Reaction Diffusion Models - A Practical Introduction

Ulrich G. Wortmann

November 27, 2025

1 A short Jupyter Notebook Introduction

A notebook is a web-based document that mixes **code**, **text**, **plots**, and **data**. The building blocks are **cells**. Below is a quick guide to creating, editing, and running code cells.

1.1 The Notebook Interface

Toolbar Buttons for saving, adding cells, cutting/pasting, etc.

Menu bar More commands (File → Download, Cell → Run. . . , etc.)

Cells Individual blocks that hold code or markdown text.

Kernel The Python (or other language) process that executes your code.

1.2 Adding a Code Cell

1. **Click** the `+` button on the toolbar **or** press `Esc` then `B` (below) / `A` (above) to insert a new cell.
2. By default the new cell is a **code cell** (green border).

1.3 Editing a Code Cell

- **Enter edit mode** – click inside the cell **or** press `Enter` when the cell is selected.
- The cell border turns **green** and you can type Python (or the kernel's language).
- Use normal editor shortcuts (Ctrl-A to select all, Ctrl-Z to undo, etc.).

1.4 Running a Code Cell

- | Run the current cell and move to the next **Shift+Enter** Executes the cell, shows output below, and selects the cell below (or creates a new one if you're at the end).
- Run the current cell and stay **Ctrl+Enter** Executes the cell but keeps the cursor in the same cell.
- Run the current cell and insert a new one below **Alt+Enter** Executes, then adds a fresh cell underneath.

When you run a cell, the kernel prints the result (e.g., numbers, plots, tables) right under the cell. If there's an error, the traceback appears there too.

1.5 Common Editing Shortcuts (while in command mode, `Esc`)

`A` Insert a new cell **above** the selected cell

`B` Insert a new cell **below**

`DD` **(press `D` twice)** Delete the selected cell

`M` Change selected cell to **Markdown** (for formatted text)

`Y` Change selected cell back to **Code**

`↑ / ↓` Move selection up/down

`Shift↑↓` Extend selection to multiple cells

`CtrlS` Save notebook (also click the floppy-disk icon)

Tip: Press `Esc` to ensure you're in **command mode** (blue border) before using these shortcuts.

1.6 Saving & Exporting

- Click the **disk** icon or press **Ctrl+S** frequently.
- When finished, you can download the notebook as `.ipynb` (File → Download as → Notebook) or as a static HTML/PDF for sharing.

1.7 Quick Checklist for First-Time Users

1. **Add** a code cell (`+` or `B`).
2. **Type** some Python, e.g. `print("Hello, Jupyter!")`.
3. **Run** it with **Shift+Enter**.

4. **Observe** the output below.
5. **Edit** any cell by clicking into it and pressing **Enter**.
6. **Save** often (`Ctrl+S`).

That's it! With these basics you can start experimenting, visualizing data, and building interactive analyses—all within the same notebook. Happy coding!

See this link https://www.youtube.com/watch?v=H9Iu49E6Mxs for a more complete intro.

1.8 Checkpoints

Checkpoints allow you to return to a previous version of your notebook.

- **Manually creating a checkpoint**
  1. Click **File → Save and Checkpoint** (or press `Ctrl+S`).
  2. The current notebook becomes the new checkpoint.
- **Reverting to a checkpoint**
  1. Choose **File → Revert to Checkpoint → <timestamp>**.
  2. Jupyter overwrites the notebook with the saved copy.
  3. The notebook shows a banner "Reverted to checkpoint".

It is a good idea to frequently press `Ctrl +S`!

1.8.1 Try me!

- Create a new checkpoint
- Create a new code cell, and type a something like `12 *5`
- execute the code cell to see the result.
- revert to the previous checkpoint

2 Preparing your python session

Before we begin, we need to install some python modules.

2.1 Installing the python modules needed in this session

You need to execute this once per session

```
%pip install fastfd
```

2.2 Import the model and the plotting interface

We now load the following modules:

Module	Purpose
`matplotlib.pyplot`	Create static, animated, and interactive figures.
`pandas`	Manipulate tabular data (DataFrames).
`diff_lib.data_container`	Wrap raw simulation output into a convenient object.
`run_methane.model`	Run the methane diffusion model.

```
import matplotlib.pyplot as plt
import pandas as pd
from diff_lib import data_container
from run_methane import model
```

3 Defining the first model:

The model we employ is derived from a more comprehensive reaction-transport model used in my research. It is not designed for a Jupyter session, so the user interface is somewhat awkward. Most parameters are defined using Python dictionaries that contain key-value pairs. The key is enclosed in quotation marks; the value may be (i) a bare word, interpreted as the name of a variable, (ii) a quoted string, interpreted verbatim, or (iii) a numeric value.

A common source of error in modeling is inconsistent units. Therefore I run all my models in SI base units - meters, seconds, and concentrations expressed in mmol/l. First we have to define a the length of the model, the number of grid points, the porosity, and sedimentation rate. Before continuing, convert the sedimentation value into meters/ky to see if the value make sense:

You can do this easily on a pocket calculator, or by typing the calculation in this notebook cell (see above on how to edit, and execute a cell): Note that `1e-5` stands for 1×10^{-5}.

```
12 *5
```

Next we need to execute this code block to activate these numbers. You can always come back later to to change them (Note, changes affect the entire notebook!).

```
# a few parameters to play with
p = {
    "max_depth": 0.35,  # meters
    "grid_points": 350,
    "phi": 0.65,  # porosity
    "w": 1.5e-10,  # sedimentation rate in m/s
}
```

3.1 Setting boundary conditions

Setting the boundary conditions requires choosing a specification method and assigning appropriate values. For diffusive and advective transport we can use three types of boundary conditions:

1. **Dirichlet** (infinite concentration) – the concentration at the boundary is fixed. For example, the sulfate concentration in seawater remains 28 mmol L^{-1} regardless of reactions in the underlying sediment.
2. **Neumann** – the concentration is unknown but its gradient is zero. This represents a depth where diagenetic reactions have ceased.
3. **Robin** – a mixture of Dirichlet and Neumann. The flux is proportional to the difference between the boundary value and the ambient value.

We will use the following concentration values as upper and lower bounds for our first model.

```
bc = {
    "ch4": [  # species
        "concentration",  # upper bc type
        0,  # upper bc value
        "concentration",  # lower bc type
        0.3,  # lower bc value
        "dissolved",  # phase
        1,  # reaction type 1 = source, -1 = sink
    ],
}
```

3.2 Define reaction rates and a reaction term

For the initial run we use a simple model with no microbial reactions; all reaction terms are set to zero. Although we consider only one species, the code requires a reaction constant for every species.

```
# -------------------------- Reaction constants ------------ *
k = data_container({"ch4": 0})
```

Furthermore, I provide a function that describes which entities react with which and how. Because there is no reaction, the function returns 0.

```
def diagenetic_reactions(a, z, c, k, f):
        """Define microbial reactions. Note that reactions
        are always positive. The sign is set by reaction type
        which can be either a 'source' or a 'sink'
```

```
        """
        f.ch4 = 0

        return f, 0
```

3.3 Run the model and plot the results

Now I am ready to run the model. I call the model code with the parameter list `p`, the boundary conditions `bc`, the reaction rates `k`, and the function that describes the diagenetic reactions. The model returns a data frame `df` with all results.

```
df = model(p, bc, k, diagenetic_reactions)
display(df.head())
```

Next, I create an X-Y graph from the dataframe with this code snippet.

```
fig, ax = plt.subplots() # Create a new plot object
ax.plot(df.c_ch4, df.z) # assign data to X & Y axes
ax.set_xlabel(r"CH$_4$ [mmol/l]") # set labels
ax.set_ylabel("Depth [mbsf]")
ax.invert_yaxis()
fig.tight_layout()
plt.show()
```

BTW, If you feel adventurous, create a checkpoint and edit the above code by dividing the z-coordinate by 10 (`df.z/10`)m and change the y-label to cm.

3.4 Now what does this all mean?

At first sight the results appear trivial, but there are hidden difficulties. By imposing a concentration boundary condition (*Dirichlet* type) we explicitly assume that both the upper and lower boundaries are infinite; that is, the boundary values remain fixed regardless of what happens inside the model. This assumption is often reasonable for the upper boundary, but it is much harder to justify for the lower boundary. In the present example we are effectively stating that there is an infinite supply of methane at the base of the core, which is unlikely to be realistic. And in the absence of any reactions, we get a straight line as we would expect for diffusive mixing.

We could try to resort to a *Neuman* boundary condition where, instead of concentration, we state that there should be no change in concentration, i.e., the gradient of the concentration equals zero. This way we assume that there are no further changes in $[CH_4]$ but now, because there is no methane source below our modeling domain (equally unlikely), and we get no methane at all.

```
bc = {
    "ch4": [  # species
```

```
        "concentration",  # upper bc type
        0,  # upper bc value
        "gradient",  # lower bc type
        0,  # lower bc value
        "dissolved",  # phase
        -1,  # reaction type -1 = source, 1 = sink
    ],
}

df = model(p, bc, k, diagenetic_reactions)
fig, ax = plt.subplots() # Create a new plot object
ax.plot(df.c_ch4, df.z) # assign data to X & Y axes
ax.set_xlabel(r"CH$_4$ [mmol/l]") # set labels
ax.set_ylabel("Depth [mbsf]")
ax.invert_yaxis()
fig.tight_layout()
plt.show()
```

So a key observation is that the boundary conditions determine the model results to a large degree.

4 Adding a methane production term

Microbial reactions often decay with time (depth), but for the sake of simplicity, we will assume a constant reaction term. We do this by modifying the `diagenetic_reactions` function, and then re-running the model.

```
def diagenetic_reactions(a, z, c, k, f):
      f.ch4 = 1e-8  #mmol/l/s
      return f, 0

df = model(p, bc, k, diagenetic_reactions)
fig, ax = plt.subplots() # Create a new plot object
ax.plot(df.c_ch4, df.z) # assign data to X & Y axes
ax.set_xlabel(r"CH$_4$ [mmol/l]") # set labels
ax.set_ylabel("Depth [mbsf]")
ax.invert_yaxis()
fig.tight_layout()
plt.show()
```

This now looks much better, but there are two issues here:

1. the curve bends much earlier than in the Angel et al. paper (something we can try to fix by changing the methane production term)
2. Our curve is artificially forced to be vertical. This is in clear violation of the fact that we produce methane at the bottom of the model.

Now go back to the Angle et al paper, and compare their methane production rate with the above number. You can use this code box for the calculation:

```
12 * 5
```

5 A better model

To avoid the bias introduced by our boundary conditions, we need to extend the modeling domain into a region where we are certain that no methane production occurs, so that the assumption of a zero gradient in $[CH_4]$ is justified. Therefore, we assume that methane production ceases in the last 5 cm above the bottom of the core.

```
def diagenetic_reactions(a, z, c, k, f):
      f.ch4[:] = 1e-8  #mmol/s # treat ch4 a vector
      f.ch4[300:] = 0 # set the last 5 cms to zero

      return f, 0

# run model
df = model(p, bc, k, diagenetic_reactions)

# plot data
fig, ax = plt.subplots() # Create a new plot object
ax.plot(df.c_ch4, df.z) # assign data to X & Y axes
ax.set_xlabel(r"CH$_4$ [mmol/l]") # set labels
ax.set_ylabel("Depth [mbsf]")
ax.invert_yaxis()

# plot f to verify
axt = ax.twiny()
axt.plot(df.f_ch4, df.z, color="C1")
axt.set_xlabel("f [mmol/s]")
fig.tight_layout()
plt.show()
```

6 The problem with advection

Advection comes in two flavors:

1. Downward directed sedimentation creates a constant downward flux. This affects dissolved ions as well as solids. Fast sedimentation rates are able to distort the linear mixing profiles into a concave shape.

2. Upward directed fluid flow. This only affects solutes, but not particles.

This model has presently no support for the second type, but since we have no particles in our model, we can simply change the sign of the sedimentation rate to emulate the effect.

6.1 Case A: Sedimentation rate is fast compared to the microbial reactions.

This is an unrealistic scenario but serves to show how sedimentation rate affects concentration profiles

```
# a few parameters to play with
p = {
        "w": 1.5e-9,  # sedimentation rate in m/s
}

df = model(p, bc, k, diagenetic_reactions)

fig, ax = plt.subplots() # Create a new plot object
ax.plot(df.c_ch4, df.z) # assign data to X & Y axes
ax.set_xlabel(r"CH$_4$ [mmol/l]") # set labels
ax.set_ylabel("Depth [mbsf]")
ax.invert_yaxis()
fig.tight_layout()
plt.show()
```

6.2 Case B: Upward directed fluid flow

To emulate the effect of upward directed fluid flow, we simply change the sign for the sedimentation rate (this is only valid if there are no solids in the model!)

```
# a few parameters to play with
p = {
        "w": -1.5e-9,  # sedimentation rate in m/s
}

df = model(p, bc, k, diagenetic_reactions)

fig, ax = plt.subplots() # Create a new plot object
ax.plot(df.c_ch4, df.z) # assign data to X & Y axes
ax.set_xlabel(r"CH$_4$ [mmol/l]") # set labels
ax.set_ylabel("Depth [mbsf]")
ax.invert_yaxis()
fig.tight_layout()
plt.show()
```

It is evident, that sedimentation/advection has a huge influence on concentration profiles. A specific problems for wetlands that are exposed to episodic flooding, or tidal cycles.