Pytesimal

Release 2.0.0

M. Murphy Quinlan, A.M. Walker, P. Selves, L.S.E. Teggin

Jul 15, 2021

CONTENTS:

- Constant or variable material properties
- Choose to return compressed `.npz` NumPy arrays of temperature and cooling rates through time and radius
- Plot temperature or cooling rate heatmaps
- Return timing of core solidification, and depth and timing of meteorite formation
- Return a parameter `.json` file with details of input parameters and results

CHAPTER

ONE

INSTALLATION

This software relies on python (version 3.6 and up) and various other python packages. Examples are distributed as Jupyter notebooks, which need Jupyter and Matplotlib to run. The software and its dependencies are best installed in a virtual environment of your choice. Installation and management of all these dependencies in an isolated conda environment can be done by running:

```
conda create -n=pytesimal python=3.8 jupyter
conda activate pytesimal
pip install pytesimal
```

1.1 Installation for development

The package can be downloaded and installed directly from Github for the most recent version. The software and its dependencies are best installed in a virtual environment of your choice. Download of the software and creation of an isolated conda environment can be done by running:

```
git clone https://github.com/murphyqm/pytesimal.git
cd pytesimal
conda create -n=pytesimal python=3.8 jupyter
conda activate pytesimal
pip install -e .
```

The `-e` flag installs the package in editable mode so that any changes to modules can be carried through. Examples can be downloaded from the gallery here.

1.2 Installation in Google Colab

Pytesimal can also be installed and used in Google's hosted Jupyter notebook service, Colaboratory. In a new notebook, install the package:

```
!pip install pytesimal
```

You can also upload one of our examples and run it, after adding a cell with the installation command above to the start of the notebook.

CHAPTER

TWO

QUICK START

To run a case with parameters loaded from file:

```
import pytesimal.load_plot_save
import pytesimal.quick_workflow
```

Define a filepath and filename for the parameter file:

```
folderpath = 'path/to/the/example/'
filename = 'example_parameters'
filepath = f'{folderpath}{filename}.txt'
```

Save a default parameters file to the filepath:

```
pytesimal.load_plot_save.make_default_param_file(filepath)
```

You can open this `json` file in a text editor and change the default values. Once you've edited and saved it, use it as an input file for a model run:

```
pytesimal.quick_workflow.workflow(filename, folderpath)
```

Let your planetesimal evolve! This will take a minute or so to run. Once it has done so, you can load the results (from the folder you specified in the parameters file):

```
filepath = 'results_folder/example_parameters_results.npz'
(temperatures,
 coretemp,
 dT_by_dt,
 dT_by_dt_core) = pytesimal.load_plot_save.read_datafile(filepath)
```

Then you can plot heatmaps of the temperatures and cooling rates within the planetesimal:

```
# Specify a figure width and height:
fig_w = 6
fig_h = 9

pytesimal.load_plot_save.two_in_one(
fig_w,
fig_h,
temperatures,
coretemp,
```

(continues on next page)

(continued from previous page)

```
dT_by_dt,
dT_by_dt_core)
```

See the Jupyter notebooks hosted on Binder for live working examples, or download the example scripts provided.

CHAPTER

THREE

PARAMETER FILES

The `pytesimal.load_plot_save` module contains a number of functions to help you build, load and save parameter files. These files allow you to set and record the values of different variables, making reproducible research easier. This section gives an overview of the different inputs and how to use these functions, while documentation for each of the individual functions is available in the API documentation section.

The `make_default_param_file()` function quickly generates a `json` format file loaded with a set of default variable values (that recreate the constant properties result from Murphy Quinlan et al., 2021):

```
folder = 'path/to/folder'
filename = 'example_param_file.txt'
filepath=f'{folder}/{filename}'

check_folder_exists(folder)
make_default_param_file(filepath=filepath)
```

The `check_folder_exists()` function does what it says on the tin, then will create the folder if one does not exist. The `filepath` needs to include the absolute path to the folder, as wel as the filename including `.txt` file extension. This parameters file in `json` format can then be opened, edited, renamed or moved, and loaded in to set parameter values for a model run.

The file content looks like a Python dictionary:

```
{
"run_ID": "example_default",
"folder": "example_default",
"timestep": 100000000000.0,
"r_planet": 250000.0,
"core_size_factor": 0.5,
"reg_fraction": 0.032,
"max_time": 400,
"temp_core_melting": 1200.0,
"mantle_heat_cap_value": 819.0,
"mantle_density_value": 3341.0,
"mantle_conductivity_value": 3.0,
"core_cp": 850.0,
"core_density": 7800.0,
"temp_init": 1600.0,
"temp_surface": 250.0,
"core_temp_init": 1600.0,
"core_latent_heat": 270000.0,
"kappa_reg": 5e-08,
"dr": 1000.0,
```

(continues on next page)

(continued from previous page)

```
"cond_constant": "y",
"density_constant": "y",
"heat_cap_constant": "y"
}
```

A description of each of these parameters is given in the docstring of `save_params_and_results`, but we'll look at a few of them here.

- `run_ID`: this is a string identifier for your model run, for your reference (set it to something short and descriptive)
- `folder`: this should be the absolute path to the folder where you want the results to be saved (if you are saving results)
- `timestep`, `dr` and material properties: make sure you check that your combination of discretisation scheme and material properties is stable (see section below on stability); `tiemstep` is in s and `dr` is in m.
- `max_time`: in Myr, how long the model will run for.
- `cond_constant`, `density_constant`, `heat_cap_constant`: string values that define whether to use constant or temperature-dependent material properties. Feed these parameters in as arguments when instantiating the mantle properties.

Once you have edited/copied/renamed/moved this file as you wish, it can be loaded in using the following function call:

```
load_params_from_file(filepath)
```

Where filepath again must be the absolute path to the file, including the filename with `.txt` extension.

The `save_params_and_results` function should be called after you have run your model, to record the parameters used. This output parameter file is formatted so that it can be read as an input parameter file too, allowing model runs to be reproduced and rerun exactly. It adds a number of fields to the original input parameter file, in order to save results:

- `"core_begins_to_freeze": time_core_frozen / myr` - this takes the modelled start time of the period of core crystallisation (in seconds) and converts it to millions of years
- `"core finishes freezing": fully_frozen / myr` - saves the modelled end time of the period of core crystallisation and converts from seconds to millions of years
- `meteorite_results` - optional; this should be formatted as a dictionary, listing any timing or depth data that should be saved. Extra notes can also be added here.
- `latent_list_len` - optional; `len(latent)` should be passed in as an argument to record the length of the latent heat list (to later calculate core crystallisation timing). This is also an optional argument for `save_result_arrays`.

Important: note that depending on the analysis carried out, you may not yet have results for all of the above. The parameter file can still be saved by substituting an obvious placeholder string, `0`, or `None` for one of the above values. Make note of this within the `meteorite_results` or `run_ID` fields.

CHAPTER

FOUR

NUMERICAL STABILITY

You can check whether your choice of diffusivity, timestep and radial discretisation meet Von Neumann stability criteria using the functions provided in the `pytesimal.numerical_methods` module.

Thermal diffusivity can be calculated from the conductivity, heat capacity and density using `calculate_diffusivity`, and then tested using the `check_stability` function. You should use the maximum diffusivity of your system to find the most restrictive criteria, and use this to inform your choice of timestep.

Note that other instabilities may arise when defining custom functions for thermal properties; please regularly plot the output data to check for unexpected behaviour.

CHAPTER

FIVE

CONTRIBUTE

- Issue Tracker
- Source Code

If you are having issues, please let us know. You can email us at eememq@leeds.ac.uk

The project is licensed under the MIT license.

CHAPTER

SIX

SUPPORT

If you are having issues, please let us know. You can email us at eememq@leeds.ac.uk

CHAPTER

SEVEN

LICENSE

The project is licensed under the MIT license.

CHAPTER

EIGHT

EXAMPLES GALLERY

Here are some example scripts. Each example is downloadable as a Jupyter Notebook or as a Python file. To try out some of the examples without installing the package, have a look at the Jupyter Notebooks hosted on Binder (link above).

8.1 0. Introduction

Pytesimal is a finite difference code to perform numerical models of a conductively cooling planetesimal, both with constant and temperature-dependent properties.

In this example, we walk through the theoretical background of the model and explain step-by-step how the code works.

8.1.1 Model set-up

Pytesimal allows the modelling of a conductively cooling body, with the following different regions:

- An isothermal convecting core: this can be replaced with a more complex core model or can be switched off to make a core-less body.
- A discretised, conductive mantle: this is the region of focus in Pytesimal. The material properties for this region can be temperature-dependent or constant.
- A discretised megaregolith: this region is also conductively cooling; material properties can only be constant in this region (constant diffusivity). This region can be switched off.

These different configurations, along with different material properties, can be set up to replicate a wide variety of different planetesimal geometries.

In order to set up our model, we first import the required packages:

```
import pytesimal.setup_functions
import pytesimal.load_plot_save
import pytesimal.numerical_methods
import pytesimal.analysis
import pytesimal.core_function
import pytesimal.mantle_properties
```

One way of setting up a model run is to use a parameter file. The parameter file is essentially a dictionary holding values for different variables, including the planetesimal radius, core size and regolith thickness, material properties for the body, and values to define the numerical discretisation.

We can generate a default parameter file by calling the *make_default_param_file* function. This function can be called with a *filepath* argument, specifying where the file should be saved and what it should be called:

```
filepath = 'parameters.txt'
pytesimal.load_plot_save.make_default_param_file(filepath)
```

This provides a template file for you to edit to suit your specific model set up (see documentation on parameter files for more information on the content and layout of the parameter file). This file can be edited and then loaded in - in practise, you wouldn't do this all in one script like we have here - you would create and edit a parameter file (or copy and edit a pre-existing one), and then in a separate script, would load the parameter file and run the model.

As we're just going to use the default values from the parameter file, we'll just load it straight back in without editing it:

```
(run_ID, folder, timestep, r_planet, core_size_factor,
 reg_fraction, max_time, temp_core_melting, mantle_heatcap_value,
 mantle_density_value, mantle_cond_value, core_heatcap, core_density,
 temp_init, temp_surface, core_temp_init, core_latent_heat,
 kappa_reg, dr, cond_constant, density_constant,
 heat_cap_constant) = pytesimal.load_plot_save.load_params_from_file(filepath)
```

This big collection of parameters will be fed in to our model!

To make this example run a bit faster, we're going to change the timestep from 1×10^{11} to 2×10^{11}

```
timestep = 2e11

#
# In order to discretise the different regions and record temperatures
# at radii points at each timestep, we need to set up some arrays
# that match the geometry we've passed in using the parameter file.
# These arrays will be placeholders until the numerical method fills
# them with values:

(r_core,
 radii,
 core_radii,
 reg_thickness,
 where_regolith,
 times,
 mantle_temperature_array,
 core_temperature_array) = pytesimal.setup_functions.set_up(timestep,
                                                            r_planet,
                                                            core_size_factor,
                                                            reg_fraction,
                                                            max_time,
                                                            dr)
```

8.1.2 The planetesimal core

Before we set up the numerical scheme for the discretised regions of the planetesimal, we need to instantiate the core object. This will keep track of the temperature of the core as the model runs, cooling as heat is extracted across the core-mantle boundary. The heat extracted in one timestep (P_{CMB}) is:

$$P_{\mathrm{CMB}} = -A_{\mathrm{c}} k_{\mathrm{m}} \left. \frac{\partial T}{\partial r} \right|_{r=r_{\mathrm{c}}} \quad (8.1)$$

where A_{c} is the core surface area, r_{c} is the core radius, and k_{m} is the thermal conductivity at the base of the mantle or discretised region. The corresponding change in core boundary temperature ΔT is:

$$\Delta T = -\frac{P_{\mathrm{CMB}}}{\rho_{\mathrm{c}} C_{\mathrm{c}} V_{\mathrm{c}}} \delta t \quad (8.2)$$

where ρ_{c} and C_{c} are the density and heat capacity of the core, and V_{c} is the volume of the core.

Once the core reaches its freezing temperature, the temperature is pinned. Latent heat is extracted until the total latent heat associated with core crystallisation has been removed. We need to set up an empty list to keep track of this latent heat:

```
latent = []
```

This simple eutectic core model doesn't track inner core growth, but this is still a required argument to allow for future incorporation of more complex core objects:

```
core_values = pytesimal.core_function.IsothermalEutecticCore(initial_temperature=core_
↪temp_init,
                                                             melting_temperature=temp_
↪core_melting,
                                                             outer_r=r_core, inner_r=0,␣
↪rho=core_density,
                                                             cp=core_heatcap,
                                                             core_latent_heat=core_
↪latent_heat)
```

8.1.3 Conductive cooling

The conductively cooling regions in the planetesimal can be described in 1D by the heat equation in spherical geometry:

$$\frac{\partial T}{\partial t} \rho C = \frac{1}{r^2} \frac{\partial}{\partial r} \left(k r^2 \frac{\partial T}{\partial r} \right),$$

where T is temperature, t is time, ρ is density, C is heat capacity, k is thermal conductivity, and r is radius.

We need to define the thermal properties for this region (ρ, C, and k). In our parameter file, we've already defined these as constant in temperature and have listed values. We just need to pass those arguments to the *set_up_mantle_properties* function:

```
(mantle_conductivity, mantle_heatcap, mantle_density) = pytesimal.mantle_properties.set_
↪up_mantle_properties(
    cond_constant,
    density_constant,
    heat_cap_constant,
    mantle_density_value,
    mantle_heatcap_value,
    mantle_cond_value)
```

The conduction equation is solved numerically using an explicit finite difference scheme, Forward-Time Central-Space (FTCS). FTCS gives first-order convergence in time and second-order in space, and is conditionally stable when applied to the heat equation. We can quickly calculate the diffusivity in the mantle and then check we meet Von Neumann stability criteria for the mantle and the megaregolith:

```
mantle_diffusivity = pytesimal.numerical_methods.calculate_diffusivity(mantle_cond_value,
↪ mantle_heatcap_value,
                                                                        mantle_density_
↪value)

mantle_stability = pytesimal.numerical_methods.check_stability(mantle_diffusivity,␣
↪timestep, dr)

reg_stability = pytesimal.numerical_methods.check_stability(kappa_reg, timestep, dr)
```

Out:

```
Von Neumann stability criteria met
Von Neumann stability criteria met
```

We set up the boundary conditions for the mantle. For this example, we're using fixed temperature boundary conditions at both the surface and the core-mantle boundary: at the planetesimal's surface, the temperature is held at a fixed temperature specified in the parameter file (250 K), while at the core-mantle boundary, the temperature is updated by the core.

```
top_mantle_bc = pytesimal.numerical_methods.surface_dirichlet_bc
bottom_mantle_bc = pytesimal.numerical_methods.cmb_dirichlet_bc
```

Now we pass our boundary conditions, core object, initial temperature, material properties, geometry, and arrays to the *discretisation* function, which returns arrays of temperatures in the mantle and the core, and a list of latent heat values during core crystallisation:

```
(mantle_temperature_array,
 core_temperature_array,
 latent,
 ) = pytesimal.numerical_methods.discretisation(
    core_values,
    latent,
    temp_init,
    core_temp_init,
    top_mantle_bc,
    bottom_mantle_bc,
    temp_surface,
    mantle_temperature_array,
```

(continues on next page)

(continued from previous page)

```
    dr,
    core_temperature_array,
    timestep,
    r_core,
    radii,
    times,
    where_regolith,
    kappa_reg,
    mantle_conductivity,
    mantle_heatcap,
    mantle_density)
```

8.1.4 Analysing results

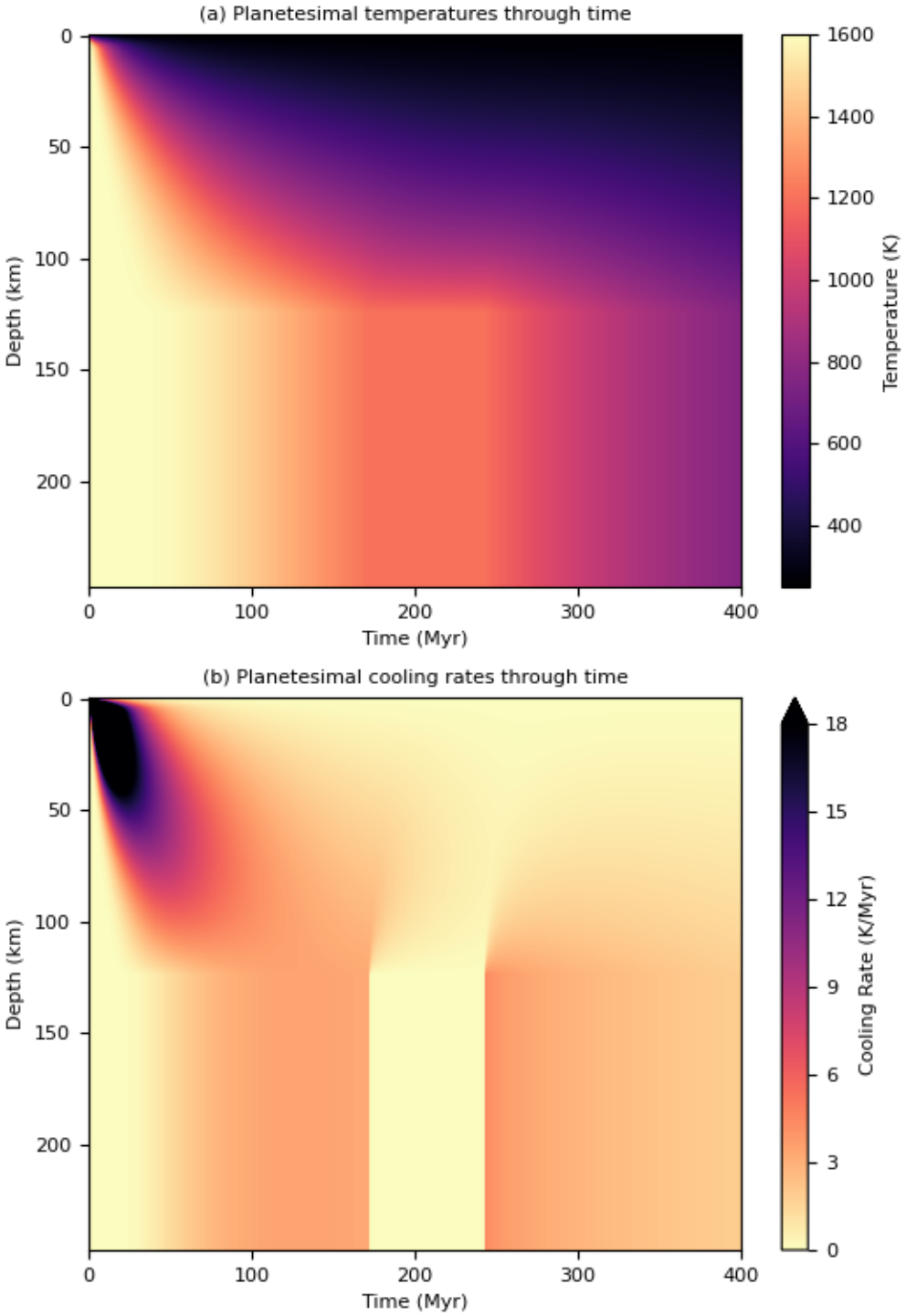
We can calculate the cooling rates in the planetesimal with the *analysis* module:

```
mantle_cooling_rates = pytesimal.analysis.cooling_rate(mantle_temperature_array,␣
↪timestep)
core_cooling_rates = pytesimal.analysis.cooling_rate(core_temperature_array, timestep)
```

We can then plot the results using the *load_plot_save* module. The *two_in_one* function can be called to quickly plot both the temperature and cooling rates:

```
fig_w = 6
fig_h = 9

pytesimal.load_plot_save.two_in_one(
    fig_w,
    fig_h,
    mantle_temperature_array,
    core_temperature_array,
    mantle_cooling_rates,
    core_cooling_rates,
    timestep=timestep)
```


(a) Planetesimal temperatures through time
Depth (km)
Time (Myr)
Temperature (K)
0
50
100
150
200
0
100
200
300
400
1600
1400
1200
1000
800
600
400
(b) Planetesimal cooling rates through time
Depth (km)
Time (Myr)
Cooling Rate (K/Myr)
18
15
12
9
6
3
0

8.1.5 Further information

Other examples are available in the gallery, and further information on the theoretical background can be found in Murphy Quinlan et al. (2021).

Total running time of the script: (0 minutes 43.549 seconds)

8.2 1. Quick Start

This script produces a default parameter file then uses this to set up a model run. The default parameter file reproduces the constant material properties case in Murphy Quinlan et al., 2021.

```
import pytesimal.load_plot_save
import pytesimal.quick_workflow
```

Define a filepath and filename for the parameter file:

```
folderpath = 'example_default/'
filename = 'example_parameters'
filepath = f'{folderpath}{filename}.txt'
```

Save a default parameters file to the filepath:

```
pytesimal.load_plot_save.make_default_param_file(filepath)
```

You can open this *json* file in a text editor and change the default values. For this example, we're just leaving the default values as they are and loading it without editing, and starting a model run:

```
pytesimal.quick_workflow.workflow(filename, folderpath)
```

Just wait for a minute or two for your planetesimal to evolve!

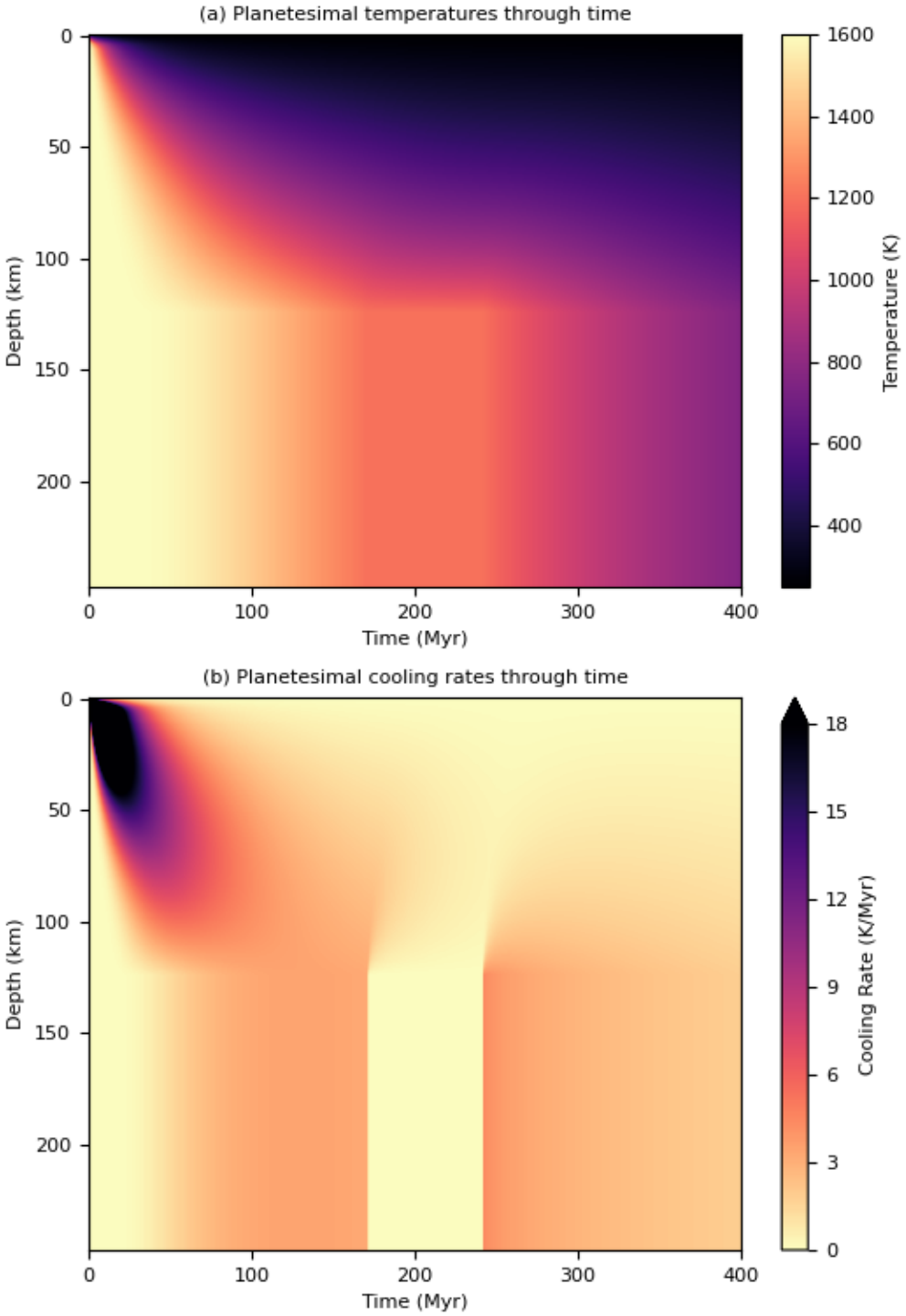
```
# Once 400 millions years has passed and your planetesimal has cooled down,
# we can load the results in to analyse and plot:

filepath = 'example_default/example_parameters_results.npz'
(temperatures,
 coretemp,
 dT_by_dt,
 dT_by_dt_core) = pytesimal.load_plot_save.read_datafile(filepath)
```

We can visualise the cooling history of the planetesimal:

```
# Specify a figure width and height
fig_w = 6
fig_h = 9

pytesimal.load_plot_save.two_in_one(fig_w,
                                    fig_h,
                                    temperatures,
                                    coretemp,
                                    dT_by_dt,
                                    dT_by_dt_core)
```


(a) Planetesimal temperatures through time
Depth (km)
Time (Myr)
Temperature (K)
0
50
100
150
200
0
100
200
300
400
1600
1400
1200
1000
800
600
400
(b) Planetesimal cooling rates through time
Depth (km)
Time (Myr)
Cooling Rate (K/Myr)
18
15
12
9
6
3
0

Total running time of the script: (1 minutes 53.256 seconds)

8.3 2. Constant Properties

This example shows step by step how to set up and run a model of a cooling planetesimal without using a parameters file. This example uses constant material properties in the mantle and reproduces the results of Murphy Quinlan et al. (2021).

As we're setting this up step-by-step instead of using the *pytesimal.quick_workflow* module, we need to import a selection of modules:

```
import pytesimal.setup_functions
import pytesimal.load_plot_save
import pytesimal.numerical_methods
import pytesimal.analysis
import pytesimal.core_function
import pytesimal.mantle_properties
```

Instead of creating and loading a parameter file, we're going to define variables here. The values match those of the constant thermal properties case in Murphy Quinlan et al. (2021):

```
timestep = 1E11  # s
r_planet = 250_000.0  # m
core_size_factor = 0.5  # fraction of r_planet
reg_fraction = 0.032  # fraction of r_planet
max_time = 400  # Myr
temp_core_melting = 1200.0  # K
core_cp = 850.0  # J/(kg K)
core_density = 7800.0  # kg/m^3
temp_init = 1600.0  # K
temp_surface = 250.0  # K
core_temp_init = 1600.0  # K
core_latent_heat = 270_000.0  # J/kg
kappa_reg = 5e-8  # m^2/s
dr = 1000.0  # m
```

The *setup_functions.set_up()* function creates empty arrays to be filled with resulting temperatures:

```
(r_core,
 radii,
 core_radii,
 reg_thickness,
 where_regolith,
 times,
 mantle_temperature_array,
 core_temperature_array) = pytesimal.setup_functions.set_up(timestep,
                                                            r_planet,
                                                            core_size_factor,
                                                            reg_fraction,
                                                            max_time,
                                                            dr)

# We define an empty list of latent heat that will
```

(continues on next page)

(continued from previous page)

```
# be filled as the core freezes
latent = []
```

Next, we instantiate the core object. This will keep track of the temperature of the core as the model runs, cooling as heat is extracted across the core-mantle boundary. This simple eutectic core model doesn't track inner core growth, but this is still a required argument to allow for future incorporation of more complex core objects:

```
core_values = pytesimal.core_function.IsothermalEutecticCore(
    initial_temperature=core_temp_init,
    melting_temperature=temp_core_melting,
    outer_r=r_core,
    inner_r=0,
    rho=core_density,
    cp=core_cp,
    core_latent_heat=core_latent_heat)
```

Then we define the mantle properties. The default is to have constant values, so we don't require any arguments for this example:

```
(mantle_conductivity,
 mantle_heatcap,
 mantle_density) = pytesimal.mantle_properties.set_up_mantle_properties()
```

You can check (or change) the value of these properties after they've been set up using one of the *MantleProperties* methods:

```
print(mantle_conductivity.getk())
```

Out:

```
3.0
```

If temperature dependent properties are used, temperature can be passed in as an argument to return the value at that temperature.

We need to set up the boundary conditions for the mantle. For this example, we're using fixed temperature boundary conditions at both the surface and the core-mantle boundary.

```
top_mantle_bc = pytesimal.numerical_methods.surface_dirichlet_bc
bottom_mantle_bc = pytesimal.numerical_methods.cmb_dirichlet_bc

# Now we let the temperature inside the planestesimal evolve. This is the
# slowest part of the code, because it has to iterate over all radii and
# time.
# This will take a minute or two!

(mantle_temperature_array,
 core_temperature_array,
 latent,
 ) = pytesimal.numerical_methods.discretisation(
    core_values,
    latent,
    temp_init,
```

(continues on next page)

(continued from previous page)

```
    core_temp_init,
    top_mantle_bc,
    bottom_mantle_bc,
    temp_surface,
    mantle_temperature_array,
    dr,
    core_temperature_array,
    timestep,
    r_core,
    radii,
    times,
    where_regolith,
    kappa_reg,
    mantle_conductivity,
    mantle_heatcap,
    mantle_density)
```

This function fills the empty arrays produced by *setup_functions.set_up()* with calculated temperatures for the mantle and core.

Now we can use the *pytesimal.analysis* module to find out more about the model run. We can check when the core was freezing, so we can compare this to the cooling history of meteorites and see whether they can be expected to record magnetic remnants of a core dynamo:

```
(core_frozen,
 times_frozen,
 time_core_frozen,
 fully_frozen) = pytesimal.analysis.core_freezing(core_temperature_array,
                                                  max_time,
                                                  times,
                                                  latent,
                                                  temp_core_melting,
                                                  timestep)
```

Then, we can calculate arrays of cooling rates from the temperature arrays:

```
mantle_cooling_rates = pytesimal.analysis.cooling_rate(mantle_temperature_array,
                                                       timestep)
core_cooling_rates = pytesimal.analysis.cooling_rate(core_temperature_array,
                                                     timestep)
```

Meteorite data (the diameter of 'cloudy-zone particles') can be used to estimate the rate at which the meteorites cooled through a specific temperature (C. W. Yang et al., 1997). The *analysis.cooling_rate_cloudyzone_diameter* function calculates the cooling rate in K/Myr, while the *analysis.cooling_rate_to_seconds* function converts this to K/s which allows comparison to our result arrays.

```
d_im = 147  # cz diameter in nm
d_esq = 158  # cz diameter in nm

imilac_cooling_rate = pytesimal.analysis.cooling_rate_to_seconds(
    pytesimal.analysis.cooling_rate_cloudyzone_diameter(d_im))
esquel_cooling_rate = pytesimal.analysis.cooling_rate_to_seconds(
    pytesimal.analysis.cooling_rate_cloudyzone_diameter(d_esq))
```

We can use this cooling rate information to find out how deep within their parent bodies these meteorites originally formed, and when they passed through the temperature of tetrataenite formation (when magnetism can be recorded). The *analysis.meteorite_depth_and_timing()* function returns the source depth of the meteorite material in the parent body based on the metal cooling rates at 800 K (as a depth from surface in km and as a radial value from the center of the planet in m), the time that the meteorite cools through the tetrataenite formation temperature in comparison to the core crystallisation period, and a string defining this relation between paleomagnetism recording and dynamo activity:

```
(im_depth,
 im_string_result,
 im_time_core_frozen,
 im_Time_of_Crossing,
 im_Critical_Radius) = pytesimal.analysis.meteorite_depth_and_timing(
    imilac_cooling_rate,
    mantle_temperature_array,
    mantle_cooling_rates,
    radii,
    r_planet,
    core_size_factor,
    time_core_frozen,
    fully_frozen,
    dr=1000,
)

(esq_depth,
 esq_string_result,
 esq_time_core_frozen,
 esq_Time_of_Crossing,
 esq_Critical_Radius) = pytesimal.analysis.meteorite_depth_and_timing(
    esquel_cooling_rate,
    mantle_temperature_array,
    mantle_cooling_rates,
    radii,
    r_planet,
    core_size_factor,
    time_core_frozen,
    fully_frozen,
    dr=1000,
)

print(f"Imilac depth: {im_depth}; Imilac timing: {im_string_result}")
print(f"Esquel depth: {esq_depth}; Esquel timing: {esq_string_result}")
```

Out:

```
Imilac depth: 57.0; Imilac timing: Core has started solidifying
Esquel depth: 64.0; Esquel timing: Core has started solidifying
```

If you need to save the meteorite results, they can be saved to a dictionary which can then be passed to the *load_plot_save.save_params_and_results*. This allows for any number of meteorites to be analysed and only the relevant data stored:

```
meteorite_results_dict = { 'Esq results':
                                {'depth': esq_depth,
                                 'text result': esq_string_result},
```

(continues on next page)

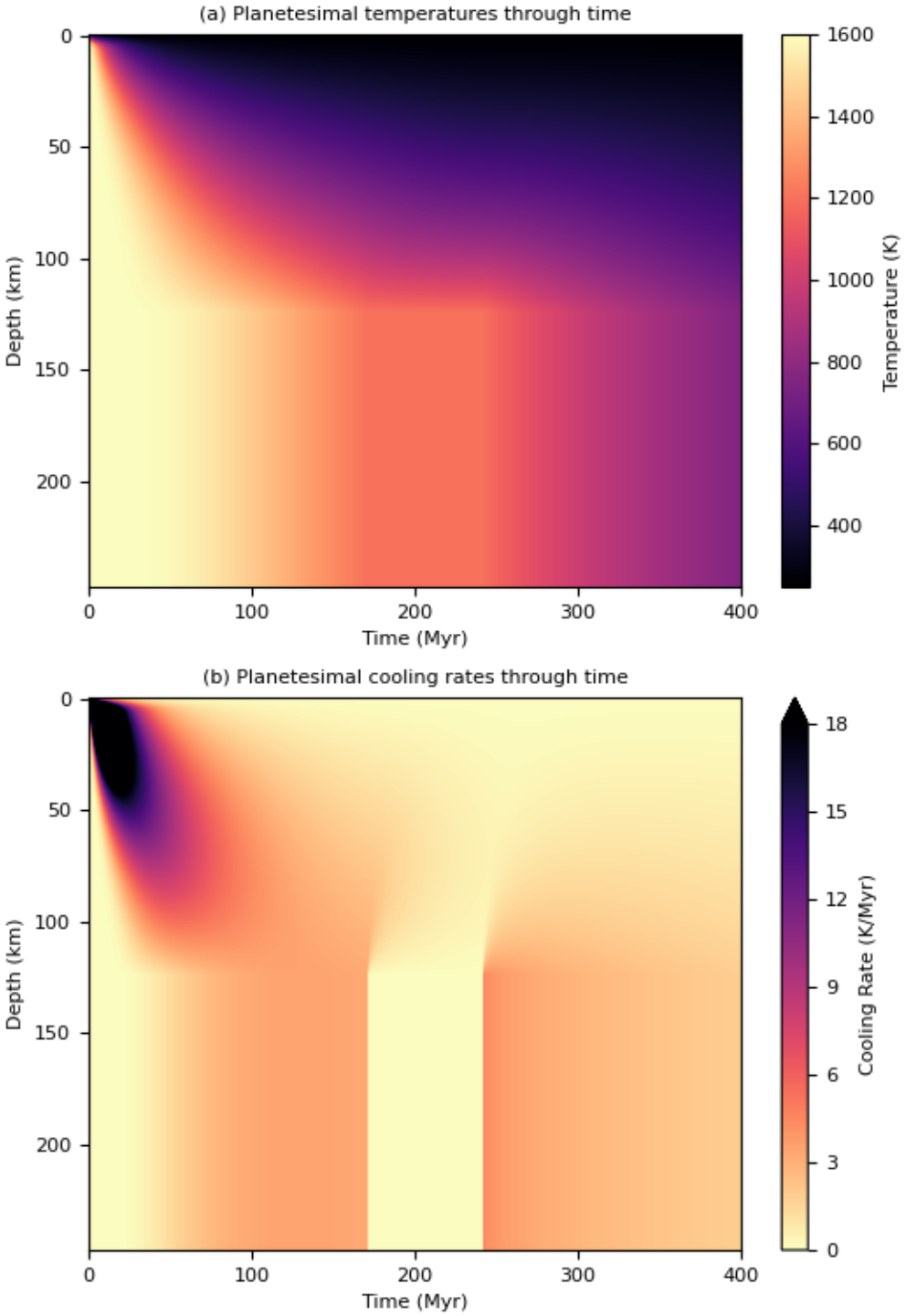
(continued from previous page)

```
                            'Im results':
                                {'depth' : im_depth,
                                 'text result': im_string_result,
                                 'critical radius': im_Critical_Radius}}
```

To get an overview of the cooling history of the body, it's very useful to plot the temperatures and cooling rates as a heatmap through time. In order to plot the results, we need to define a figure height and width, then call *load_plot_save.plot_temperature_history()*, *load_plot_save.plot_coolingrate_history()* or *load_plot_save.two_in_one()*. These functions convert the cooling rate from K/timestep to K/Myr to make the results more human-readable.

```
fig_w = 6
fig_h = 9

pytesimal.load_plot_save.two_in_one(
    fig_w,
    fig_h,
    mantle_temperature_array,
    core_temperature_array,
    mantle_cooling_rates,
    core_cooling_rates,)
```


(a) Planetesimal temperatures through time
Depth (km)
Time (Myr)
Temperature (K)
0
50
100
150
200
0
100
200
300
400
1600
1400
1200
1000
800
600
400
(b) Planetesimal cooling rates through time
Depth (km)
Time (Myr)
Cooling Rate (K/Myr)
18
15
12
9
6
3
0

There are a few formats or ways to save the results. The temperature and cooling rate arrays can be saved as compressed *.npz* arrays, to be loaded at a later time and replotted/new meteorite formation depths calculated etc. The input parameters can be saved as a *.json* file, which allows the run to be documented and provides all the metadata needed to reproduce the results. For either of these, a results folder and results filename is needed. The folder can be defined relative to the working directory, or with an absolute path. An absolute path usually results in less confusion!

```
# define folder and check it exists:
folder = 'workflow'
pytesimal.load_plot_save.check_folder_exists(folder)
# define a results filename prefix:
result_filename = 'constant_workflow_results'
```

The result arrays can now be saved:

```
pytesimal.load_plot_save.save_result_arrays(result_filename,
                                            folder,
                                            mantle_temperature_array,
                                            core_temperature_array,
                                            mantle_cooling_rates,
                                            core_cooling_rates)
```

In order to save the result parameter file, we also need to define a *run_ID*, a descriptive string to identify the model run, and clarify whether we used constant or variable thermal properties:

```
run_ID = 'Example run with default properties'
cond_constant = 'y'
density_constant = 'y'
heat_cap_constant = 'y'

pytesimal.load_plot_save.save_params_and_results(
    result_filename, run_ID, folder, timestep, r_planet, core_size_factor,
    reg_fraction, max_time, temp_core_melting, mantle_heatcap.getcp(),
    mantle_density.getrho(), mantle_conductivity.getk(), core_cp, core_density,
    temp_init, temp_surface, core_temp_init, core_latent_heat,
    kappa_reg, dr, cond_constant, density_constant,
    heat_cap_constant, time_core_frozen, fully_frozen,
    meteorite_results=meteorite_results_dict,
    latent_list_len=len(latent))
```

This results file can then be loaded as a parameter file if you want to repeat the same set up.

Total running time of the script: (1 minutes 54.210 seconds)

8.4 3. Variable Properties

This example shows step by step how to set up and run a model of a cooling planetesimal without using a parameters file. This example uses temperature dependent material properties in the mantle and reproduces the results of Murphy Quinlan et al. (2021).

As we're setting this up step-by-step instead of using the *pytesimal.quick_workflow* module, we need to import a selection of modules:

```
import pytesimal.setup_functions
import pytesimal.load_plot_save
import pytesimal.numerical_methods
import pytesimal.analysis
import pytesimal.core_function
import pytesimal.mantle_properties
```

Instead of creating and loading a parameter file, we're going to define variables here. The values match those of the constant thermal properties case in Murphy Quinlan et al. (2021):

```
timestep = 1E11  # s
r_planet = 250_000.0  # m
core_size_factor = 0.5  # fraction of r_planet
reg_fraction = 0.032  # fraction of r_planet
max_time = 400  # Myr
temp_core_melting = 1200.0  # K
core_cp = 850.0  # J/(kg K)
core_density = 7800.0  # kg/m^3
temp_init = 1600.0  # K
temp_surface = 250.0  # K
core_temp_init = 1600.0  # K
core_latent_heat = 270_000.0  # J/kg
kappa_reg = 5e-8  # m^2/s
dr = 1000.0  # m
```

The *setup_functions.set_up()* function creates empty arrays to be filled with resulting temperatures:

```
(r_core,
 radii,
 core_radii,
 reg_thickness,
 where_regolith,
 times,
 mantle_temperature_array,
 core_temperature_array) = pytesimal.setup_functions.set_up(timestep,
                                                            r_planet,
                                                            core_size_factor,
                                                            reg_fraction,
                                                            max_time,
                                                            dr)

# We define an empty list of latent heat that will
# be filled as the core freezes
latent = []
```

Next, we instantiate the core object. This will keep track of the temperature of the core as the model runs, cooling as heat is extracted across the core-mantle boundary. This simple eutectic core model doesn't track inner core growth, but this is still a required argument to allow for future incorporation of more complex core objects:

```
core_values = pytesimal.core_function.IsothermalEutecticCore(
    initial_temperature=core_temp_init,
    melting_temperature=temp_core_melting,
    outer_r=r_core,
```

(continues on next page)

(continued from previous page)

```
    inner_r=0,
    rho=core_density,
    cp=core_cp,
    core_latent_heat=core_latent_heat)
```

Then we define the mantle properties. We want to use temperature-dependent properties, so we need to specify this:

```
(mantle_conductivity,
 mantle_heatcap,
 mantle_density) = pytesimal.mantle_properties.set_up_mantle_properties(
    cond_constant='n',
    density_constant='n',
    heat_cap_constant='n'
)
```

You can check (or change) the value of these properties after they've been set up using one of the *MantleProperties* methods:

```
print(f' Mantle conductivity {mantle_conductivity.getk()} W/(m K)')
```

Out:

```
Mantle conductivity 3.411005666195278 W/(m K)
```

When temperature dependent properties are used, temperature can be passed in as an argument to return the value at that temperature. The default temperature is 295 K, so if no temperature argument is passed, the function is evaluated at *T=295*.

We need to set up the boundary conditions for the mantle. For this example, we're using fixed temperature boundary conditions at both the surface and the core-mantle boundary.

```
top_mantle_bc = pytesimal.numerical_methods.surface_dirichlet_bc
bottom_mantle_bc = pytesimal.numerical_methods.cmb_dirichlet_bc

# Now we let the temperature inside the planestesimal evolve. This is the
# slowest part of the code, because it has to iterate over all radii and
# time.
# This will take a minute or two!
# The mantle property objects are passed in in the same way as in the
# example with constant thermal properties.

(mantle_temperature_array,
 core_temperature_array,
 latent,
 ) = pytesimal.numerical_methods.discretisation(
    core_values,
    latent,
    temp_init,
    core_temp_init,
    top_mantle_bc,
    bottom_mantle_bc,
    temp_surface,
    mantle_temperature_array,
```

(continues on next page)

(continued from previous page)

```
    dr,
    core_temperature_array,
    timestep,
    r_core,
    radii,
    times,
    where_regolith,
    kappa_reg,
    mantle_conductivity,
    mantle_heatcap,
    mantle_density)
```

This function fills the empty arrays produced by *setup_functions.set_up()* with calculated temperatures for the mantle and core.

Now we can use the *pytesimal.analysis* module to find out more about the model run. We can check when the core was freezing, so we can compare this to the cooling history of meteorites and see whether they can be expected to record magnetic remnants of a core dynamo:

```
(core_frozen,
 times_frozen,
 time_core_frozen,
 fully_frozen) = pytesimal.analysis.core_freezing(core_temperature_array,
                                                  max_time,
                                                  times,
                                                  latent,
                                                  temp_core_melting,
                                                  timestep)
```

Then, we can calculate arrays of cooling rates from the temperature arrays:

```
mantle_cooling_rates = pytesimal.analysis.cooling_rate(mantle_temperature_array,
                                                       timestep)
core_cooling_rates = pytesimal.analysis.cooling_rate(core_temperature_array,
                                                     timestep)
```

Meteorite data (the diameter of 'cloudy-zone particles') can be used to estimate the rate at which the meteorites cooled through a specific temperature (C. W. Yang et al., 1997). The *analysis.cooling_rate_cloudyzone_diameter* function calculates the cooling rate in K/Myr, while the *analysis.cooling_rate_to_seconds* function converts this to K/s which allows comparison to our result arrays.

```
d_im = 147  # cz diameter in nm
d_esq = 158  # cz diameter in nm

imilac_cooling_rate = pytesimal.analysis.cooling_rate_to_seconds(
    pytesimal.analysis.cooling_rate_cloudyzone_diameter(d_im))
esquel_cooling_rate = pytesimal.analysis.cooling_rate_to_seconds(
    pytesimal.analysis.cooling_rate_cloudyzone_diameter(d_esq))
```

We can use this cooling rate information to find out how deep within their parent bodies these meteorites originally formed, and when they passed through the temperature of tetrataenite formation (when magnetism can be recorded). The *analysis.meteorite_depth_and_timing()* function returns the source depth of the meteorite material in the parent body based on the metal cooling rates at 800 K (as a depth from surface in km and as a radial value from the center of

the planet in m), the time that the meteorite cools through the tetrataenite formation temperature in comparison to the core crystallisation period, and a string defining this relation between paleomagnetism recording and dynamo activity:

```
(im_depth,
 im_string_result,
 im_time_core_frozen,
 im_Time_of_Crossing,
 im_Critical_Radius) = pytesimal.analysis.meteorite_depth_and_timing(
    imilac_cooling_rate,
    mantle_temperature_array,
    mantle_cooling_rates,
    radii,
    r_planet,
    core_size_factor,
    time_core_frozen,
    fully_frozen,
    dr=1000,
)

(esq_depth,
 esq_string_result,
 esq_time_core_frozen,
 esq_Time_of_Crossing,
 esq_Critical_Radius) = pytesimal.analysis.meteorite_depth_and_timing(
    esquel_cooling_rate,
    mantle_temperature_array,
    mantle_cooling_rates,
    radii,
    r_planet,
    core_size_factor,
    time_core_frozen,
    fully_frozen,
    dr=1000,
)

print(f"Imilac depth: {im_depth}; Imilac timing: {im_string_result}")
print(f"Esquel depth: {esq_depth}; Esquel timing: {esq_string_result}")
```

Out:

```
Imilac depth: 61.0; Imilac timing: Core has not started solidifying yet
Esquel depth: 68.0; Esquel timing: Core has started solidifying
```

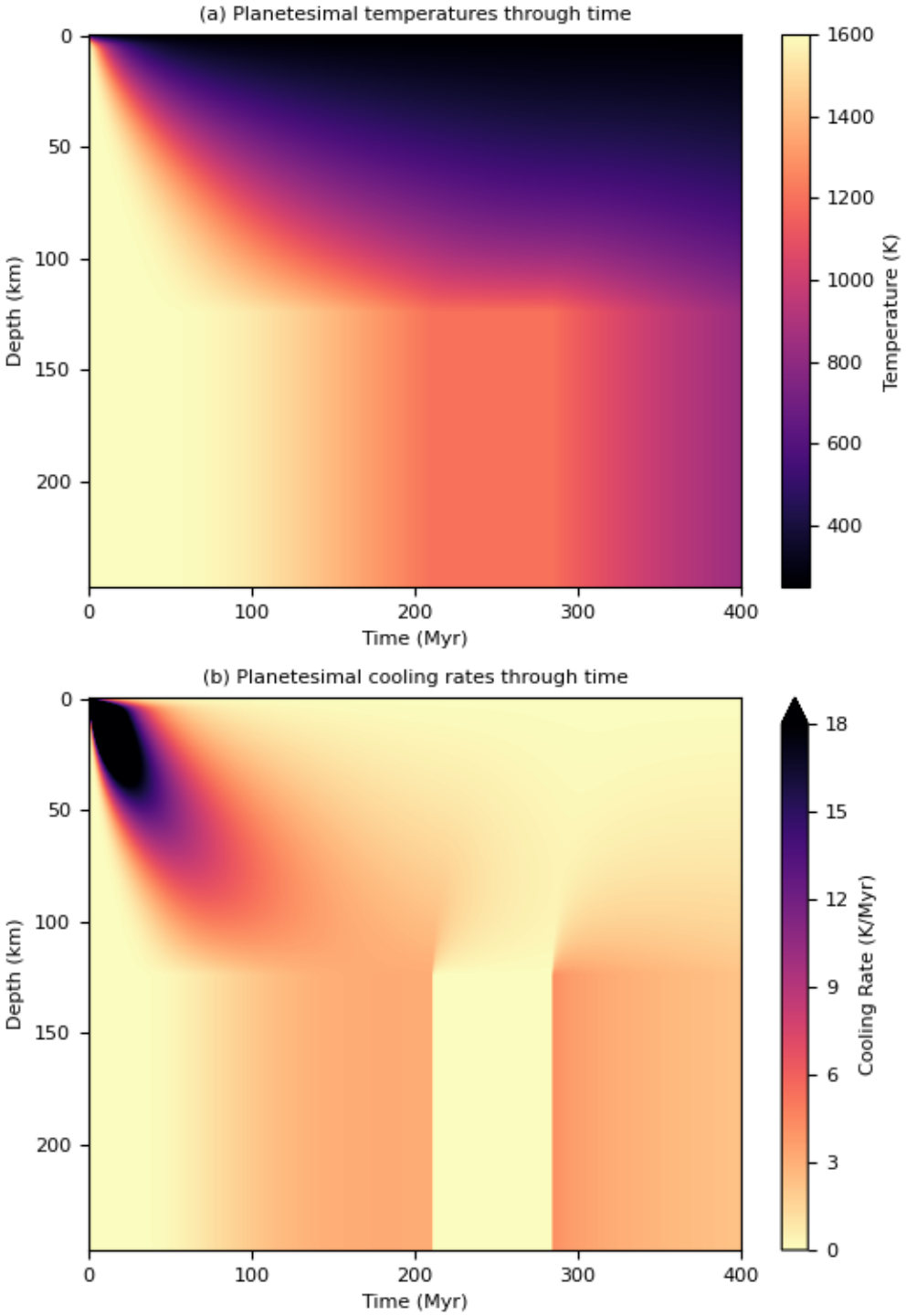
If you need to save the meteorite results, they can be saved to a dictionary which can then be passed to the *load_plot_save.save_params_and_results*. This allows for any number of meteorites to be analysed and only the relevant data stored:

```
meteorite_results_dict = {'Esq results':
                              {'depth': esq_depth,
                               'text result': esq_string_result},
                          'Im results':
                              {'depth': im_depth,
                               'text result': im_string_result,
                               'critical radius': im_Critical_Radius}}
```

To get an overview of the cooling history of the body, it's very useful to plot the temperatures and cooling rates as a heatmap through time. In order to plot the results, we need to define a figure height and width, then call *load_plot_save.plot_temperature_history()*, *load_plot_save.plot_coolingrate_history()* or *load_plot_save.two_in_one()*. These functions convert the cooling rate from K/timestep to K/Myr to make the results more human-readable.

```
fig_w = 6
fig_h = 9

pytesimal.load_plot_save.two_in_one(
    fig_w,
    fig_h,
    mantle_temperature_array,
    core_temperature_array,
    mantle_cooling_rates,
    core_cooling_rates, )
```


(a) Planetesimal temperatures through time
Depth (km)
Time (Myr)
Temperature (K)
0
50
100
150
200
0
100
200
300
400
1600
1400
1200
1000
800
600
400
(b) Planetesimal cooling rates through time
Depth (km)
Time (Myr)
Cooling Rate (K/Myr)
18
15
12
9
6
3
0

There are a few formats or ways to save the results. The temperature and cooling rate arrays can be saved as compressed *.npz* arrays, to be loaded at a later time and replotted/new meteorite formation depths calculated etc. The input parameters can be saved as a *.json* file, which allows the run to be documented and provides all the metadata needed to reproduce the results. For either of these, a results folder and results filename is needed. The folder can be defined relative to the working directory, or with an absolute path. An absolute path usually results in less confusion!

```
# define folder and check it exists:
folder = 'workflow'
pytesimal.load_plot_save.check_folder_exists(folder)
# define a results filename prefix:
result_filename = 'variable_workflow_results'
```

The result arrays can now be saved:

```
pytesimal.load_plot_save.save_result_arrays(result_filename,
                                            folder,
                                            mantle_temperature_array,
                                            core_temperature_array,
                                            mantle_cooling_rates,
                                            core_cooling_rates)
```

In order to save the result parameter file, we also need to define a *run_ID*, a descriptive string to identify the model run, and clarify whether we used constant or variable thermal properties:

```
run_ID = 'Example run with default variable properties'
cond_constant = 'n'
density_constant = 'n'
heat_cap_constant = 'n'

pytesimal.load_plot_save.save_params_and_results(
    result_filename, run_ID, folder, timestep, r_planet, core_size_factor,
    reg_fraction, max_time, temp_core_melting, mantle_heatcap.getcp(),
    mantle_density.getrho(), mantle_conductivity.getk(), core_cp, core_density,
    temp_init, temp_surface, core_temp_init, core_latent_heat,
    kappa_reg, dr, cond_constant, density_constant,
    heat_cap_constant, time_core_frozen, fully_frozen,
    meteorite_results=meteorite_results_dict,
    latent_list_len=len(latent))
```

This results file can then be loaded as a parameter file if you want to repeat the same set up.

Total running time of the script: (4 minutes 20.792 seconds)

8.5 4. Planetesimal Without a Core

This example shows step by step how to set up and run a model of a cooling planetesimal without a core. This example uses constant material properties in the mantle, see Murphy Quinlan et al. (2021) and references therein.

While the material properties used in this example are suitable for modelling the olivine mantle of a differentiated body, the model set-up may be useful for modelling undifferentiated meteorite parent bodies with appropriate thermal properties.

This model set-up also allows for comparison to an analytical solution for conductive cooling in a sphere (see Murphy Quinlan et al. (2021)).

As we're setting this up step-by-step instead of using the *pytesimal.quick_workflow* module, we need to import a selection of modules:

```
import pytesimal.setup_functions
import pytesimal.load_plot_save
import pytesimal.numerical_methods
import pytesimal.analysis
import pytesimal.core_function
import pytesimal.mantle_properties
```

Instead of creating and loading a parameter file, we're going to define variables here. The values match those of the constant thermal properties case in Murphy Quinlan et al. (2021), except for the *core_size_factor* value:

```
timestep = 1E11  # s
r_planet = 250_000.0  # m
reg_fraction = 0.032  # fraction of r_planet
max_time = 400  # Myr
temp_core_melting = 1200.0  # K
core_cp = 850.0  # J/(kg K)
core_density = 7800.0  # kg/m^3
temp_init = 1600.0  # K
temp_surface = 250.0  # K
core_temp_init = 1600.0  # K
core_latent_heat = 270_000.0  # J/kg
kappa_reg = 5e-8  # m^2/s
dr = 1000.0  # m
```

As we want to model an undifferentiated body, we don't want to include a core in our model set up. Because we wanted it to be easy to swap between different model set-ups, the *numerical_methods.discretisation()* function still requires a core object to be instantiated. When the boundary conditions are set up correctly, the core object does not interact with the mantle and does not influence the cooling. Setting *core_size_factor=0* can lead to scalar overflow errors, which can be avoided by setting *core_size_factor* to a small, non-zero number that is smaller than the grid spacing (so the core_temperature_array has dimension 0 along radius):

```
core_size_factor = 0.001
```

The *setup_functions.set_up()* function creates empty arrays to be filled with resulting temperatures:

```
(r_core,
 radii,
 core_radii,
 reg_thickness,
 where_regolith,
 times,
 mantle_temperature_array,
 core_temperature_array) = pytesimal.setup_functions.set_up(timestep,
                                                            r_planet,
                                                            core_size_factor,
                                                            reg_fraction,
                                                            max_time,
                                                            dr)

# We define an empty list of latent heat
latent = []
```

We can check that our "no-core" set-up has worked:

```
print(core_temperature_array.shape)
```

Out:

```
(0, 126229)
```

Next, we instantiate the core object. As explained previously, this model set up does not include a core, but in order to make the code as modular as possible, a core object still needs to be passed in to the main timestepping function, *numerical_methods.discretisation*. We've just copied across the default arguments for convenience:

```
core_values = pytesimal.core_function.IsothermalEutecticCore(
    initial_temperature=core_temp_init,
    melting_temperature=temp_core_melting,
    outer_r=r_core,
    inner_r=0,
    rho=core_density,
    cp=core_cp,
    core_latent_heat=core_latent_heat)
```

Then we define the mantle properties. The default is to have constant values, so we don't require any arguments for this example:

```
(mantle_conductivity,
 mantle_heatcap,
 mantle_density) = pytesimal.mantle_properties.set_up_mantle_properties()
```

You can check (or change) the value of these properties after they've been set up using one of the *MantleProperties* methods:

```
print(mantle_conductivity.getk())
```

Out:

```
3.0
```

If temperature dependent properties are used, temperature can be passed in as an argument to return the value at that temperature.

We need to set up the boundary conditions for the mantle. For this example, we're using fixed temperature boundary conditions at the surface, and a zero-flux condition at the bottom to ensure symmetry.

```
top_mantle_bc = pytesimal.numerical_methods.surface_dirichlet_bc
bottom_mantle_bc = pytesimal.numerical_methods.cmb_neumann_bc

# Now we let the temperature inside the planestesimal evolve. This is the
# slowest part of the code, because it has to iterate over all radii and
# time.
# This will take a minute or two!
# Note that this function call is the exact same as in the default constant
# and variable cases that included a core.

(mantle_temperature_array,
 core_temperature_array,
```

(continues on next page)

(continued from previous page)

```
 latent,
 ) = pytesimal.numerical_methods.discretisation(
    core_values,
    latent,
    temp_init,
    core_temp_init,
    top_mantle_bc,
    bottom_mantle_bc,
    temp_surface,
    mantle_temperature_array,
    dr,
    core_temperature_array,
    timestep,
    r_core,
    radii,
    times,
    where_regolith,
    kappa_reg,
    mantle_conductivity,
    mantle_heatcap,
    mantle_density)
```

This function fills the empty arrays produced by *setup_functions.set_up()* with calculated temperatures.

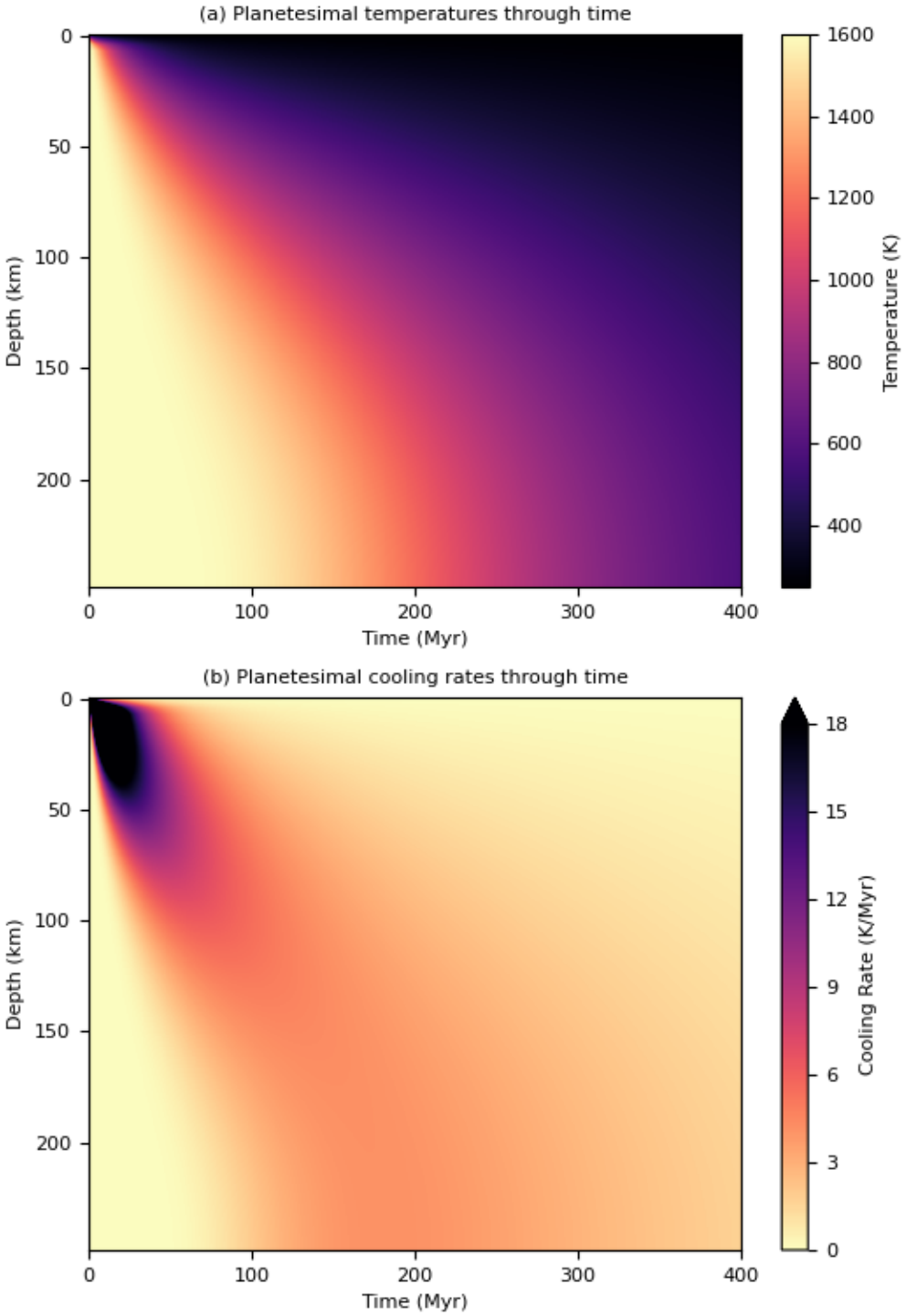
Our next step is to calculate cooling rates for the body. Usually, we calculate cooling rates for the mantle and the core in the same way. As the core temperature array is empty, but we still need a *core_cooling_rates* array to pass to our plotting function, we just set *core_cooling_rates* = *core_temperature_array* to save time computing a meaningless empty array:

```
mantle_cooling_rates = pytesimal.analysis.cooling_rate(mantle_temperature_array,␣
↪timestep)
core_cooling_rates = core_temperature_array
```

To get an overview of the cooling history of the body, it's very useful to plot the temperatures and cooling rates as a heatmap through time. In order to plot the results, we need to define a figure height and width, then call *load_plot_save.plot_temperature_history()*, *load_plot_save.plot_coolingrate_history()* or *load_plot_save.two_in_one()*. These functions convert the cooling rate from K/timestep to K/Myr to make the results more human-readable.

```
fig_w = 6
fig_h = 9

pytesimal.load_plot_save.two_in_one(
    fig_w,
    fig_h,
    mantle_temperature_array,
    core_temperature_array,
    mantle_cooling_rates,
    core_cooling_rates,)
```


(a) Planetesimal temperatures through time
Depth (km)
Time (Myr)
Temperature (K)
0
50
100
150
200
0
100
200
300
400
1600
1400
1200
1000
800
600
400
(b) Planetesimal cooling rates through time
Depth (km)
Time (Myr)
Cooling Rate (K/Myr)
18
15
12
9
6
3
0

Results can be saved in the same way as for the constant and variable properties examples.

Total running time of the script: (2 minutes 52.410 seconds)

8.6 5. Reproducing Previous Results

This example recreates the results of Bryson et al. (2015) giving the same depths of formation of two pallasite meteorites.

As we're setting this up step-by-step instead of using the *pytesimal.quick_workflow* module, we need to import a selection of modules:

```
import pytesimal.setup_functions
import pytesimal.load_plot_save
import pytesimal.numerical_methods
import pytesimal.analysis
import pytesimal.core_function
import pytesimal.mantle_properties
```

Instead of creating and loading a parameter file, we're going to define variables here. The values are from and recreate the results of Bryson et al. (2015), with explanatory comments:

```
# These values are quoted in Bryson et al. (2015) or
# the references therein:

# material properties:
mantle_diffusivity = 5e-7
mantle_conductivity_value = 3.0
mantle_density_value = 3000.0

kappa_reg = 5e-8  # m^2/s

core_cp = 850.0  # J/(kg K)
core_density = 7800.0  # kg/m^3

# geometry:
r_planet = 200_000.0  # planetesimal radius in m
reg_m = 8_000.0  # megaregolith thickness in m

# temperatures:
temp_core_melting = 1200.0  # K
temp_init = 1600.0  # K
temp_surface = 250.0  # K
core_temp_init = 1600.0  # K
core_latent_heat = 270_000.0  # J/kg

# discretisation:
timestep = 2E11  # s
dr = 1000.0  # m
max_time = 400  # Myr

# This value isn't explicitly listed in Bryson et al., or references
# as Bryson et al. (2015) uses diffusivity instead
```

(continues on next page)

(continued from previous page)

```
mantle_heatcap_value = mantle_conductivity_value / (mantle_density_value * mantle_
↪diffusivity)

# Bryson et al. (2015) list regolith in km as opposed to
# as a fraction of body radius
reg_fraction = reg_m / r_planet  # fraction of r_planet

# We don't want to incorporate the 8 km regolith when calculating core size:
# Bryson et al. (2015) don't seem to include the regolith when calculating
# the core size, ie the core is 50% of the non-regolith body radius.
# We don't want to incorporate the 8 km regolith when calculating core size:
core_m = (r_planet - reg_m) * 0.5  # 100_000.0 # core size in m
core_size_factor = core_m / r_planet  # fraction of r_planet
```

The *setup_functions.set_up()* function creates empty arrays to be filled with resulting temperatures:

```
(r_core,
 radii,
 core_radii,
 reg_thickness,
 where_regolith,
 times,
 mantle_temperature_array,
 core_temperature_array) = pytesimal.setup_functions.set_up(timestep,
                                                            r_planet,
                                                            core_size_factor,
                                                            reg_fraction,
                                                            max_time,
                                                            dr)

# We define an empty list of latent heat that will
# be filled as the core freezes
latent = []
```

Next, we instantiate the core object. This will keep track of the temperature of the core as the model runs, cooling as heat is extracted across the core-mantle boundary. This simple eutectic core model doesn't track inner core growth, but this is still a required argument to allow for future incorporation of more complex core objects:

```
core_values = pytesimal.core_function.IsothermalEutecticCore(
    initial_temperature=core_temp_init,
    melting_temperature=temp_core_melting,
    outer_r=r_core,
    inner_r=0,
    rho=core_density,
    cp=core_cp,
    core_latent_heat=core_latent_heat)
```

Then we define the mantle properties. The default is to have constant values, so we don't require any arguments for this example:

```
(mantle_conductivity,
 mantle_heatcap,
 mantle_density) = pytesimal.mantle_properties.set_up_mantle_properties()
```

You can check (or change) the value of these properties after they've been set up using one of the *MantleProperties* methods. We want to set these values equal to the values used by Bryson et al. (2015):

```
mantle_conductivity.setk(mantle_conductivity_value)
mantle_heatcap.setcp(mantle_heatcap_value)
mantle_density.setrho(mantle_density_value)
```

You can check that the correct values have been assigned:

```
print(mantle_conductivity.getk())
print(mantle_heatcap.getcp())
print(mantle_density.getrho())
```

Out:

```
3.0
2000.0
3000.0
```

If temperature dependent properties are used, temperature can be passed in as an argument to return the value at that temperature.

We need to set up the boundary conditions for the mantle. For this example, we're using fixed temperature boundary conditions at both the surface and the core-mantle boundary.

```
top_mantle_bc = pytesimal.numerical_methods.surface_dirichlet_bc
bottom_mantle_bc = pytesimal.numerical_methods.cmb_dirichlet_bc

(mantle_temperature_array,
 core_temperature_array,
 latent,
 ) = pytesimal.numerical_methods.discretisation(
    core_values,
    latent,
    temp_init,
    core_temp_init,
    top_mantle_bc,
    bottom_mantle_bc,
    temp_surface,
    mantle_temperature_array,
    dr,
    core_temperature_array,
    timestep,
    r_core,
    radii,
    times,
    where_regolith,
    kappa_reg,
    mantle_conductivity,
    mantle_heatcap,
    mantle_density)
```

This function fills the empty arrays produced by *setup_functions.set_up()* with calculated temperatures for the mantle and core.

Now we can use the *pytesimal.analysis* module to find out more about the model run. We can check when the core was

freezing, so we can compare this to the cooling history of meteorites and see whether they can be expected to record magnetic remnants of a core dynamo:

```
(core_frozen,
 times_frozen,
 time_core_frozen,
 fully_frozen) = pytesimal.analysis.core_freezing(core_temperature_array,
                                                  max_time,
                                                  times,
                                                  latent,
                                                  temp_core_melting,
                                                  timestep)
```

Then, we can calculate arrays of cooling rates from the temperature arrays:

```
mantle_cooling_rates = pytesimal.analysis.cooling_rate(
    mantle_temperature_array,
    timestep)
core_cooling_rates = pytesimal.analysis.cooling_rate(core_temperature_array,
                                                     timestep)
```

Meteorite data (the diameter of 'cloudy-zone particles') can be used to estimate the rate at which the meteorites cooled through a specific temperature (C. W. Yang et al., 1997). The *analysis.cooling_rate_cloudyzone_diameter* function calculates the cooling rate in K/Myr, while the *analysis.cooling_rate_to_seconds* function converts this to K/s which allows comparison to our result arrays.

```
d_im = 147  # cz diameter in nm
d_esq = 158  # cz diameter in nm

imilac_cooling_rate = pytesimal.analysis.cooling_rate_to_seconds(
    pytesimal.analysis.cooling_rate_cloudyzone_diameter(d_im))
esquel_cooling_rate = pytesimal.analysis.cooling_rate_to_seconds(
    pytesimal.analysis.cooling_rate_cloudyzone_diameter(d_esq))
```

We can use this cooling rate information to find out how deep within their parent bodies these meteorites originally formed, and when they passed through the temperature of tetrataenite formation (when magnetism can be recorded). The *analysis.meteorite_depth_and_timing()* function returns the source depth of the meteorite material in the parent body based on the metal cooling rates at 800 K (as a depth from surface in km and as a radial value from the center of the planet in m), the time that the meteorite cools through the tetrataenite formation temperature in comparison to the core crystallisation period, and a string defining this relation between paleomagnetism recording and dynamo activity:

```
(im_depth,
 im_string_result,
 im_time_core_frozen,
 im_Time_of_Crossing,
 im_Critical_Radius) = pytesimal.analysis.meteorite_depth_and_timing(
    imilac_cooling_rate,
    mantle_temperature_array,
    mantle_cooling_rates,
    radii,
    r_planet,
    core_size_factor,
    time_core_frozen,
    fully_frozen,
```

(continues on next page)

(continued from previous page)

```
    dr=1000,
    dt=timestep
)

(esq_depth,
 esq_string_result,
 esq_time_core_frozen,
 esq_Time_of_Crossing,
 esq_Critical_Radius) = pytesimal.analysis.meteorite_depth_and_timing(
    esquel_cooling_rate,
    mantle_temperature_array,
    mantle_cooling_rates,
    radii,
    r_planet,
    core_size_factor,
    time_core_frozen,
    fully_frozen,
    dr=1000,
    dt=timestep
)

print(f"Imilac depth: {im_depth}; Imilac timing: {im_string_result}")
print(f"Esquel depth: {esq_depth}; Esquel timing: {esq_string_result}")
```

Out:

```
Imilac depth: 38.0; Imilac timing: Core has not started solidifying yet
Esquel depth: 45.0; Esquel timing: Core has started solidifying
```

If you need to save the meteorite results, they can be saved to a dictionary which can then be passed to the *load_plot_save.save_params_and_results*. This allows for any number of meteorites to be analysed and only the relevant data stored:

```
meteorite_results_dict = {'Esq results':
                              {'depth': esq_depth,
                               'text result': esq_string_result},
                          'Im results':
                              {'depth': im_depth,
                               'text result': im_string_result,
                               'critical radius': im_Critical_Radius}}
```

To get an overview of the cooling history of the body, it's very useful to plot the temperatures and cooling rates as a heatmap through time. In order to plot the results, we need to define a figure height and width, then call *load_plot_save.plot_temperature_history()*, *load_plot_save.plot_coolingrate_history()* or *load_plot_save.two_in_one()*. These functions convert the cooling rate from K/timestep to K/Myr to make the results more human-readable.

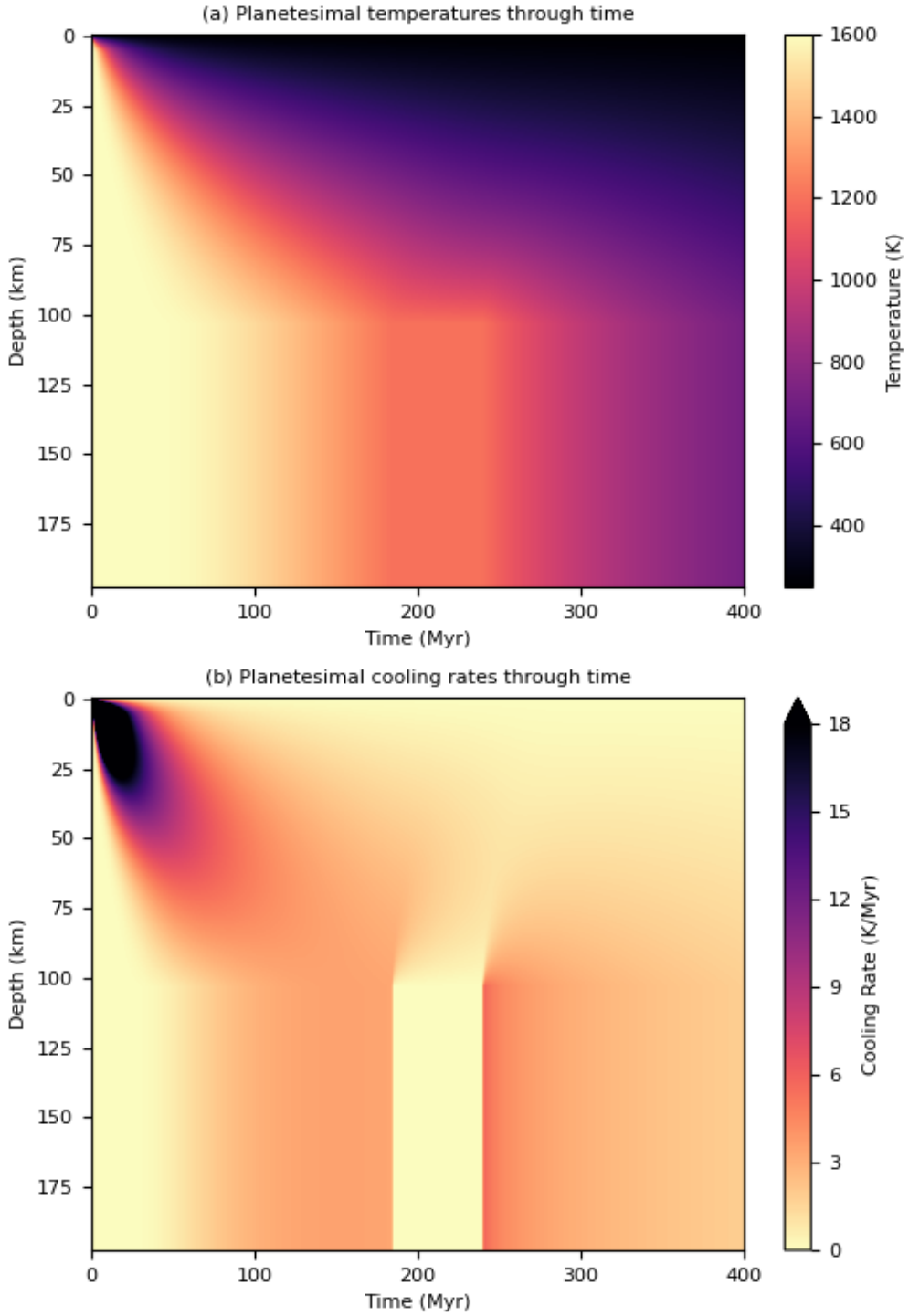
```
fig_w = 6
fig_h = 9

pytesimal.load_plot_save.two_in_one(
    fig_w,
    fig_h,
```

(continues on next page)

(continued from previous page)

```
    mantle_temperature_array,
    core_temperature_array,
    mantle_cooling_rates,
    core_cooling_rates,
    timestep=timestep)
```


(a) Planetesimal temperatures through time
Depth (km)
Time (Myr)
Temperature (K)
0
25
50
75
100
125
150
175
0
100
200
300
400
1600
1400
1200
1000
800
600
400
(b) Planetesimal cooling rates through time
Depth (km)
Time (Myr)
Cooling Rate (K/Myr)
18
15
12
9
6
3
0

Total running time of the script: (0 minutes 37.896 seconds)

CHAPTER

NINE

API REFERENCE

9.1 pytesimal.analysis module

Analyse the results of the conductive cooling model for a planetesimal.

This module contains functions to calculate the cooling rates of meteorites based on the empirical relations suggested by Yang et al. (2010); see full references in Murphy Quinlan et al. (2021). It also contains functions to analyse the temperature arrays produced by the *pytesimal.numerical_methods* module, allowing estimation of the depth of genesis of pallasite meteorites, the relative timing of paleomagnetic recording in meteorites and core dynamo action, and calculation of cooling rates in the mantle of the planetesimal through time.

pytesimal.analysis.**cooling_rate**(*temperature_array*, *timestep*)
Calculate an array of cooling rates from temperature array.

pytesimal.analysis.**cooling_rate_cloudyzone_diameter**(*d*)
Cooling rate calculated using cloudy zone particle diameter in nm.

Constants from Yang et al., 2010; obtained by comparing cz particles and tetrataenite bandwidth to modelled Ni diffusion in kamacite and taenite.

Parameters **d** (*float*) – Cloudy zone particle size in nm.

Returns **cz_rate** – The cooling rate in K/Myr.

Return type float

pytesimal.analysis.**cooling_rate_tetra_width**(*tw*)
Cooling rate calculated using tetrataenite bandwidth in nm.

Constants from Yang et al., 2010; obtained by comparing cz particles and tetrataenite bandwidth to modelled Ni diffusion in kamacite and taenite.

Parameters **tw** (*float*) – Tetrataenite bandwidth in nm.

Returns **t_rate** – The cooling rate in K/Myr.

Return type float

pytesimal.analysis.**cooling_rate_to_seconds**(*cooling_rate*)
Convert cooling rates to seconds.

Parameters **cooling_rate** (*float*) – The cooling rate in K/Myr.

Returns **new_cooling_rate** – The cooling rate in K/s.

Return type float

pytesimal.analysis.**core_freezing**(*coretemp*, *max_time*, *times*, *latent*, *temp_core_melting*, *timestep=100000000000.0*)
Calculate when the core starts and finishes solidifying.

Takes core temperature and returns boolean array of when the core is below the freezing/melting temperature.

Parameters

- **coretemp** (*numpy.ndarray*) – Array of temperatures in the core, in Kelvin.
- **max_time** (*float*) – Length of time the model runs for, in seconds.
- **times** (*numpy.ndarray*) – Array from 0 to the max time +0.5* the timestep, with a spacing equal to the timestep.
- **latent** (*list*) – List of total latent heat extracted since core began freezing, at each timestep.
- **temp_core_melting** (*float*) – Melting point of core material, in Kelvin.
- **timestep** (*float, default 1e11*) – Discretisation timestep in seconds.

Returns

- **core_frozen** (*boolean array where temperature <= 1200 K*)
- **times_frozen** (*array of indices of times where the temp <= 1200 K*)
- **time_core_frozen** (*when the core starts to freeze, in seconds*)
- **fully_frozen** (*when the core finished freezing, in seconds*)

pytesimal.analysis.**meteorite_depth_and_timing**(*CR*, *temperatures*, *dT_by_dt*, *radii*, *r_planet*, *core_size_factor*, *time_core_frozen*, *fully_frozen*, *dr=1000.0*, *dt=100000000000.0*)

Find depth of genesis given the cooling rate.

Function finds the depth, given the cooling rate, and checks if the 593K contour crosses this depth during core solidification, implying whether or not the meteorite is expected to record core dynamo action.

Parameters

- **CR** (*float*) – cooling rate of meteorite, in K/s.
- **temperatures** (*numpy.ndarray*) – Array of mantle temperatures, in Kelvin.
- **dT_by_dt** (*numpy.ndarray*) – Array of mantle cooling rates, in K/dt.
- **radii** (*numpy.ndarray*) – Mantle radii spaced by *dr*, in m.
- **r_planet** (*float*) – Planetesimal radius, in m.
- **core_size_factor** (*float, <1*) – Radius of the core, expressed as a fraction of *r_planet*.
- **time_core_frozen** (*float*) – The time the core begins to freeze, in dt.
- **fully_frozen** (*float*) – The time the core is fully frozen, in dt.
- **dr** (*float, default 1000.0*) – Radial step for numerical discretisation, in m.

Returns

- **depth** (*float*) – Depth of genesis of meteorite, in km.
- **string** (*string*) – Relative timing of tetrataenite formation and core crystallisation, in a string format
- **time_core_frozen** (*float*) – The time the core begins to freeze, in dt.
- **Time_of_Crossing** (*float*) – When the meteorite cools through tetrataenite formation temperature, in dt.
- **Critical_Radius** (*float*) – Depth of meteorite genesis given as radius value, in m.

9.2 pytesimal.core_function module

Create and track the temperature in the planetesimal core.

This module allows the user to track the temperature of a simple isothermal eutectic core and calculate the change in temperature in the core over a timestep based on the heat extracted across the core-mantle boundary.

The class *IsothermalEutecticCore* allows a core object to be instantiated. This core object keeps track of its temperature as it cools, and this temperature history can be called at any time in the form of a 1D timeseries or cast across the core radius (as the core is isothermal).

Classes: IsothermalEutecticCore

Notes

The core object can be replaced with a more complex core that interacts with the mantle in the same way (by extracting energy in Watts across the CMB over a timestep and providing a resulting boundary temperature).

`class pytesimal.core_function.IsothermalEutecticCore(`*initial_temperature*, *melting_temperature*, *outer_r*, *inner_r*, *rho*, *cp*, *core_latent_heat*, *lat=0*`)`

Bases: `object`

Core class to represent and manipulate core temperature and latent heat.

`initial_temperature`
Initial uniform temperature of the core, in K.

Type float

`melting_temperature`
Temperature at which core crystallisation initiates, in K.

Type float

`outer_r`
Outer core radius, in m.

Type float

`inner_r`
Inner core radius, not used by this simple implementation of the core (set to zero), but included so that more complex core models can be coupled to the mantle discretisation function.

Type float

`rho`
Core density, kg m^-3.

Type float

`cp`
Core heat capacity, J kg^-1 K^-1.

Type float

`core_latent_heat`
Latent heat of crystallisation of the core, used to calculate time for core to solidify fully, J kg^-1.

Type float

lat
Tracks latent heat of core, always initially zero in current implementation but included for forward compatibility with a coupled model where core has already cooled by some degree, in J kg^-1.

Type float, optional

__init__(*initial_temperature*, *melting_temperature*, *outer_r*, *inner_r*, *rho*, *cp*, *core_latent_heat*, *lat=0*)
Create a new core with temperature and latent heat.

extract_heat(*power*, *timestep*)
Extract heat (in W) across the core-mantle boundary

Given power (W) and timestep (s), update the *boundary_temperature* and *temperature* to reflect the associated cooling. If *temperature* is equal or less than the melting temperature of the core material, then the core begins to freeze and *temperature* does not change. Instead, latent heat is tracked (*latent*). Once *latent* is greater or equal to the maximum latent heat of the core, the core is fully frozen and begins to cool again as before.

Parameters

- **power** (*float*) – Heat extracted across the CMB in Watts.
- **timestep** (*float*) – The time over which the heat is extracted (in s).

temperature_array_1D()
Return a time-series of core boundary temperatures

Returns **temp_array** – Time series of *boundary_temperature*, in K.

Return type numpy.ndarray

temperature_array_2D(*coretemp_array*)
Cast the core boundary temperatures to an array of radii in time

Parameters **coretemp_array** (*numpy.ndarray*) – Array of zeros to be filled wth core temperature history.

Returns **coretemp_array** – Array of core temperature history, in K.

Return type numpy.ndarray

9.3 pytesimal.load_plot_save module

Load Plot and Save Module

This module contains functions to load parameters and results from files, to plot results either following a model run or from file, and to save parameters and results to file following a model run.

Example

A directory and parameter file can be quickly generated:

```
folder = 'path/to/folder'
filename = 'example_param_file.txt'
filepath=f'{folder}/{filename}'

check_folder_exists(folder)
make_default_param_file(filepath=filepath)
```

This parameters file in json format can then be opened, edited, renamed or moved, and loaded in to set parameter values for a model run.

Notes

Depending on usage, some functions take a *folder* and *filename* argument and create an absolute path with these to save or load a file, while some take a full filepath. Please check which argument is required.

pytesimal.load_plot_save.**check_folder_exists**(*folder*)
Check directory exists and make directory if not.

pytesimal.load_plot_save.**get_million_years_formatters**(*timestep*, *maxtime*)
Return a matplotlib formatter.

Creates two matplotlib formatters, one to go from timesteps to myrs and one to go from cooling rate per timestep to cooling rate per million years.

Parameters

- **timestep** (*float*) – Numerical timestep, in s.
- **maxtime** (*float*) – Total time for model run, in s.

pytesimal.load_plot_save.**load_params_from_file**(*filepath='example_params.txt'*)
Load parameters from a json file and return variable values.

Parameters **filepath** (*str*) – Absolute or relative path to load the json parameter file from.

Returns

- **run_ID** (*str*) – Identifier for the specific model run.
- **folder** (*str*) – Absolute path to directory where file is to be saved. Existence of the directory can be checked with the *check_folder_exists()* function.
- **timestep** (*float*) – The timestep used in numerical method, in s.
- **r_planet** (*float*) – The radius of the planet in m.
- **core_size_factor** (*float*) – The core size as a fraction of the total planet radius.
- **reg_fraction** (*float*) – The regolith thickness as a fraction of the total planet radius.
- **max_time** (*float*) – The total run-time of the model, in millions of years (Myr).
- **temp_core_melting** (*float*) – The melting temperature of the core, in K.
- **mantle_heat_cap_value** (*float*) – The heat capacity of mantle material, in J kg^-1 K^-1.
- **mantle_density_value** (*float*) – The density of mantle material, in kg m^-3.
- **mantle_conductivity_value** (*float*) – The conductivity of the mantle, in W m^-1 K^-1.
- **core_cp** (*float*) – The heat capacity of the core, in J kg^-1 K^-1.
- **core_density** (*float*) – The density of the core, in kg m^-3.
- **temp_init** (*float, list, numpy array*) – The initial temperature of the body, in K.
- **temp_surface** (*float*) – The surface temperature of the planet, in K.
- **core_temp_init** (*float*) – The initial temperature of the core, in K.
- **core_latent_heat** (*float*) – The latent heat of crystallisation of the core, in J kg^-1.
- **kappa_reg** (*float*) – The regolith constant diffusivity, m^2 s^-1.

- **dr** (*float*) – The radial step used in the numerical model, in m.
- **cond_constant** (*str*) – Flag of *y* or *n* to specify if mantle conductivity is constant.
- **density_constant** (*str*) – Flag of *y* or *n* to specify if mantle density is constant.
- **heat_cap_constant** (*str*) – Flag of *y* or *n* to specify if mantle heat capacity is constant.

pytesimal.load_plot_save.**make_default_param_file**(*filepath='example_params.txt'*)

Save an example parameter json file with default parameters.

Save a dictionary of default parameters as a human-readable json txt file. This example file can then be used as an input file as-is, or can be edited in order to modify the model set up. References for the example values can be found in Murphy Quinlan et al. (2021).

Murphy Quinlan, M., Walker, A. M., Davies, C. J., Mound, J. E., Müller, T., & Harvey, J. (2021). The conductive cooling of planetesimals with temperature-dependent properties. Journal of Geophysical Research: Planets, 126, e2020JE006726. https://doi.org/10.1029/2020JE006726

Parameters **filepath** (*str, optional*) – Absolute or relative path to save the json parameter file to.

pytesimal.load_plot_save.**plot_coolingrate_history**(*dT_by_dt*, *dT_by_dt_core*, *timestep*, *maxtime*, *ax=None*, *fig=None*, *savefile=None*, *fig_w=8*, *fig_h=6*, *show=True*)

Generate a heat map of cooling rate vs time.

Generate a heat map of cooling rate vs time, with the colormap showing variation in cooling rate.

Input cooling rate in a n-steps by n-depths array *dT_by_dt* for the mantle and n-steps by n-depths array *dT_by_dt_core* for the core. *timestep* is the length of a timestep and *maxtime* is the maximum time (both in seconds).

Optional arguments fig and ax can be set to plot on existing matplotlib figure and axis objects. Passing a string via outfile causes the figure to be saved as an image in a file.

pytesimal.load_plot_save.**plot_temperature_history**(*temperatures*, *coretemp*, *timestep*, *maxtime*, *ax=None*, *fig=None*, *savefile=None*, *fig_w=8*, *fig_h=6*, *show=True*)

Generate a heat map of depth vs time; colormap shows variation in temp.

Input temperature in a n-steps by n-depths array *temperatures* for the mantle and n-steps by n-depths array *coretemp* for the core. *timestep* is the length of a timestep and *maxtime* is the maximum time (both in seconds).

Optional arguments *fig* and *ax* can be set to plot on existing matplotlib figure and axis objects. Passing a string via outfile causes the figure to be saved as an image in a file.

pytesimal.load_plot_save.**read_datafile**(*filepath*)

Read the contents of a model run into numpy arrays.

Reads the content of the numpy 'npz' data file representing a model run and returns arrays of the mantle temperature, core temperature, and cooling rates of the mantle and core.

Parameters **filepath** (*str*) – Location of .npz data file, including file name and npz suffix.

Returns

- **temperatures** (*numpy.ndarray*) – Array filled with mantle temperatures, in K.
- **coretemp** (*numpy.ndarray*) – Array filled with core temperatures, in K.
- **dT_by_dt** (*numpy.ndarray*) – Array filled with mantle cooling rates, in K/dt.
- **dT_by_dt_core** (*numpy.ndarray*) – Array filled with core cooling rates, in K/dt.

pytesimal.load_plot_save.**read_datafile_with_latent**(*filepath*)

Read contents of a model run into numpy arrays, including latent heat.

Reads the content of the numpy 'npz' data file representing a model run and returns arrays of the mantle temperature, core temperature, cooling rates of the mantle and core, and the number of timesteps the core was crystallising for.

Parameters **filepath** (*str*) – Location of .npz data file, including file name and npz suffix.

Returns

- **temperatures** (*numpy.ndarray*) – Array filled with mantle temperatures, in K.
- **coretemp** (*numpy.ndarray*) – Array filled with core temperatures, in K.
- **dT_by_dt** (*numpy.ndarray*) – Array filled with mantle cooling rates, in K/dt.
- **dT_by_dt_core** (*numpy.ndarray*) – Array filled with core cooling rates, in K/dt.
- **latent_array** (*numpy.ndarray*) – Array filled with latent heat of the core, as it crystallises, J kg^-1.

pytesimal.load_plot_save.**save_params_and_results**(*result_filename*, *run_ID*, *folder*, *timestep*, *r_planet*, *core_size_factor*, *reg_fraction*, *max_time*, *temp_core_melting*, *mantle_heat_cap_value*, *mantle_density_value*, *mantle_conductivity_value*, *core_cp*, *core_density*, *temp_init*, *temp_surface*, *core_temp_init*, *core_latent_heat*, *kappa_reg*, *dr*, *cond_constant*, *density_constant*, *heat_cap_constant*, *time_core_frozen*, *fully_frozen*, *meteorite_results='None given'*, *latent_list_len=0*)

Save parameters and results from model run to a json file.

Save the parameters used, core crystallisation timing, and meteorite results to a json file. The resulting file can also be read in as a parameter file to reproduce the same modelling run.

Meteorite results can be excluded, or can be passed in as a string, value, or dictionary of results.

Parameters

- **result_filename** (*str*) – Filename without file suffix; .txt will be appended when the file is saved.
- **run_ID** (*str*) – Identifier for the specific model run.
- **folder** (*str*) – Absolute path to directory where file is to be saved. Existence of the directory can be checked with the *check_folder_exists()* function.
- **timestep** (*float*) – The timestep used in numerical method, in s.
- **r_planet** (*float*) – The radius of the planet in m.
- **core_size_factor** (*float*) – The core size as a fraction of the total planet radius.
- **reg_fraction** (*float*) – The regolith thickness as a fraction of the total planet radius.
- **max_time** (*float*) – The total run-time of the model, in millions of years (Myr).
- **temp_core_melting** (*float*) – The melting temperature of the core, in K.
- **mantle_heat_cap_value** (*float*) – The heat capacity of mantle material, in J kg^-1 K^-1.
- **mantle_density_value** (*float*) – The density of mantle material, in kg m^-3.
- **mantle_conductivity_value** (*float*) – The conductivity of the mantle, in W m^-1 K^-1.

- **core_cp** (*float*) – The heat capacity of the core, in J kg^-1 K^-1.
- **core_density** (*float*) – The density of the core, in kg m^-3.
- **temp_init** (*float, list, numpy array*) – The initial temperature of the body, in K.
- **temp_surface** (*float*) – The surface temperature of the planet, in K.
- **core_temp_init** (*float*) – The initial temperature of the core, in K.
- **core_latent_heat** (*float*) – The latent heat of crystallisation of the core, in J kg^-1.
- **kappa_reg** (*float*) – The regolith constant diffusivity, m^2 s^-1.
- **dr** (*float*) – The radial step used in the numerical model, in m.
- **cond_constant** (*str*) – Flag of *y* or *n* to specify if mantle conductivity is constant.
- **density_constant** (*str*) – Flag of *y* or *n* to specify if mantle density is constant.
- **heat_cap_constant** (*str*) – Flag of *y* or *n* to specify if mantle heat capacity is constant.
- **time_core_frozen** (*float*) – Time when the core begins to freeze in Myr.
- **fully_frozen** (*float*) – Time when the core finishes freezing, in Myr.
- **meteorite_results** (*dict, str, float, list, optional*) – Depth and timing results of meteorites, can be passed as a dictionary of results for different samples, as a list of results, or as a single result in the form of a float or string.
- **latent_list_len** (*float, optional*) – The length of the latent heat list, needed for further analysis of core crystallisation duration at a later point.

Returns

Return type File is saved to *folder/result_filename*.txt in the json format.

pytesimal.load_plot_save.**save_result_arrays**(*result_filename*, *folder*, *mantle_temperature_array*, *core_temperature_array*, *mantle_cooling_rates*, *core_cooling_rates*, *latent=[]*)

Save results as a compressed Numpy array (npz).

Result arrays of temperatures and cooling rates for both the mantle and the core (numpy arrays) are saved to a specified file.

Parameters

- **result_filename** (*str*) – Filename without file suffix; .npz will be appended when the file is saved.
- **folder** (*str*) – Absolute path to directory where file is to be saved. Existence of the directory can be checked with the *check_folder_exists()* function.
- **mantle_temperature_array** (*numpy.ndarray*) – Temperatures in the mantle for all radii through time, in K.
- **core_temperature_array** (*numpy.ndarray*) – Temperatures in the core through time, in K.
- **mantle_cooling_rates** (*numpy.ndarray*) – Cooling rates in the mantle for all radii through time, in K/dt.
- **core_cooling_rates** (*numpy.ndarray*) – Cooling rates in the core through time, in K/dt.
- **latent** (*list, optional*) – List of latent heat values for the core; needed to calculate timing of core crystallisation, in J kg^-1.

Returns

- File is saved to *folder*/*result_filename*.npz in the compressed Numpy
- *array format.*

pytesimal.load_plot_save.**two_in_one**(*fig_w*, *fig_h*, *temperatures*, *coretemp*, *dT_by_dt*, *dT_by_dt_core*, *savefile=None*, *timestep=100000000000.0*)

Return a heat map of depth vs time; colormap shows variation in temp.

Change save="n" to save="y" when function is called to produce a png image named after the data filename.

9.4 pytesimal.mantle_properties module

Define mantle properties as constant or temperature-dependent.

Set mantle conductivity, density and heat capacity as constant values or define them as functions of temperature. The *MantleProperties* class has methods which define constant values for mantle properties, which can then individually be overridden by the *VariableConductivity*, *VariableDensity* and *VariableHeatCapacity* subclasses. The *VariableConductivity* subclass also contains a method to calculate the derivative of the conductivity with respect to time.

The functions used for variable properties are based on experimental results and mineral physics theory, discussed in Murphy Quinlan et al. (2021) and the references therein.

class pytesimal.mantle_properties.**MantleProperties**(*rho=3341.0*, *cp=819.0*, *k=3.0*)

Bases: object

Mantle properties class to define thermal properties of mantle material.

The value of conductivity, heat capacity or density can be called using the *get* methods, and can be changed using the *set* methods. They can be overridden with temperature-dependent functions using subclasses for each individual property.

Temperature and pressure are optional arguments for the *get* methods; these are not used when the values are temperature-independent but allow for easy insertion of temperature or pressure dependent functions into pre-existing code with minimal changes.

rho

The density of the mantle material (constant), in kg m^-3.

Type float, default 3341.0

cp

The heat capacity of mantle material (constant), in J kg^-1 K^-1.

Type float, default 819.0

k

The conductivity of mantle material (constant), in W m^-1 K^-1.

Type float, default 3.0

__init__(*rho=3341.0*, *cp=819.0*, *k=3.0*)

Initialise mantle properties.

getcp(*T=295*, *P=0.1*)

Get heat capacity.

getdkdT(*T=295*, *P=0.1*)

Get gradient of conductivity.

getk(*T=295*, *P=0.1*)
Get conductivity.

getkappa()
Get diffusivity.

getrho(*T=295*, *P=0.1*)
Get density.

setcp(*value*)
Set heat capacity.

setk(*value*)
Set conductivity.

setrho(*value*)
Set density.

class pytesimal.mantle_properties.**VariableConductivity**(*rho=3341.0*, *cp=819.0*, *k=3.0*)
Bases: pytesimal.mantle_properties.MantleProperties

Make conductivity T-dependent.

getdkdT(*T=295*)
Get derivative of conductivity with respect to temperature.

getk(*T=295*, *P=0.1*)
Get conductivity.

class pytesimal.mantle_properties.**VariableDensity**(*rho=3341.0*, *cp=819.0*, *k=3.0*)
Bases: pytesimal.mantle_properties.MantleProperties

Make density T-dependent.

getrho(*T=295.0*)
Get density.

class pytesimal.mantle_properties.**VariableHeatCapacity**(*rho=3341.0*, *cp=819.0*, *k=3.0*)
Bases: pytesimal.mantle_properties.MantleProperties

Make heat capacity T-dependent.

getcp(*T=295*)
Get heat capacity.

pytesimal.mantle_properties.**set_up_mantle_properties**(*cond_constant='y'*, *density_constant='y'*, *heat_cap_constant='y'*, *mantle_density=3341.0*, *mantle_heat_capacity=819.0*, *mantle_conductivity=3.0*)

Define mantle properties quickly

A quick set-up function that can read parameters and flags from loaded parameter files to set up mantle properties.

Parameters

- **cond_constant** (*str, default 'y'*) – Flag to define if conductivity is constant in temperature or not. Default *'y'* results in constant conductivity, while any other string produces variable conductivity.
- **density_constant** (*str, default 'y'*) – Flag to define if density is constant in temperature or not. Default *'y'* results in constant density, while any other string produces variable density.

- **heat_cap_constant** (*str, default 'y'*) – Flag to define if heat capacity is constant in temperature or not. Default *'y'* results in constant heat capacity, while any other string produces variable heat capacity.
- **mantle_density** (*float, default 3341.0*) – Constant value for mantle density, in kg m^-3.
- **mantle_heat_capacity** (*float, default 819.0*) – Constant value for mantle heat capacity, in J kg^-1 K^-1.
- **mantle_conductivity** (*float*) – Constant value for mantle conductivity, in W m^-1 K^-1.

Returns

- **conductivity** (*object*) – Conductivity object, with constant or temperature dependent value
- **heat_capacity** (*object*) – Heat capacity object, with constant or temperature dependent value
- **density** (*object*) – Density object, with constant or temperature dependent value

9.5 pytesimal.numerical_methods module

Forward-Time Central-Space (FTCS) discretisation for the mantle.

Set up boundary conditions, calculate the heat extracted across the core-mantle boundary in a timestep, and numerically discretise the conductive cooling of the mantle of a planetesimal. This module also contains functions to calculate the thermal diffusivity of a material form the thermal conductivity, heat capacity and the density, and to check whether the diffusivity, timestep and radial discretisation meet Von Neumann stability criteria.

class pytesimal.numerical_methods.**EnergyExtractedAcrossCMB**(*outer_r*, *timestep*, *radius_step*)

Bases: object

Class to calculate the energy extracted across the cmb in one timestep.

outer_r
Core radius, in m.

Type float

timestep
Time over which heat is extracted, in s.

Type float

radius_step
Radial step for numerical discretisation, in m.

Type float

__init__(*outer_r*, *timestep*, *radius_step*)
Initialize self. See help(type(self)) for accurate signature.

energy_extracted(*mantle_temperatures*, *i*, *k*)
Calculate energy extracted in one timestep

power(*mantle_temperatures*, *i*, *k*)
Calculate heat (power) extracted in one timestep

pytesimal.numerical_methods.**calculate_diffusivity**(*conductivity*, *heat_capacity*, *density*)
Calculate diffusivity from conductivity, heat capacity and density.

Returns a value for diffusivity at a certain temperature, given float values for conductivity, heat capacity and density.

Parameters

- **conductivity** (*float*) – Thermal conductivity of the material, in W m^-1 K^-1.
- **heat_capacity** (*float*) – Heat capacity of the material, in J kg^-1 K^-1.
- **density** (*float*) – Density of the material, in kg m^-3.

Returns **diffusivity** – The calculated diffusivity, in m^2 s^-1.

Return type float

pytesimal.numerical_methods.**check_stability**(*max_diffusivity*, *timestep*, *dr*)

Check adherence to Von Neumann stability criteria.

Use the maximum diffusivity of the system to return the most restrictive criteria. Diffusivity can be calculated using the *calculate_diffusivity* function, with max diffusivity where conductivity is maximised and density and heat capacity are minimised.

Parameters

- **max_diffusivity** (*float*) – The highest diffusivity of the system to impose the most restrictive conditions, , in m^2 s^-1.
- **timestep** (*float*) – The timestep used for the numerical scheme, in s.
- **dr** (*float*) – The radial step used for the numerical scheme, in m.

Returns **result** – Boolean, True if parameters pass stability criteria and false if they fail.

Return type bool

pytesimal.numerical_methods.**cmb_dirichlet_bc**(*temperatures*, *core_boundary_temperature*, *i*)

Set a fixed temperature boundary condition at the base of the mantle.

Parameters

- **temperatures** (*numpy.ndarray*) – Numpy array of mantle temperatures to apply condition to, in K.
- **core_boundary_temperature** (*float*) – The temperature at the core mantle boundary, in K.
- **i** (*int*) – Index along time axis where condition is to be set.

Returns **temperatures** – Temperature array with condition applied, in K.

Return type numpy.ndarray

pytesimal.numerical_methods.**cmb_neumann_bc**(*temperatures*, *core_boundary_temperature*, *i*)

Set a zero flux boundary condition at the base of the mantle

Note that core radius must be set to zero for this to approximate the analytical solution of a conductively cooling sphere or to model an undifferentiated meteorite parent body. Sets zero flux at base of the mantle by approximating dT/dr using forward differences and finding the necessary temperature. See eq. 6.31 of http://folk.ntnu.no/leifh/teaching/tkt4140/._main056.html

Parameters

- **temperatures** (*numpy.ndarray*) – Numpy array of mantle temperatures to apply condition to, in K.

- **core_boundary_temperature** (*float*) – The temperature at the core mantle boundary; this is not used by this boundary condition but inclusion allows functions to be easily swapped. In K.
- **i** (*int*) – Index along time axis where condition is to be set.

Returns **temperatures** – Temperature array with condition applied, in K.

Return type numpy.ndarray

pytesimal.numerical_methods.**discretisation**(*core_values*, *latent*, *temp_init*, *core_temp_init*, *top_mantle_bc*, *bottom_mantle_bc*, *temp_surface*, *temperatures*, *dr*, *coretemp_array*, *timestep*, *r_core*, *radii*, *times*, *where_regolith*, *kappa_reg*, *cond*, *heatcap*, *dens*, *non_lin_term='y'*)

Finite difference solver with variable k.

Uses variable heat capacity, conductivity, density as required.

Uses diffusivity for regolith layer.

Parameters

- **core_values** (*core object*) – An object that represents the state of the layer inside the current layer (normally a metallic core). The object must provide one method and two attributes. The method extract_heat(power, timestep) is called on each timestep and represents the amount of heat that is lost from the the inner layer to the present layer (power, in W) over an amount of time (timestep, in s). The attribute temperature gives the temperature at the top of the inner layer and this is used (after calling extract_heat) as input to the basal boundary condition calculation. The attribute latent reports any latent heat released by freezing and this is not explicitly used in the evaluation of mantle temperatures.
- **latent** (*list*) – Empty list (unless coupling two models) of latent heat extracted from the core.
- **temp_init** (*float, numpy.ndarray*) – The initial temperature (in K) of the mantle with float implying initial homogeneous temperature distribution.
- **core_temp_init** (*float, numpy.ndarray*) – Initial temperature of the core; current core object is isothermal so only accepts float but more complex core models could track the temperature distribution in the core. In K.
- **top_mantle_bc** (*callable*) – Calleable function that defines the boundary condition at the planetesimal surface. The calling signature is top_mantle_bc(temperatures, surface_temperature, timestep_index) where temperatures is the temperatures array to be updated with the boundary condition, core_boundary_temperature is the temperature (that may be involved in the calculation) and timestep_index is the column index of the current timestep. The function must return an updated temperatures array. See surface_dirichlet_bc for an example.
- **bottom_mantle_bc** (*callable*) – Calleable function that defines the boundary condition at the base of the planetesimal mantle. The calling signature is bottom_mantle_bc(temperatures, core_boundary_temperature, timestep_index) where temperatures is the temperatures array to be updated with the boundary condition, core_boundary_temperature is the temperature (that may be involved in the calculation) and timestep_index is the column index of the current timestep. The function must return an updated temperatures array. See cmb_neumann_bc for an example.
- **temp_surface** (*float*) – Temperature at the surface of the planetesimal, in K.
- **temperatures** (*numpy.ndarray*) – Numpy array to fill with mantle temperatures in K.

- **dr** (*float*) – Radial step for numerical discretisation, in m.
- **coretemp_array** (*numpy.ndarray*) – Numpy array to fill with core temperatures
- **timestep** (*float*) – Timestep for numerical discretisation, in s.
- **r_core** (*float*) – Radius of the core in m.
- **radii** (*numpy.ndarray*) – Numpy array of radii values in the mantle, with spacing defined by *dr*, in m.
- **times** (*numpy.ndarray*) – Numpy array of time values in s, up to the maximum time, with spacing controlled by *timestep*, in s.
- **where_regolith** (*numpy.ndarray*) – Boolean array recording presence of regolith.
- **kappa_reg** (*float*) – Constant diffusivity of the regolith, in m^2 s^-1.
- **cond** (*callable*) – Callable function or method that defines the mantle conductivity. The calling signature is *cond.getk(temperatures[radial_index, timestep_index])*, where temperatures is the temperatures array, radial_index is the row index of the radius, and timestep_index is the column index of the timestep, that define the value in temperatures at which conductivity should be evaluated. The function must return a value for conductivity in in W m^-1 K^-1.
- **heatcap** (*callable*) – Callable function or method that defines the mantle heat capacity. The calling signature is *heatcap.getcp(temperatures[radial_index, timestep_index])*, where temperatures is the temperatures array, radial_index is the row index of the radius, and timestep_index is the column index of the timestep, that define the value in temperatures at which heat capacity should be evaluated. The function must return a value for heat capacity in J kg^-1 K^-1.
- **dens** (*callable*) – Callable function or method that defines the mantle density. The calling signature is *dens.getrho(temperatures[radial_index, timestep_index])*, where temperatures is the temperatures array, radial_index is the row index of the radius, and timestep_index is the column index of the timestep, that define the value in temperatures at which heat capacity should be evaluated. The function must return a value for density in kg m^-3.
- **non_lin_term** (str, default *'y'*) – Flag to switch off the non-linear term when temperature-dependent conductivity is being used.

Returns

- **temperatures** (*numpy.ndarray*) – Array filled with mantle temperatures, in K.
- **coretemp** (*numpy.ndarray*) – Array filled with core temperatures, in K.
- **latent** (*list*) – List of latent heat values during core crystallisation, in J kg^-1.

pytesimal.numerical_methods.**surface_dirichlet_bc**(*temperatures*, *temp_surface*, *i*)

Set a fixed temperature boundary condition at the planetesimal's surface.

Parameters

- **temperatures** (*numpy.ndarray*) – Numpy array of mantle temperatures to apply condition to, in K.
- **temp_surface** (*float*) – The temperature at the surface boundary, in K.
- **i** (*int*) – Index along time axis where condition is to be set.

Returns **temperatures** – Temperature array with condition applied, in K.

Return type numpy.ndarray

9.6 pytesimal.quick_workflow module

Set up complete runs with a single function call and a parameter file.

This module provides a simple *workflow* function which follows a basic default workflow, with parameters set by an input parameter file, and results saved as a json file and as compressed numpy arrays. These results files can then be loaded and analysed, with meteorite results calculated and the results plotted.

The function takes two arguments, *filename* which is the name of a parameters file to be loaded, without the extension (the extension of the file must be .txt). The location of this input file is given by *folder_path* which should be the relative or absolute path to the location of the parameters file. Within this parameters file, the "folder" field defines the path of the directory where results

pytesimal.quick_workflow.**workflow**(*filename*, *folder_path*)

Run model in full with parameters set by an input file.

Saves a results file (json format, .txt) and a results array file (.npz) to the folder specified in the parameter file. If you want the results to save to the same folder that the parameter file is in, ensure that the field "folder" in the json parameter file is the same as *folder_path*.

Results files will be saved under the *filename* with *_results* and the appropriate file extension appended.

The json (.txt) file contains a list of parameter names and values, while the array file (.npz) contains four different arrays: *mantle_temperature_array* - an array of mantle temperatures in K, *core_temperature_array* - an array of core temperatures in K, *mantle_cooling_rates* - an array of mantle cooling rates in K/dt, *core_cooling_rates* - an array of core cooling rates in K/dt.

Parameters

- **filename** (*str*) – The filename of the parameters file to read in, without file extension. File must be in json format with a .txt extension. It's recommended to generate an example parameters file using the *pytesimal.load_plot_save.make_default_param_file(filepath)* function and edit the default settings in this file
- **folder_path** (*str*) – The absolute path to the directory that holds the parameters file. If the "folder" field of the parameters file == *folder_path*, the results file will be saved alongside the parameters file.

9.7 pytesimal.setup_functions module

Define the geometry of planetesimal and set up required empty arrays.

This module allows the user to set up a basic geometry based on parameters instead of manually defining 'numpy.ndarrays'.

pytesimal.setup_functions.**set_up**(*timestep=100000000000.0*, *r_planet=250000.0*, *core_size_factor=0.5*, *reg_fraction=0.032*, *max_time=400.0*, *dr=1000.0*)

Define the geometry and set up corresponding arrays.

Parameters

- **timestep** (*float, default 1e11*) – A timestep for the numerical discretisation, in s
- **r_planet** (*float, default 250000.0*) – The radius of the planetesimal, in m
- **core_size_factor** (*float, < 1.0, default 0.5*) – The core radius expressed as a fraction of *r_planet*
- **reg_fraction** (*float, <1.0, default 0.032*) – The core thickness expressed as a fraction of *r_planet*

- **max_time** (*float, default 400.0*) – Total time for model to run, in millions of years (Myr)
- **dr** (*float, default 1000.0*) – Radial step for the numerical discretisation, in m

Returns

- **r_core** (*float,*) – Radius of the core, in m
- **radii** (*numpy.ndarray*) – Numpy array of radius values in m for the mantle, with spacing defined by *dr*
- **core_radii** (*numpy.ndarray*) – Numpy array of radius values in m for the core, with spacing defined by *dr*
- **reg_thickness** (*float*) – Regolith thickness in m
- **where_regolith** (*numpy.ndarray*) – Boolean array with location of regolith
- **times** (*numpy.ndarray*) – Numpy array starting at 0 and going to 400 Myr, with timestep controlling the spacing
- **mantle_temperature_array** (*numpy.ndarray*) – Numpy array of zeros to be filled with mantle temperatures in K
- **core_temperature_array** (*numpy.ndarray*) – Numpy array of zeros to be filled with core temperatures in K

CHAPTER

TEN

PYTESIMAL

Pytesimal models the conductive cooling of planetesimals with temperature-dependent material properties.

Pytesimal is a finite difference code to perform numerical models of a conductively cooling planetesimal, both with constant and temperature-dependent properties. It returns a thermal history of the planetesimal, and the estimated timing and depth of pallasite meteorite genesis. The conduction equation is solved numerically using an explicit finite difference scheme, FTCS (Forward-Time Central-Space). FTCS gives first-order convergence in time and second-order in space, and is conditionally stable when applied to the heat equation. In 1D, it must satisfy Von Neumann stability analysis - please see Murphy Quinlan et al. (2021) for more information on choice of time-step.

The code currently recreates the cases described in Murphy Quinlan et al. (2021). References for the default parameters used can be found therein. We plan to extend it and make it more modular in future updates.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

C

D

E

G

I

K