The tip of the iceberg

MOD510: Mandatory project #1

Deadline: 12 September 2021 (23:59)

Aug 30, 2021

Learning objectives. By completing this project, the student will:

- Get experience in structuring and writing a report.
- Write new functions and classes in Python.
- Explore numerical round-off and truncation errors.
- Be introduced to useful Python libraries for scientific computing.

Abstract

Because computers have finite memory, numerical errors must always be taken into account when doing calculations, especially when working with floating-point numbers [2]. In the first parts of this project we investigate round-off errors and truncation errors using the Python programming language, and we discuss how different implementation strategies affect code efficiency and code clarity. In particular, we show how coding with classes can simplify the implementation of numerical algorithms. Finally, we highlight the power of available Python libraries to quickly visualize and manipulate large data sets. To this end, we look at how ice melting in Antarctica could affect sea level rise [1].

1 Before you start

In the last part of this project we are going to import ice data for Antarctica using the `rockhound` library. On some platforms it is a challenge to install this library, however if you use *environments* in conda it should be much easier, see Appendix A for more information. We recommend you to do the following:

1. On the windows start menu, open Anaconda prompt, (located in the Anaconda folder)

2. Create a new conda environment by running the commands below (in the Anaconda prompt)

```
conda update conda
```

It probably goes without saying, but this command updates the conda package. The `rockhound` library and `cmocean` library are not in the default channel so you need to add `conda-forge`

```
conda config --add channels conda-forge
```

Now, we can create an environment `project1` with the following packages

```
conda create -n project1 python matplotlib numpy pandas rockhound cmocean pip spyder jupyter
```

Entering all the package names when creating the new environment forces conda to check for dependencies, and makes sure that there are no conflicts. Do

```
conda activate project1
```

Now you can do e.g. `jupyter notebook` and continue to work on the project. Later you can add packages using `conda install <package-name>`.

pip or conda.

Always try `conda install <package-name>` first, even if you find on the web that people have done `pip install <package-name>`. If `conda install` fails you can try `pip install`.

2 Exercise 1: Finite-precision arithmetic

Part 1. Run the following code snippet:

```
import sys
sys.float_info
```

- Explain the meaning of the printed-out numbers using the IEEE Standard for floating-point arithmetic.

Part 2.

- Derive the following values yourself: `max`, `min`, `epsilon`.

Part 3. In Python, typing `0.1+0.2` does not (typically) produce the same output as `0.3`.

- Why not?

Part 4.

- Should you use the `==`-operator to test whether two floating-point numbers are equal?
- Why / why not?

3 Exercise 2: Need for speed? (NumPy)

The purpose of this exercise is to learn a little bit about NumPy, which is an incredibly useful Python library. A major reason for its popularity is efficiency: doing computations with NumPy arrays (objects of the type `ndarray`) instead of using native Python lists can, by itself, speed up a program by several orders of magnitude! The mechanism for speed-up is vectorized computation.

Vectorized functions.

Using NumPy arrays allows you to create vectorized functions; functions that operate on a whole array at once, rather than looping over the elements one-by-one inside a custom written loop.

The way vectorization works behind the scenes is still via loops (optimized, pre-compiled C code), but as Python programmer you do not need to worry about the details.

Part 1. The following code block gives an example of a vectorized function:

```
x = np.linspace(0, 1, 10)
fx = np.exp(-x)  # apply f(t)=exp(-t) to each element in the array x
```

Notice the usage of `np.exp` instead of using the exponential function provided in the built-in math library; this is an example of a universal function.

- What happens if you change `x` to be a list?
- If `x` is a list, how can you modify the second line above to calculate `fx`?

Tip: Use either a custom loop, or a list comprehension.

Part 2. As already hinted at, the NumPy library comes with a plethora of useful features and functions. The code snippets below show some examples:

```
np.zeros(20)
```

```
np.ones(20)
```

```
np.linspace(0, 10, 11)
```

```
np.linspace(0, 10, 11, endpoint=False)
```

```
vector = np.arange(5) + 1
2*vector
```

- Explain what each line of code does.
- How would you produce the same output using native Python lists?

Part 3. Frequently you will want to extract a subset of values from an array based on some kind of criterion. For example, you might want to count the number of non-zero numbers, or identify all values exceeding a certain threshold. With NumPy, suchs tasks are easily achieved using boolean masking, e.g.:

```
array_of_numbers = np.array([4, 8, 15, 16, 23, 42])
nnz = np.count_nonzero(array_of_numbers)
print(f'There are {nnz} non-zero numbers in the array.')
is_even = (array_of_numbers % 2 == 0)
is_greater_than_17 = (array_of_numbers > 17)
is_even_and_greater_than_17 = is_even & is_greater_than_17
```

However, the following code line does not execute:

```
is_even_and_greater_than_17 = is_even and is_greater_than_17
```

- Why not?

Part 4. The function np.where can also be used to select elements from an array.

- Explain the output of the following two lines of code:

```
np.where(array_of_numbers > 17)[0]
```

```
np.where(array_of_numbers > 17, 1, 0)
```

4 Exercise 3, Part I: Finite Differences (FD) with Functions

In scientific computing one often needs to calculate derivatives of functions. However, exact formulas may not be available, in which case numerical estimates are needed. To evaluate the correctness of our programmed numerical methods, it is still wise to choose test functions where the derivative is known beforehand.

In this exercise, we consider a function that is relevant for describing wave phenomenona:

$$f(x) = \sin(bx) \cdot e^{-ax^2} \tag{1}$$

One way to implement the function in Python is:

```
def f(x, a=0.1, b=10):
    return np.sin(b*x)*np.exp(-a*x*x)
```

We have chosen to define `a` and `b` as *default arguments*, which allows us to evaluate the function at $x = 1$ by simply typing `f(1)`; this is equivalent to the command `f(1, 0.1, 10)`. If you want to change the `b` parameter, you can do, e.g., `f(1, b=2)`. Note also that the function works both when `x` is a single number *and* when it is a Numpy array. This is because we use the Numpy versions of the sine (`np.sin`) and exponential (`np.exp`) functions.

Python functions are first-class!

An important feature of Python is that functions are first-class objects, meaning that you can assign them to variables, you can store them inside various containers and data structures, they can be passed as input arguments to other functions, and they may be return values of other functions.

We will exploit this property of Python several times during this project.

Part 1. It is always a good idea to start by visualizing the function in a plot.

- Make a Python function that plots $f(x)$ from equation (1) over an arbitrary closed interval.
- Use the function to plot $f(x)$ in the range $[-10, 10]$. Try to make your figure similar to the one shown in figure 1

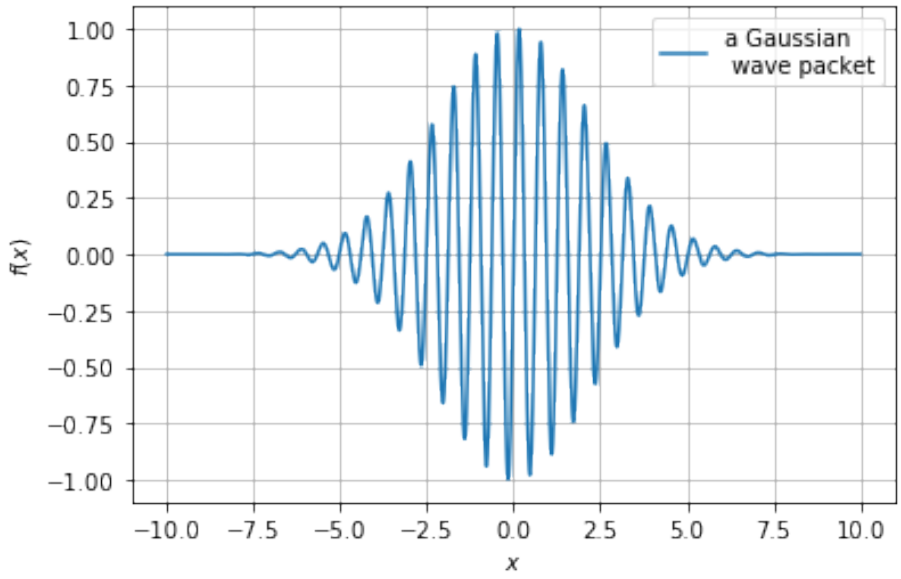

Figure 1: A plot of $f(x)$.

Part 2. The analytical derivative of $f(x)$ is

$$f'(x) = b\cos(bx) \cdot e^{-ax^2} - 2ax\sin(bx) \cdot e^{-ax^2} \tag{2}$$

- Write a Python function that calculates the derivative defined in equation (2).

Part 3. Next, we are going to write a Python function that calculates the numerical derivative of an *arbitrary* single-variable function `f` at a point x using finite differences. If `f` only depends on x, this is extremely easy: we can simply take `f`, `x`, and `h` (step-size) as input arguments to our derivative function. Assuming it is called `calc_derivative`, it could work something like this:

```
df_x = calc_derivative(f, x=1.0, h=1.0e-3)
```

However, the function in equation (1) depends on two additional input parameters, `a` and `b`. We can of course add these two as extra arguments to the derivative function, but then we would lose generality, for not every function has the same two parameters. A way out of this dilemma could be to create a new function each time you choose values for `a` and `b`, e.g:

```
def g(x):
    """
    The function g(x) = sin(10x) * exp(-0.1x^2).
    """
    return f(x, 0.1, 10.0)
```

Another possibility is to use the the args mechanism, which lets you pass around a variable number of parameters to a function. An example of how this works is:

```
def calc_derivative(f, x, h, *args):
    return (f(x, *args) - f(x-h, *args))/h
```

- Write a Python function that calculates the derivative of an arbitrary function using the *forward difference* method (see section 1.3 in [3]).
- Apply your function to the particular case of equation (1) and $x = 1$. Use a suitable value of h, e.g. $h = 1e - 2$, and check that your estimate agrees reasonably well with the *analytical solution*.

Part 4.

- Write another Python function that calculates $f'(x)$ with the *central difference* method (see section 1.4 in [3]).

Part 5. We want to quantify the error in our numerical estimates. To this end, we start by choosing a point at which to evaluate the numerical derivative of the function in equation (1), e.g., $x = 1$. Next:

- Make a figure in which you plot the *absolute error* versus step size, h.
- Vary the step size logarithmically between 10^{-16} and 1.
- The figure should include one error curve for the *forward difference* method, and another for the *central difference* methods.

Part 6.

- Comment on what you observe in the figure you made. When is the numerical error smallest, and why? Is it what you expect from a theoretical analysis using Taylor's formula?

5 Exercise 3, Part II: FD with Classes

Implementing numerical algorithms with free functions, as we did in the previous exercise, is perfectly fine, and you can complete MOD510 by only coding in this way. However, experience has taught us that it is easy to introduce unnecessary errors when using this approach. In many cases you are better off by also using classes, and maybe object-oriented design. In this exercise, you will get some practice in coding with classes. This knowledge will come in handy in later projects, and in any case it is a good tool to have in your programming toolkit.

Previously, we worked with a function having two input parameters, a and b. Implementing numerical algorithms using free functions was then simple. However, in a more complicated situation there could be dozens, or even hundreds, of parameters to keep track of. Most of these parameters might have fixed values, but frequently you will want to re-run a model with slightly different parameters than before. If you are not using classes, it is very easy to use the wrong parameters. This is especially true when working in a Jupyter notebook, since you might run code blocks in any order; if you forget to excute a cell that is responsible for updating one of your variables, your subsqeuent calculations will be wrong!

Key take-away: Classes provide encapsulation.

By wrapping parts of your code into classes, and particular realizations of classes (objects), you facilitate code re-use, and it can make your code easier to understand and work with, thus reducing the probability of introducing bugs which may be hard to track down.

5.1 A Crash Course on Classes

To get started, there are really only a couple of things you need to know. First, all of your classes should include a special function called `__init__`, in which you declare the variables (attributes) you wish an instance / object of the class to keep track of.

Second, inside the class all class attributes should be prefixed with `self`, followed by a dot. Third, functions defined inside a class should have `self` as the first function argument. All of this is best understood via an example:

```
class WavePacket:
    """
    A class representation of a wave packet-function.
    """
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def f(self, x):
        return np.sin(self.b*x)*np.exp(-self.a*x*x)
```

```
    def plot(self, min=-10, max=10, dx=0.01):
        """
        A simple plotting routine for plotting f(x) in some range.
        """
        x = np.arange(min, max, dx)
        y = self.f(x)
        fig = plt.figure()
        plt.plot(x, y)
        plt.grid()
```

Besides the initialization method and a function that calculates $f(x)$ from equation (1), the class includes a simple plotting routine. A major difference from before is the following: when our function $f(x)$ is defined inside a class, we do not have to pass around a and b as arguments to the function `f`. Instead, we simply access a and b from inside the class itself (using the `self`-prefix).

Below is an example of how to use the class:

```
# Create two WavePacket objects, having their own parameter values
WP1 = WavePacket(0.1, 2)  # a=0.1, b=2
WP2 = WavePacket(0.1, 10)  # a = 0.1, b=10

# Evaluate the two functions at a specific point
x = 1
print(WP1.f(x))
print(WP2.f(x))

# Plot the two functions
WP1.plot()
WP2.plot()
```

Although we had to write slightly more code, we hope you appreciate how easy this makes running parallel simulations with different parameters!

Part 1.

- Add another function (instance method) to the class above called `df_fe`. This function should return the forward difference approximation to the derivative of the function `f` at a point `x` given as input. Include the step-size `h` as input argument (i.e., from the outside the class).
- Add a second function, `df_cd`, which calculates the central difference approximation.

Part 2.

- Make a third function inside the class that plots the absolute error versus step size for the two finite difference approximations. Use step sizes in the range from 10^{-16} to 1 (with logarithmic spacing).

Hint: You should re-use the first two functions when making the third one.

6 Exercise 4: A song of ice and fire?

There is currently a great deal of concern about global warming. Some critical issues are whether we are more likely to observe extreme local temperatures, increased frequencies of natural disasters like forest fires and droughts, and if there are "tipping points" in the climate system that are, at least on the human timescale, irreversible [4]. One particular question to ask is: How much ice is likely to melt? And, what would be the consequence of ice melting for sea level rise (SLR)?

Since most of the ice on Planet Earth is located in Antarctica, substantial effort has been spent in mapping the ice and the bedrock of this continent. Most of the data is freely available, and we can use them to investigate different scenarios.

Preparations for exercise: Install helpful Python packages.

We will work with the *bedmap2* dataset [1]. The rockhound library can be used to load the data. As an aid to plotting, we will also employ color maps from the cmocean library [6]. Therefore, you should start by making sure that you have both cmocean and rockhound installed in the Python version that you use.

To avoid potential problems down the road, it is recommended to work with different *Python environments* on your computer, rather than a single global installation. To this end, you can use either `venv`, or (recommended for this course) the conda package manager. In the following you should use the environment you created at the beginning of the project.

6.1 Theory

To calculate SLR, we need to know not only how thick the ice is, but also its elevation above the bedrock. In this exercise you will see how we can use Python, together with available data and libraries, to do quite advanced calculations.

Melting of an iceberg. Let us start by deriving a result that you may have seen before. We shall consider an iceberg that is floating and which is not impacted by any other forces than gravity. The total volume of ice in figure 2 is $V_f + V_{\mathrm{H2O}}^{\mathrm{disp}}$, where V_f is the ice volume floating above the sea, and $V_{\mathrm{H2O}}^{\mathrm{disp}}$ is the volume of displaced sea water (submerged ice). According to Newton's 2nd law, the total weight of ice is therefore

$$W = m_{ice}g = \rho_{ice}V_{ice}g = \rho_{ice}(V_f + V_{\mathrm{H_2O}}^{\mathrm{disp}})g\,, \tag{3}$$

where ρ_{ice} is the density (mass divided by volume), and $g = 9.81\mathrm{m/s}^2$ the gravitational constant. On the other hand, the Archimedean principle tells us that buoyancy is proportional to *the mass of displaced water*, hence another

expression for the same weight is

$$W = \rho_w V_{\mathrm{H_2O}}^{\mathrm{disp}} g \,, \tag{4}$$

where ρ_w is the density of the surrounding sea-water. As the ice melts, it must be turned into an equal mass of liquid ice-water:

$$\begin{aligned} m_{\mathrm{ice}} &= m_{\text{ice water}} \\ \rho_{\mathrm{ice}} V_{\mathrm{ice}} &= \rho_{\text{ice water}} V_{\text{ice water}} \,. \end{aligned} \tag{5}$$

By combining the above equations, we therefore get:

$$V_{\mathrm{H_2O}}^{\mathrm{disp}} = \frac{\rho_{\text{ice water}}}{\rho_w} V_{\text{ice water}}. \tag{6}$$

Therefore, the net contribution to SLR is captured by the volume change

$$\Delta V \equiv V_{\text{ice water}} - V_{\mathrm{H_2O}}^{\mathrm{disp}} = \left(1 - \frac{\rho_{\text{ice water}}}{\rho_w}\right) V_{\text{ice water}}. \tag{7}$$

If the melted ice has the same density as seawater, it follows that there is *no increase in sea level*. On the other hand, if the density is lower than that of seawater, there is a contribution. Typically, the melting of ice dilutes the salinity of the ocean, which leads to a small increase in sea level [5].

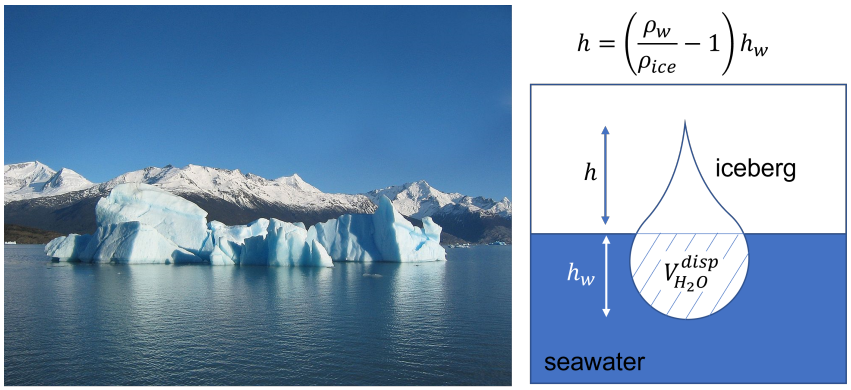

Figure 2: A schematic illustration of an iceberg. $V_{\mathrm{H_2O}}^{\mathrm{disp}}$ is the volume of water displaced by the iceberg.

Freeboard. The vast majority of ice in Antarctica is not freely floating. In addition to the salinity effect described above, the main contribution to SLR is from ice located above *hydrostatic equilibrium*. It is useful to introduce the concept of the *freeboard*, which is the height above seawater of a floating iceberg. Again referring to figure 2, the freeboard height, h, is

$$h = \left(\frac{\rho_w}{\rho_{\mathrm{ice}}} - 1\right) h_w \,. \tag{8}$$

For the *bedmap2* dataset, the freeboard level can be computed from (figure 3)

$$h = -(\text{surface} - \text{thickness}) \cdot \left(\frac{\rho_w}{\rho_{ice}} - 1 \right) \tag{9}$$

Note that this formula assumes that the bedrock is below sea-level.

Part 1. The code below is taken from the rockound library documentation:

```
bedmap = rh.fetch_bedmap2(datasets=["thickness","surface","bed"])
plt.figure(figsize=(8, 7))
ax = plt.subplot(111)
bedmap.surface.plot.pcolormesh(ax=ax,cmap=cmocean.cm.ice,
                        cbar_kwargs=dict(pad=0.01, aspect=30))
plt.title("Bedmap2 Antarctica")
plt.tight_layout()
plt.show()
```

- Run the code and reproduce figure 3.

This may take quite some time, so it is recommended that you do not execute the cell generating the plot more often than you have to.

Part 2. The Thwaites glacier, named after Fredrik T. Thwaites, is of particular interest. Also referred to as the Doomsday glacier, it is a fast moving glacier (up to 4 km/year) at the coast of Antarctica, roughly 120 km wide. It is called the Doomsday glacier because it is believed that it may trigger a collapse of west Antarctica. We can use the bedmap2 dataset to view a cross section of the glacier, $y = -0.5 \cdot 10^6$ and $x \in [-1.6 \cdot 10^6, -1.35 \cdot 10^6]$:

```
# Extract cross section using the original coordinates
bed1d = bedmap.sel(y=-0.5e6, x=slice(-1.6e6,-1.35e6))

# Add a second x-coordinate to make prettier plots
# (shift x-axis to start at zero, and convert from m to km)
bed1d = bed1d.assign_coords({"x2":((bed1d.x+1.6e6)/1e3)})
```

It is now possible to plot the values in the `bed1d` data array by simply typing

```
bed1d.surface.plot(x='x2')
```

A note concerning the xarray library.

The *bedmap2* data is stored in data structures provided by the xarray package, which is built on top of NumPy and pandas. While xarray allows you to apply plot commands directly to Datasets and DataArrays, it might be easier to use the `matplotlib.pyplot` module manually. To this end, you might also want to extract the underlying NumPy arrays containing

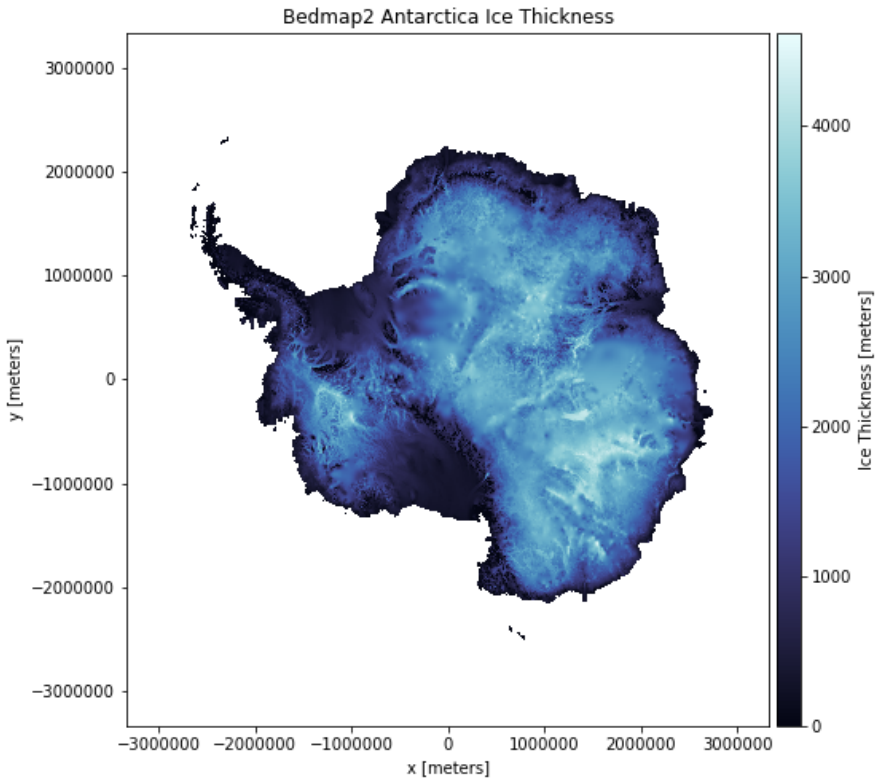

Figure 3: Visualization of the ice thickness in Antarctica.

the data you are interested in. This can easily be done as in the following example: `surface_elevation_values = bed1d['surface'].values`.

- Use equation (9) to calculate the freeboard level.
- Make a cross section plot of the ice thickness, bed rock, and freeboard level, similar to (the right) figure (4)
- Explain what you see. Is the shape of the bedrock important for sea level rise?

Part 3.

- Use the whole data set to estimate the total SLR if all the ice of Antarctica melts.

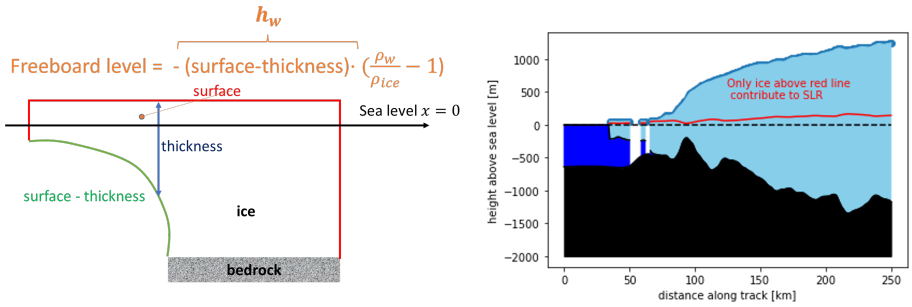

Figure 4: (left) How to calculate the freeboard level. (right) Cross section plot of Thwaites glacier.

Hints to get you started:

1. Start by calculating, for each grid cell in the dataset, the height of ice that can contribute to SLR.
2. Multiply the height with the area of each cell (1000×1000 meters) in order to estimate the volume of ice.

Part 4.

- How can we use the bedmap data to improve decision making?
- What are some limitations in your estimated calculation of sea level rise?

7 Appendix A: Package management with `conda`

Package installation in Python can be quite tricky, and it is easy to make mistakes, especially if the packages you wish to use have many dependencies. A good way of avoiding unnecessary installation problems is to work with *virtual environments*; isolated installations of Python on your computer. This allows you to switch between different Python versions depending on the needs of a particular project. We prefer to use the `conda` package manager that follows along with the Anaconda Python distribution.

7.1 Getting `conda` up and running

It is possible to use a graphical user interface (GUI) to work with `conda`, but we will work exclusively from the command line. On Windows, this will probably require some additional setup, to make the relevant file paths available to `cmd.exe`. Alternatively, you can use the Anaconda Prompt that comes with the Anaconda distribution. To get started, it is recommended to check out the

official user guide, as well as cheat sheets with an overview of the most frequently used commands.

Assuming that you have `conda` installed and available to you from the terminal, typing the following command should report back the version number of `conda` on your system:

```
conda --version
```

If everything works, you will get something like this as output:

```
conda 4.10.3
```

You can list all of your available conda environments by typing:

```
conda info --envs
```

A star asterisk points to the currently active environment. For a fresh Anaconda installation, you will only have one, the base environment. You can find out exactly which packages that belongs to this environment by writing

```
conda list
```

7.2 Creating virtual environments

It is now very easy to create your own, custom Python environment, e.g.:

```
conda create --name MyExampleEnv python=3.9 numpy scipy matplotlib
```

As you might have guessed, we just created an environment with the name MyExampleEnv, along the way installing version `3.9` of `python.exe` into it, as well as several libraries we wish to use. This will require you to answer a query on the command line, after which the installation should commence.

We can activate the environment as follows:

```
conda activate MyExampleEnv
```

Additional packages can be installed later as needed, you just have to remember to activate the environment first (if it has not been done already), e.g.:

```
conda activate MyExampleEnv
conda install pandas jupyter
```

You can verify that `pandas` and `jupyter` have been installed by again typing `conda list` in the terminal.

Many packages will be possible to install exactly as above; or possibly by adding a few additional steps. For example, the `cmocean` and `rockhound`. packages can be installed as follows:

```
conda activate MyExampleEnv
conda install -c conda-forge cmocean
conda install rockhound --channel conda-forge
```

As always, you should check the installation instructions provided together with the package!

7.3 Selecting the version of Python to use

Each conda environment comes with its own installation directory, and with its own Python executable. If you program in an integrated development environment (IDE), like Spyder or Visual Studio Code, you will typically be able to choose the Python version to use by changing the project preferences. On the command-line, you can switch between different environments via the `conda activate` command.

When working with Jupyter notebooks, it may require some additional effort to be able to use your own environments. By writing

```
conda activate MyExampleEnv
python -m ipykernel install --name MyExampleEnv
```

the `conda` environment MyExampleEnv will show up as a possible Python kernel for your Jupyter notebooks.

8 Appendix B: Some intermediate Python tips

If you want to make your code more elegant, it is very useful to learn about special functions besides `__init__`. When working with mathematical functions, the `__call__` function is particularly useful. Again, this is best understood via an example, a simple modification of the class we suggested in the main text:

```
class WavePacket:
    """
    An updated class representation of a wave packet-function.
    """
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self, x):
        return np.sin(self.b*x)*np.exp(-self.a*x*x)

    def plot(self, min=-10, max=10, dx=0.01):
        """
        A simple plotting routine for plotting f(x) in some range.
        """
        x = np.arange(min, max, dx)
        y = self.__call__(x)
        fig = plt.figure()
        plt.plot(x, y)
        plt.grid()
```

We have simply replaced the original function `f` with `__call__`. This allows us to treat objects of the class `WavePacket` as if they are ordinary functions:

```
WP1 = WavePacket(0.1, 2)
WP1(1.0)  # Evaluates the function at x=1.0
```

Similarly, the `__str__`-function allows you to send your custom-made objects to the `print` statement, e.g:

```
class WavePacket:

    ...

    def __str__(self):
        """
        :return: A string representation of the function
                 (NB: uses LaTeX).
        """
        return f'sin({self.b}x)' + r'$\cdot$' \
                                 + f'exp(-{self.a}'+ r'$x^2$)'
```

```
wave_func = WavePacket(0.1, 5.0)
print(wave_func)
```

9 Guidelines for project submission

You should bear the following points in mind when working on the project:

- Start your notebook by providing a short introduction in which you outline the nature of the problem(s) to be investigated.
- End your notebook with a brief summary of what you feel you learned from the project (if anything). Also, if you have any general comments or suggestions for what could be improved in future assignments, this is the place to do it.
- All code that you make use of should be present in the notebook, and it should ideally execute without any errors (especially run-time errors). If you are not able to fix everything before the deadline, you should give your best understanding of what is not working, and how you might go about fixing it.
- Avoid duplicating code! If you find yourself copying and pasting a lot of code, it is a strong indication that you should define reuseable functions and/or classes.
- If you use an algorithm that is not fully described in the assignment text, you should try to explain it in your own words. This also applies if the method is described elsewhere in the course material.

- In some cases it may suffice to explain your work via comments in the code itself, but other times you might want to include a more elaborate explanation in terms of, e.g., mathematics and/or pseudocode.
- In general, it is a good habit to comment your code (though it can be overdone).
- When working with approximate solutions to equations, it is very useful to check your results against known exact (analytical) solutions, should they be available.
- It is also a good test of a model implementation to study what happens at known 'edge cases'.
- Any figures you include should be easily understandable. You should label axes appropriately, and depending on the problem, include other legends etc. Also, you should discuss your figures in the main text.
- It is always good if you can reflect a little bit around *why* you see what you see.

References

[1] Peter Fretwell, Hamish D. Pritchard, David G. Vaughan, Jonathan L. Bamber, Nicholas E. Barrand, R. Bell, C. Bianchi, RG Bingham, Donald D. Blankenship, and G. Casassa. Bedmap2: Improved ice bed, surface and thickness datasets for antarctica. *The Cryosphere*, 7(1):375–393, 2013.

[2] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

[3] Aksel Hiorth. *Computational Engineering and Modeling*. https://github.com/ahiorth/CompEngineering, 2021.

[4] V. Masson-Delmotte, P. Zhai, A. Pirani, S. L. Connors, C. Pean, S. Berger, N. Caud, Y. Chen, L. Goldfarb, M. I. Gomis, M. Huang, K. Leitzell, E. Lonnoy, J. B. R. Matthews, T. K. Maycock, T. Waterfield, O. Yelekci, R. Yu, and B. Zhou (eds.). Climate Change 2021: the Physical Science Basis. Contribution of Working Group I to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change. Technical report, Cambridge University Press. In Press., August 2021.

[5] Peter D. Noerdlinger and Kay R. Brower. The melting of floating ice raises the ocean level. *Geophysical Journal International*, 170(1):145–150, 2007.

[6] Kristen M. Thyng, Chad A. Greene, Robert D. Hetland, Heather M. Zimmerle, and Steven F. DiMarco. True colors of oceanography: Guidelines for effective and accurate colormap selection. *Oceanography*, 29(3):9–13, 2016.