation.

Working with the code

Before you contribute, you will need to learn how to work with GitHub and the OGGM code base.

Version control, Git, and GitHub

The code is hosted on GitHub. To contribute you will need to sign up for a free GitHub account. We use Git for version control to allow many people to work together on the project.

Some great resources for learning Git:

- the GitHub help pages.
- the NumPy's documentation.
- Matthew Brett's Pydagogue.

Getting started with Git

GitHub has instructions for installing git, setting up your SSH key, and configuring git. All these steps need to be completed before you can work seamlessly between your local repository and GitHub.

Forking

You will need your own fork to work on the code. Go to the OGGM project page and hit the `Fork` button. You will want to clone your fork to your machine:

```
git clone git@github.com:your-user-name/oggm.git oggm-yourname
cd oggm-yourname
git remote add upstream git://github.com/OGGM/oggm.git
```

This creates the directory *oggm-yourname* and connects your repository to the upstream (main project) oggm repository.

Creating a branch

You want your master branch to reflect only production-ready code, so create a feature branch for making your changes. For example:

```
git branch shiny-new-feature
git checkout shiny-new-feature
```

The above can be simplified to:

```
git checkout -b shiny-new-feature
```

This changes your working directory to the shiny-new-feature branch. Try to keep any changes in this branch specific to one bug or feature. You can have many shiny-new-features and switch in between them using the git checkout command.

To update this branch, you need to retrieve the changes from the master branch:

```
git fetch upstream
git rebase upstream/master
```

This will replay your commits on top of the lastest oggm git master. If this leads to merge conflicts, you must resolve these before submitting your pull request. If you have uncommitted changes, you will need to `stash` them prior to updating. This will effectively store your changes and they can be reapplied after updating.

Creating a development environment

An easy way to create a OGGM development environment is explained in *Installing OGGM*.

Contributing to the code base

Code standards

OGGM uses the PEP8 standard, as far as possible. There are several tools to ensure you abide by this standard, and some IDE (for example PyCharm) will warn you if you don't follow PEP8.

Test-driven development/code writing

OGGM is serious about testing and strongly encourages contributors to embrace test-driven development (TDD). Like many packages, OGGM uses the pytest testing system and the convenient extensions in numpy.testing.

All tests should go into the `tests` subdirectory of OGGM. This folder contains many current examples of tests, and we suggest looking to these for inspiration.

Running the test suite

The tests can then be run directly inside your Git clone by typing:

```
pytest .
```

The tests can run for several minutes. If everything worked fine, you should see something like:

```
==== test session starts ====
platform linux -- Python 3.4.3, pytest-3.0.5, py-1.4.31, pluggy-0.4.0
rootdir:
plugins:
collected 92 items

oggm/tests/test_graphics.py ..............
oggm/tests/test_models.py .........s....sssssssssssssssss
oggm/tests/test_prepro.py ...s.................s.s...
oggm/tests/test_utils.py ...sss..ss.sssss.
oggm/tests/test_workflow.py ssss

===== 57 passed, 35 skipped in 102.50 seconds ====
```

You can safely ignore deprecation warnings and other DLL messages as long as the tests end with `OK`.

Often it is worth running only a subset of tests first around your changes before running the entire suite. This is done using one of the following constructs:

```
pytest oggm/tests/[test-module].py
pytest oggm/tests/[test-module].py:[TestClass]
pytest oggm/tests/[test-module].py:[TestClass].[test_method]
```

Contributing to the documentation

Contributing to the documentation is of huge value. Something as simple as rewriting small passages for clarity is a simple but effective way to contribute.

About the documentation

The documentation is written in **reStructuredText**, which is almost like writing in plain English, and built using Sphinx. The Sphinx Documentation has an excellent introduction to reST. Review the Sphinx docs to perform more complex changes to the documentation as well.

Some other important things to know about the docs:

- The OGGM documentation consists of two parts: the docstrings in the code itself and the docs in this folder `oggm/docs/`.

  The docstrings *should* provide a clear explanation of the usage of the individual functions (currently this is not the case everywhere, ufortunately), while the documentation in this folder consists of tutorial-like overviews per topic together with some other information (what's new, installation, etc).

- The docstrings follow the **Numpy Docstring Standard**, which is used widely in the Scientific Python community. This standard specifies the format of the different sections of the docstring. See this document for a detailed explanation, or look at some of the existing functions to extend it in a similar manner.
- Some pages make use of the ipython directive sphinx extension. This directive lets you put code in the documentation which will be run during the doc build.

How to build the documentation

Requirements

There are some extra requirements to build the docs: you will need to have `sphinx`, `numpydoc` and `ipython` installed.

If you have a conda environment named `oggm-env`, you can install the extra requirements with:

```
conda install -n oggm-env sphinx ipython numpydoc
```

Building the documentation

So how do you build the docs? Navigate to your local `oggm/docs/` directory in the console and run:

```
make html
```

Then you can find the HTML output in the folder `oggm/docs/_build/html/`.

The first time you build the docs, it will take quite a while because it has to run all the code examples and build all the generated docstring pages. In subsequent evocations, sphinx will try to only build the pages that have been modified.

If you want to do a full clean build, do:

```
make clean
make html
```

Open the following file in a web browser to see the full documentation you just built:

```
oggm/docs/_build/html/index.html
```

And you'll have the satisfaction of seeing your new and improved documentation!

Contributing your changes

Committing your code

Keep style fixes to a separate commit to make your pull request more readable.

Once you've made changes, you can see them by typing:

```
git status
```

If you have created a new file, it is not being tracked by git. Add it by typing:

```
git add path/to/file-to-be-added.py
```

Doing 'git status' again should give something like:

```
# On branch shiny-new-feature
#
#       modified:   /relative/path/to/file-you-added.py
#
```

Finally, commit your changes to your local repository with an explanatory message:

```
git commit -a -m 'added shiny feature'
```

You can make as many commits as you want before submitting your changes to OGGM, but it is a good idea to keep your commits organised.

Pushing your changes

When you want your changes to appear publicly on your GitHub page, push your forked feature branch's commits:

```
git push origin shiny-new-feature
```

Here `origin` is the default name given to your remote repository on GitHub. You can see the remote repositories:

```
git remote -v
```

If you added the upstream repository as described above you will see something like:

```
origin  git@github.com:yourname/oggm.git (fetch)
origin  git@github.com:yourname/oggm.git (push)
upstream        git://github.com/OGGM/oggm.git (fetch)
upstream        git://github.com/OGGM/oggm.git (push)
```

Now your code is on GitHub, but it is not yet a part of the OGGM project. For that to happen, a pull request needs to be submitted on GitHub.

Review your code

When you're ready to ask for a code review, file a pull request. Before you do, once again make sure that you have followed all the guidelines outlined in this document regarding code style, tests, and documentation. You should also double check your branch changes against the branch it was based on:

1. Navigate to your repository on GitHub – https://github.com/your-user-name/oggm
2. Click on `Branches`
3. Click on the `Compare` button for your feature branch
4. Select the `base` and `compare` branches, if necessary. This will be `master` and `shiny-new-feature`, respectively.

Finally, make the pull request

If everything looks good, you are ready to make a pull request. A pull request is how code from a local repository becomes available to the GitHub community and can be looked at and eventually merged into the master version. This pull request and its associated changes will eventually be committed to the master branch and available in the next release. To submit a pull request:

1. Navigate to your repository on GitHub
2. Click on the `Pull Request` button
3. You can then click on `Commits` and `Files Changed` to make sure everything looks okay one last time
4. Write a description of your changes in the `Preview Discussion` tab
5. Click `Send Pull Request`.

This request then goes to the repository maintainers, and they will review the code. If you need to make more changes, you can make them in your branch, push them to GitHub, and the pull request will be automatically updated. Pushing them to GitHub again is done by:

```
git push -f origin shiny-new-feature
```

This will automatically update your pull request with the latest code and restart the Travis-CI tests.

Delete your merged branch (optional)

Once your feature branch is accepted into upstream, you'll probably want to get rid of the branch. First, merge upstream master into your branch so git knows it is safe to delete your branch:

```
git fetch upstream
git checkout master
git merge upstream/master
```

Then you can just do:

```
git branch -d shiny-new-feature
```

Make sure you use a lower-case `-d`, or else git won't warn you if your feature branch has not actually been merged.

The branch will still exist on GitHub, so to delete it there do:

```
git push origin --delete shiny-new-feature
```

CHAPTER 2

Get in touch

- View the source code on GitHub.
- Report bugs or share your ideas on the issue tracker.
- Improve the model by submitting a pull request.
- Or you can always send us an e-mail the good old way.

CHAPTER 3

License and citation

OGGM is available under the open source GNU GPLv3 license.

OGGM is free software. This implies that you are free to use the model and copy or modify its code at your wish, under certain conditions:

1. When using this software, please acknowledge the original authors of this contribution. Currently, we recommend to use the Zenodo citation for this purpose.

   An example BibTeX entry:

```
@misc{OGGM_v0.1.1,
  author       = {Fabien Maussion and
                  Timo Rothenpieler and
                  Ben Marzeion and
                  Johannes Landmann and
                  Felix Oesterle and
                  Alexander Jarosch and
                  Beatriz Recinos and
                  Anouk Vlug},
  title        = {OGGM/oggm: v0.1.1},
  month        = feb,
  year         = 2017,
  doi          = {10.5281/zenodo.292630},
  url          = {https://doi.org/10.5281/zenodo.292630}
}
```

2. Your modifications to the code belong to you, but if you decide to share these modifications with others you'll have to do so under the same license as OGGM (the GNU General Public License as published by the Free Software Foundation).

See the wikipedia page about GPL and the OGGM license for more information.

CHAPTER 4

About

Status Experimental - in development

License GNU GPLv3

Authors See the *Version history* for a list of all contributors.

Bibliography

[Cuffey_Paterson_2010] Cuffey, K., and W. S. B. Paterson (2010). The Physics of Glaciers, ButterworthHeinemann, Oxford, U.K.

[Farinotti_etal_2009] Farinotti, D., Huss, M., Bauder, A., Funk, M., & Truffer, M. (2009). A method to estimate the ice volume and ice-thickness distribution of alpine glaciers. Journal of Glaciology, 55 (191), 422–430.

[Golledge_Levy_2011] Golledge, N. R., and Levy, R. H. (2011). Geometry and dynamics of an East Antarctic Ice Sheet outlet glacier, under past and present climates. Journal of Geophysical Research: Earth Surface, 116(3), 1–13.

[Jarosch_etal_2013] Jarosch, a. H., Schoof, C. G., & Anslow, F. S. (2013). Restoring mass conservation to shallow ice flow models over complex terrain. Cryosphere, 7(1), 229–240. http://doi.org/10.5194/tc-7-229-2013

[Oerlemans_1997] Oerlemans, J. (1997). A flowline model for Nigardsbreen, Norway: projection of future glacier length based on dynamic calibration with the historic record. Journal of Glaciology, 24, 382–389.

[Cuffey_Paterson_2010] Cuffey, K., and W. S. B. Paterson (2010). The Physics of Glaciers, ButterworthHeinemann, Oxford, U.K.

[Farinotti_etal_2009] Farinotti, D., Huss, M., Bauder, A., Funk, M., & Truffer, M. (2009). A method to estimate the ice volume and ice-thickness distribution of alpine glaciers. Journal of Glaciology, 55 (191), 422–430.

[Golledge_Levy_2011] Golledge, N. R., and Levy, R. H. (2011). Geometry and dynamics of an East Antarctic Ice Sheet outlet glacier, under past and present climates. Journal of Geophysical Research: Earth Surface, 116(3), 1–13.

[Jarosch_etal_2013] Jarosch, a. H., Schoof, C. G., & Anslow, F. S. (2013). Restoring mass conservation to shallow ice flow models over complex terrain. Cryosphere, 7(1), 229–240. http://doi.org/10.5194/tc-7-229-2013

[Oerlemans_1997] Oerlemans, J. (1997). A flowline model for Nigardsbreen, Norway: projection of future glacier length based on dynamic calibration with the historic record. Journal of Glaciology, 24, 382–389.

Index

Symbols

A

C

D

E

F

G

I

L

M

O

P

R

T

V