Finite Element Modeling in Geodynamics

Lars Ruepke

Apr 20, 2021

COURSE INFORMATION

Welcome to the website for the course Numerical Fluid Dynamics II (FEM in geodynamics) taught at Kiel University within the Geophysics master. The focus of this course is on learning how we can solve partial differential equations using the Finite Element Method (FEM) in the context of geodynamical modeling. The Finite Element Method is one of the main numerical methods in mechanics and is also used in the popular geodynamics community code ASPECT. We will first learn about the basics before progressing towards writing our own mantle convection code and using it to address various geodynamic problems ranging from subduction to seafloor spreading. If we have time, we will also cover the marker-in-cell method, which becomes increasingly popular in the geodynamics community. All examples will be implemented in MATLAB and/or Python using the FENICS framework.

CHAPTER

ONE

COURSE CONTENT

These are the main topics:

1. Introduction to the finite element method (FEM)
2. Matlab/Python implementation of the FEM in 1D und 2D
3. Stokes-Flow
4. Project work on geodynamical topics
5. Introduction to marker-in-cell method

CHAPTER

TWO

COURSE GOALS

1. Ability to solve geodynamics problems using numerical models.
2. Knowledge of numerical techniques and of how to solve partial differential equations using numerical methods using MATLAB or Python
3. Ability to understand how the popular geodynamics community codes like ASPECT and LaMEM “work”.
4. Preparation to a possible MSc project in geophysical fluid dynamics.

CHAPTER

THREE

COURSE FORMAT

The majority of this course will be spent in front of a computer working on exercises related to FEM and problems in marine geodynamics.

Open access!

Note that the materials for this course are **open to everyone**; the course is, however, taught as an on-site class for registered students at Kiel University.

3.1 Course details

3.1.1 Course schedule

Summer Semester 2021

- Class sessions
  - Wednesdays at 14:00 CET (21.04.2021 - 07.07.2021), zoom link available from instructors
- Semester breaks
  - no scheduled breaks so far

3.1.2 Instructors

Prof. Lars Ruepke

- Email: lruepke@geomar.de
- Institute webpage

Dr. Zhikui Guo

- Email: zguo@geomar.de

3.1.3 Course website

- Public site : https://lruepke.github.io/FEM_lecture/
- University site: https://lms.uni-kiel.de/url/RepositoryEntry/4012113942

3.1.4 Further readings

There are many good online resources on numerical modeling in marine geosciences:

- The CIG website https://geodynamics.org
- Website of the community code ASPECT https://aspect.geodynamics.org
- Marc Spiegelmann's script on numerical methods https://earth.usc.edu/~becker/teaching/557/reading/spiegelman_mmm.pdf

3.1.5 Credits

- Some of the material on finite differences are based on material that **Boris Kaus** initially developed for a modeling workshop at Oslo University.
- Special thanks also to **Ria Fischer**, who helped with a previous version of this lecture.

3.2 Lecture overview

In this lecture we will go through all the technical prerequisites and will explore some simple finite-differences examples. First, we will learn about the the explicit finite differences discretization by solving for the cooling of a hot intrusion into colder country rock. We will discuss the strengths and limitations of explicit methods FDM. Afterwards, we will derive and implement the implicit form, which involves solving a system of equations. Finally, we will learn about different boundary conditions and will test those on the example of how periodic changes in seafloor temperatures propagate into the seafloor.

3.3 Installation guide

We will use Python for learning the basics of the finite element method. Later in the course we will also use FENICS to explore some more advanced problems. Most of the work will be done in Jupyter notebooks. Let's get all of this to work.

3.3.1 Visual Studio Code

We will do a lot of editing of text files and you can use your favorite text editor for this. However, we recommend to use Microsoft's Visual Studio Code.

3.3.2 Python

In case already have a working python environment, you can adapt it for this course (by e.g. creating a new virtual environment). If not, we recommend Miniconda. Follow the miniconda installation instructions and afterwards create a virtual environment for this course. If you are asked to automatically activate the base environment (add it to the system path), chose "no". It's usually a good idea to keep the normal OS python environment intact and only activate a miniconda environment when you need it.

Download and install miniconda

Just follow the installation instructions and keep the default options. We recommend to install python 3. During the installation, choose to **not** add it to your path (that's the default). Adding miniconda/anaconda to your `$PATH` may seem convenient but there are several reason to not do it.

- Python is used by many different tools on your computer, which probably expect that just calling python will use the Python (and additional packages) installed by the operating system. None of these will be available to Miniconda's Python.
- The conda environment we will create contains several binary dependencies and we do not want to interfer with defaults on your system when, e.g. compiling software unrelated to our lecture.

Create a virtual environment

We will create a so-called virtual environment with all the python packages we will use during this class. To not infer with your default python installation, we will do this in a virtual environment. To get started open a terminal with activated base miniconda installation.

Starting python

If you are on Windows, start an Anaconda Powershell Prompt from the start menu.

On MacOS / Linux, open a terminal and type

```
conda activate base
```

You can also do that in the terminal within Visual Studio Code (on MacOS).

Now we are ready to create a virtual environment. We can create it with this command:

```
conda create -n py37_fem_class python=3.7 numpy pandas matplotlib vtk h5py ipython␣
↪scipy ipykernel
```

We are using python 3.7 here (instead of the newest 3.8) because of an incompatibility with vtk. Activate the new environment

```
conda activate py37_fem_class
```

Switching between environments

You can activate and deactivate environments like this:

```
conda activate py37_fem_class
conda deactivate
```

Working with jupyter notebooks

We will do most excercises using jupyter notebooks. A good workflow is to start jupyter labs in working directory.

```
cd "your working directory"
jupyter lab
```

One possible issue is that you need to make sure that your jupyter notebooks use the correct python environment. One way of doing this is to install nb_conda_kernels in your base environment and afterwards registering your virtual environment. This can be done like this:

```
conda activate base
conda install nb_conda_kernels
conda activate py37_fem_class
ipython kernel install --user --name=py37_fem_class
```

Restart you jupyter lab and try to select the correct python kernel.

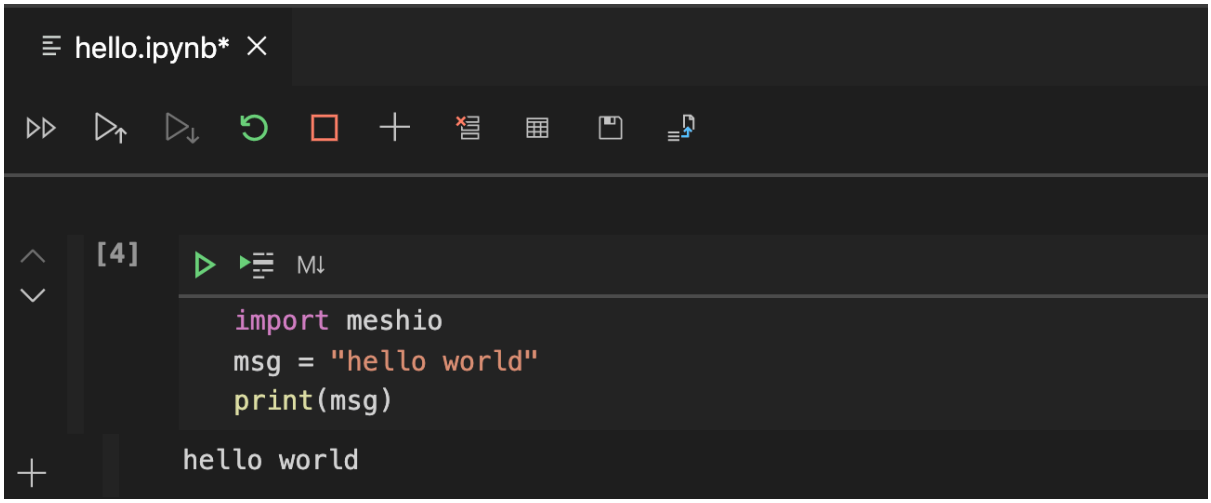
Integration with Visual Studio Code

You will need to install Microsoft's Python extension. Just search for Python under Extensions and chose the one from Microsoft (usually the first option). Finally, you will have to set the Python interpreter. Do this by pushing CMD/CTRL+SHIFT+P. Type Python: Select Interpretor and select our newly created anaconda environment. If it doesn't show up, close and re-open Visual Studio Code.

An alternative is to use jupyter lab; we will use both options.

Tip: Test your installation by doing this:

- choose the right python interpretor STRG/CMD+SHIFT+P
- `code hello.ipynb`
- type in the example code from the figure below
- execute the cell with SHIFT+RETURN



3.4 Introduction Finite Differences

Before progressing towards finite element modeling, we will learn about the Finite Difference Method (FDM), which is a somewhat easier method to solve partial differential equations. We will do so by looking at how heat conduction "works".

3.4.1 Background on conductive heat transport

Temperature is one of the key parameters that control and affect the geological processes that we are exploring during this class. Mantle melting, rock rheology, and metamorphism are all functions of temperature. It is therefore essential that we understand how rocks exchange heat and change their temperature.

One fundamental equation in the analysis of heat transfer is Fourier's law for heat flow:

$$\vec{q} = -k\nabla T = -k \begin{bmatrix} \frac{\partial T}{\partial x} \\ \frac{\partial T}{\partial y} \\ \frac{\partial T}{\partial z} \end{bmatrix} \tag{3.1}$$

It states that heat flow is directed from high to low temperatures (that's where the minus sign comes from) and is proportional to the geothermal gradient. The proportionality constant, k, is the thermal conductivity which has units of W/m/K and describes how well a rock transfers heat. k is a typically a complex function of rock type, porosity, and temperature yet is often simplified to a constant value.

In most cases, we are interested in temperature and not heat flow so that we would like to have an equation that describes the temperature evolution in a rock. We can get this by deriving a conservation law for heat.

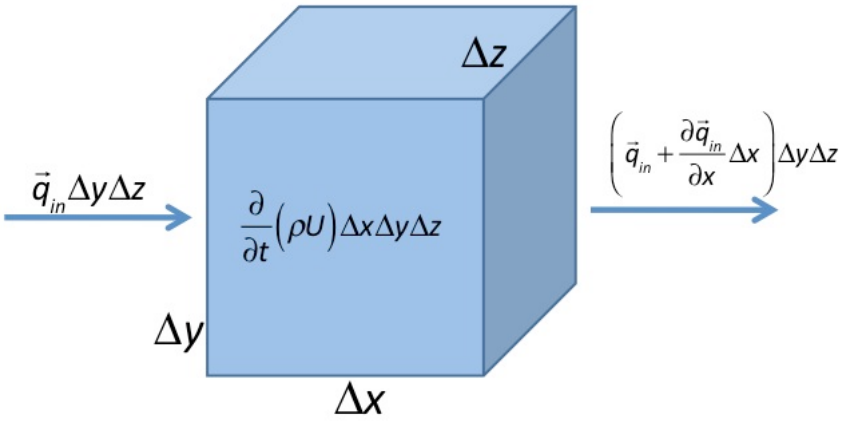


Fig. 3.1: Derivation of the energy equation. Change in internal energy is related to changes in heat fluxes into and out of the box (and a lot of other terms (e.g. advection) that are neglected here).

For simple incompressible cases, changes in internal energy can be expressed as changes in temperature timesdue density and specific heat. The change in internal energy with time can now be written as $\rho c_p \frac{\partial}{\partial t} T \Delta x \Delta y \Delta z$ (ρ is density, c_p specific heat, and T temperature), has units of (J/s), and must be equal to the difference between the heat flow into the box $q_{in}\Delta y\Delta z$ $\left(\frac{W}{mK}\frac{K}{m}m^2 = \frac{J}{s}\right)$ and the heat flow out of the box $\left(q_{in} + \frac{\partial q_{in}}{\partial x}\Delta x\right)\Delta y\Delta z$ (the y and z directions are done in the same way). With these considerations, we can write a conservation equation for energy:

$$\begin{aligned} \rho c_p \frac{\partial T}{\partial t} &= -\frac{\partial q_{in}}{\partial x} = \frac{\partial}{\partial x} k \frac{\partial T}{\partial x} \\ \frac{\partial T}{\partial t} &= \frac{k}{\rho c_p}\frac{\partial^2 T}{\partial x^2} = \kappa \frac{\partial^2 T}{\partial x^2} \end{aligned} \tag{3.2}$$

equation (3.2) is called the heat transfer or heat diffusion equation and is one of the most fundamental equations in Earth Sciences. If the thermal conductivity is constant, we can define a thermal diffusivity, κ, and write the simpler second form of the equation.

Note how the changes in heat flow can be written in terms of a divergence:

$$\rho c_p \frac{\partial T}{\partial t} = -\nabla \cdot (\vec{q}) = \frac{\partial}{\partial x} k \frac{\partial T}{\partial x} + \frac{\partial}{\partial y} k \frac{\partial T}{\partial y} + \frac{\partial}{\partial z} k \frac{\partial T}{\partial z} \tag{3.2}$$

Or in vector notation:

$$\rho c_p \frac{\partial T}{\partial t} = -\nabla \cdot (\vec{q}) = \begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \end{bmatrix} k \begin{bmatrix} \frac{\partial T}{\partial x} \\ \frac{\partial T}{\partial y} \\ \frac{\partial T}{\partial z} \end{bmatrix} \tag{3.3}$$

3.4.2 Finite Differences discretization

equation (3.2) is a partial differential equation that describes the evolution of temperature. There are two fundamentally different ways to solve it: 1) analytically or 2) numerically. Analytical solutions have the virtue that they are exact but it is often not possible to find one for complex systems. Numerical solutions are always approximations but can be found also for very complex systems. We will first use one numerical technique called finite differences. To learn how partial differential equations are solved using finite differences, we have to go back to the definition of a derivative:

$$\frac{\partial T}{\partial x} = \lim_{\Delta x \to 0} \frac{T(x + \Delta x) - T(x)}{\Delta x} \tag{3.4}$$

In our case, the above derivative describes the change in temperature with x (our space coordinate). In numerics, we always deal with discrete functions (as opposed to continuous functions), which only exist at grid points Fig. 3.2. We can therefore translate the above equation into computer readable form:

$$\frac{\partial T}{\partial x} \approx \frac{T(i+1) - T(i)}{x(i+1) - x(i)} = \frac{T(i+1) - T(i)}{\Delta x} \tag{3.5}$$

$T(i)$ is the temperature at a grid point i and Δx is the grid spacing between two grid points. Using this definition of a derivative, we can now compute the heat flow from the calculated temperature solution:

$$q_x = -k \frac{\partial T}{\partial x} \approx -k \left(\frac{T(i+1) - T(i)}{x(i+1) - x(i)} \right) \tag{3.6}$$

This form is called the *finite differences* form and is a first step towards solving partial differential equations numerically.

Note that it actually matters in which direction you count: usually it makes life much easier if indices and physical coordinates point in the same direction, e.g. x coordinate and index increase to the right.

We have learned how we can compute derivatives numerically. The next step is to solve the heat conduction equation equation (3.2) completely numerically. We are interested in the temperature evolution versus time $T(x,t)$ which satisfies equation (3.2), given an initial temperature distribution. We know already from the heat flow example how to calculate first derivatives (forward differencing):

$$\frac{\partial T}{\partial t} = \frac{T_i^{n+1} - T_i^n}{\Delta t} \tag{3.7}$$

The index n corresponds to the time step and the index i to the grid point (x-coordinate). Next, we need to know how to write second derivatives. A second derivative is just a derivative of a derivate. So we can write (central differencing):

$$\kappa \frac{\partial^2 T}{\partial x^2} \approx \kappa \frac{\frac{T_{i+1}^n - T_i^n}{\Delta x} - \frac{T_i^n - T_{i-1}^n}{\Delta x}}{\Delta x} = \kappa \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2} \tag{3.8}$$

If we combine equation equation (3.7) and equation (3.8) we get:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \kappa \left(\frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{\Delta x^2} \right) \tag{3.9}$$

Tip: Notice how we have *conveniently* used the time index n for the temperatures in the spatial derivatives. This results in the explicit form of the final discretized equation. The implicit form, which we will learn about later, would used the unknown new (time index $n+1$) temperatures for the spatial derivatives, which requires solving a system of equations.

The last step is a rearrangement of the discretized equation, so that all known quantities (i.e. temperature at time n) are on the right-hand side and the unknown quantities on the left-hand side (properties at $n+1$). This results in:

$$T_i^{n+1} = \frac{\kappa \Delta t}{\Delta x^2}\left(T_{i+1}^n - 2T_i^n + T_{i-1}^n\right) + T_i^n \tag{3.10}$$

We have now translated the heat conduction equation equation (3.2) into a computer readable finite differences form.

3.4.3 Appendix

Taylor-series expansions

Finite difference approximations can be derived through the use of Taylor-series expansions. Suppose we have a function $f(x)$, which is continuous and differentiable over the range of interest. Let's also assume we know the value $f(x_0)$ and all the derivatives at $x = x_0$. The forward Taylor-series expansion for $f(x_0 + \Delta x)$ about x_0 gives

$$f(x_0 + \Delta x) = f(x_0) + \frac{\partial f(x_0)}{\partial x}\Delta x + \frac{\partial^2 f(x_0)}{\partial x^2}\frac{(\Delta x)^2}{2!} + \frac{\partial^3 f(x_0)}{\partial x^3}\frac{(\Delta x)^3}{3!} + \frac{\partial^n f(x_0)}{\partial x^n}\frac{(\Delta x)^n}{n!} + O(\Delta x)^{n+1} \tag{3.11}$$

We can compute the first derivative by rearranging equation ref{eqs:Taylor_series_expansion}

$$\frac{\partial f(x_0)}{\partial x} = \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} - \frac{\partial^2 f(x_0)}{\partial x^2}\frac{(\Delta x)}{2!} - \frac{\partial^3 f(x_0)}{\partial x^3}\frac{(\Delta x)^2}{3!} \ldots \tag{3.12}$$

This can also be written in discretized notation as:

$$\frac{\partial f(x_i)}{\partial x} = \frac{f_{i+1} - f_i}{\Delta x} + O(\Delta x) \tag{3.13}$$

here $O(\Delta x)$ is called the *truncation error*, which means that if the distance Δx is made smaller and smaller, the (numerical approximation) error decreases with Δx. This derivative is also called first order accurate.

We can also expand the Taylor series backward

$$f(x_0 - \Delta x) = f(x_0) - \frac{\partial f(x_0)}{\partial x}\Delta x + \frac{\partial^2 f(x_0)}{\partial x^2}\frac{(\Delta x)^2}{2!} - \frac{\partial^3 f(x_0)}{\partial x^3}\frac{(\Delta x)^3}{3!} + \cdots \tag{3.14}$$

In this case, the first (backward) derivative can be written as

$$\begin{aligned} \frac{\partial f(x0)}{\partial x} = \frac{f(x_0) - f(x_0 - \Delta x)}{\Delta x} + \frac{\partial^2 f(x_0)}{\partial x^2}\frac{(\Delta x)}{2!} - \frac{\partial^3 f(x_0)}{\partial x^3}\frac{(\Delta x)^2}{3!} \cdots \\ \frac{\partial f(x_i)}{\partial x} = \frac{f_i - f_{i-1}}{\Delta x} + O(\Delta x) \end{aligned} \tag{3.15}$$

By adding equation (3.12) and equation (3.15) and dividing by two, a second order accurate first order derivative is obtained

$$\frac{\partial f(x_i)}{\partial x} = \frac{f_{i+1} - f_{i-1}}{2\Delta x} + O(\Delta x)^2 \tag{3.16}$$

By adding equations equation (3.12) and equation (3.14) an approximation of the second derivative is obtained

$$\frac{\partial f^2(x_i)}{\partial x^2} = \frac{f_{i+1} - 2f_i + f_{i-1}}{(\Delta x)^2} + O(\Delta x)^2 \tag{3.17}$$

With this approach we can basically derive all possible finite difference approximations. A different way to derive the second derivative is by computing the first derivative at $i + \frac{1}{2}$ and at $i - \frac{1}{2}$ and computing the second derivative at i by using those two first derivatives:

$$\begin{aligned} \frac{\partial f(x_{i+1/2})}{\partial x} &= \frac{f_{i+1}-f_i}{x_{i+1}-x_i} \\ \frac{\partial f(x_{i-1/2})}{\partial x} &= \frac{f_i-f_{i-1}}{x_i-x_{i-1}} \\ \frac{\partial f^2(x_i)}{\partial x^2} = \frac{\frac{\partial f(x_{i+1/2})}{\partial x} - \frac{\partial f(x_{i-1/2})}{\partial x}}{x_{i+1/2}-x_{i-1/2}} &= \frac{\frac{f_{i+1}-f_i}{x_{i+1}-x_i} - \frac{f_i-f_{i-1}}{x_i-x_{i-1}}}{0.5(x_{i+1}-x_{i-1})} \end{aligned} \tag{3.18}$$

Similarly we can derive higher order derivatives. Note that the highest order derivative that usually occurs in geodynamics is the 4^th-order derivative.

Finite difference approximations

The following equations are common finite difference approximations of derivatives. If you, in the future, need to write a finite difference approximation, come back here.

Left-sided first derivative, first order

$$\left|\frac{\partial u}{\partial x}\right|_{i-1/2} = \frac{u_i - u_{i-1}}{\Delta x} + O(\Delta x) \tag{3.18}$$

Right-sided first derivative, first order

$$\left|\frac{\partial u}{\partial x}\right|_{i+1/2} = \frac{u_{i+1} - u_i}{\Delta x} + O(\Delta x) \tag{3.19}$$

Central first derivative, second order

$$\left|\frac{\partial u}{\partial x}\right|_{i} = \frac{u_{i+1} - u_{i-1}}{2\Delta x} + O(\Delta x)^2 \tag{3.20}$$

Central first derivative, fourth order

$$\left|\frac{\partial u}{\partial x}\right|_{i} = \frac{-u_{i+2} + 8u_{i+1} - 8u_{i-1} + u_{i-2}}{12\Delta x} + O(\Delta x)^4 \tag{3.21}$$

Central second derivative, second order

$$\left|\frac{\partial^2 u}{\partial x^2}\right|_{i} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + O(\Delta x)^2 \tag{3.22}$$

Central second derivative, fourth order

$$\left|\frac{\partial^2 u}{\partial x^2}\right|_{i} = \frac{-u_{i+2} + 16u_{i+1} - 30u_i + 16u_{i-1} - u_{i-2}}{12\Delta x^2} + O(\Delta x)^4 \tag{3.23}$$

Central third derivative, second order

$$\left|\frac{\partial^3 u}{\partial x^3}\right|_{i} = \frac{u_{i+2} - 2u_{i+1} + 2u_{i-1} - u_{i-2}}{2\Delta x^3} + O(\Delta x)^2 \tag{3.24}$$

Central third derivative, fourth order

$$\left|\frac{\partial^3 u}{\partial x^3}\right|_{i} = \frac{-u_{i+3} + 8u_{i+2} - 13u_{i+1} + 13u_{i-1} - 8u_{i-2} + u_{i-3}}{8\Delta x^3} + O(\Delta x)^4 \tag{3.25}$$

Central fourth derivative

$$\left|\frac{\partial^4 u}{\partial x^4}\right|_{i} = \frac{u_{i+2} - 4u_{i+1} + 6u_i - 4u_{i-1} + u_{i-2}}{\Delta x^4} + O(\Delta x)2 \tag{3.26}$$

Note that the higher the order of the finite difference scheme, the more adjacent points are required. It is also important to note that derivatives of the following form

$$\frac{\partial}{\partial x}\left(k\frac{\partial u}{\partial x}\right) \tag{3.27}$$

should be formed as follows

$$\left|\frac{\partial}{\partial x}\left(k\frac{\partial u}{\partial x}\right)\right|_i = \frac{k_{i+1/2}\frac{u_{i+1}-u_i}{\Delta x} - k_{i-1/2}\frac{u_i-u_{i-1}}{\Delta x}}{\Delta x} + O(\Delta x)^2 \tag{3.28}$$

3.5 Example 1: Cooling dike

3.5.1 Problem description

As a first example, we will use the finite differences form of the heat diffusion equation equation (3.2) to explore the cooling of a dike. We will look at a $2m$ wide dike that intruded with a temperature of $1200°C$ into $300°C$ warm country rock. The initial conditions can then be written like this:

$$\begin{aligned} T(x < -1 \mid x > 1) &= 300 \\ T(x > -1 \& x < 1) &= 1200 \end{aligned} \tag{3.29}$$

In addition, we assume that the temperature far away from the dike center (at $|L/2|$) remains at a constant temperature. The boundary conditions are thus:

$$\begin{aligned} T(x = -\frac{L}{2}) &= 300 \\ T(x = \frac{L}{2}) &= 300 \end{aligned} \tag{3.29}$$

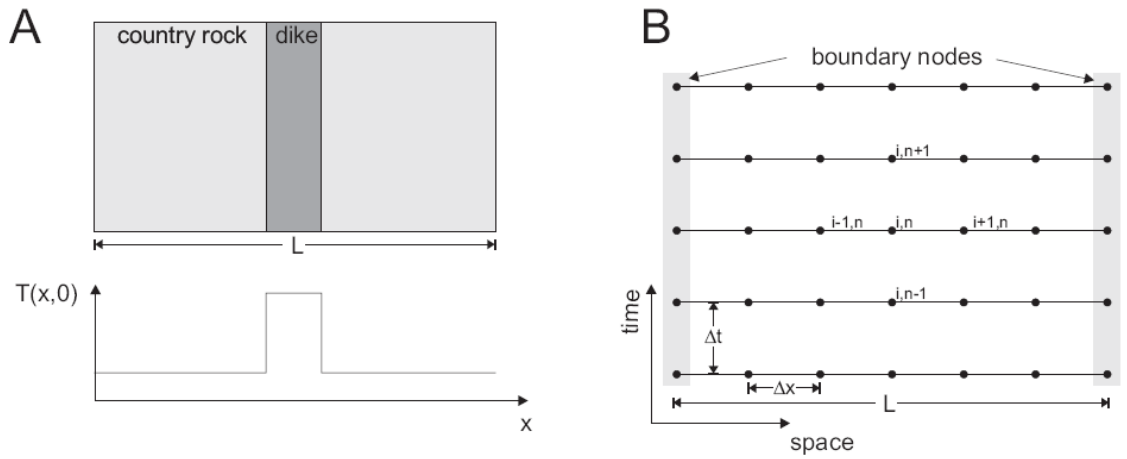


Fig. 3.2: Setup of the model considered here (A). A hot basaltic dike intrudes into colder country rock. Only variations in x-direction are considered; properties in the other directions are assumed to be constant. The initial temperature distribution $T(x, 0)$ has a step-like perturbation. B) Finite difference discretization of the 1D heat equation. The finite difference method approximates the temperature at given grid points, with spacing Δx. The time-evolution is also computed at given times with timestep Δt.

Fig. 3.2 summarizes the setup of the cooling dike problem.

3.5.2 FDM notebook

Script for solving the 1D diffusion equation

We will solve for the cooling of hot dike that was emplaced into cooler country rock. The method we are using is a simple explicit 1D finite differences scheme.

First we load some useful libraries

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
from IPython.display import HTML
```

Next we define some physics constants

[2]:
```
L       = 100           # Length of modeled domain [m]
Tmagma  = 1200          # Temperature of magma [C]
Trock   = 300           # Temperature of country rock [C]
kappa   = 1e-6          # Thermal diffusivity of rock [m2/s]
W       = 5;            # Width of dike [m]
day     = 60*60*24       # # seconds per day
dt      = 1*day         # Timestep [s]
```

and some numerical constants constants

[3]:
```
nx      = 201                                           # Number of gridpoints in x-
↪direction
nt      = 500                                           # Number of timesteps to␣
↪compute
Xvec,dx = np.linspace(-L/2, L/2, nx,  retstep=True)     # X coordinate vector,␣
↪constant spacing
beta    = kappa*dt/dx**2
```

Set the initial conditions

[4]:
```
T_init  = np.ones(nx)*Trock;              # everything is cold initially
T_init[np.nonzero(np.abs(Xvec) <= W/2)] = Tmagma   # and hot where the dike is
time    = 0
```

Make a function that computes temperature

[5]:
```
# only store the latest time step solution to Told
def fdm_solve(Told):
    Tnew=Told.copy()
    for i in range(1,len(Xvec)-1):
        Tnew[i] = beta * (Told[i+1] - 2*Told[i] + Told[i-1]) + Told[i]
    return Tnew
```

Compute and visualize everything

[6]:
```
# First set up the figure, the axis, and the plot element we want to animate
fig = plt.figure(figsize=(10,5))
ax = plt.axes(xlim=(-L/2, L/2), ylim=(0, Tmagma))
line, = ax.plot([], [], lw=1)
timeLabel=ax.text(0.02,0.98,'Time: ',transform=ax.transAxes,va='top')
ax.set_xlabel('X (m)')
ax.set_ylabel('Temperature ($^{\circ}$C)')

# Initialization function: plot the background of each frame
def init():
    line.set_data([], [])
```

(continues on next page)

(continued from previous page)

```
    return line,

# Initialize Tnew
Tnew=T_init
# Animation function which updates figure data.  This is called sequentially
def animate(i):
    timeLabel._text='Time: %.1f day'%(i*dt/day)

    # use global keyword to store the latest solution and update it using fdm_
↪solve function
    global Tnew
    Tnew=fdm_solve(Tnew)
    line.set_data(Xvec, Tnew)

    return line,

# Call the animator.  blit=True means only re-draw the parts that have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=nt, interval=30, blit=True)

plt.close(anim._fig)

# Call function to display the animation
#HTML(anim.to_html5_video())  # lower resolution
HTML(anim.to_jshtml())  # higher resolution
```

[6]:
```
<IPython.core.display.HTML object>
```

[ ]:

3.5.3 Excercises

The first step is to get the jupyter notebook to work. A good starting point is to make your own copy of the notebook. One way is to start-up your course python environment in a shell, start a jupyter lab, and create a new notebook. You can then copy the code blocks (cells) from the script into your notebook and complete the missing pieces.

```
conda activate py37_fem_lecture
cd $your_working_directory$
jupyter lab
```

Now you can explore the numerical solution.

- Complete the notebook and get it to work
- Vary the parameters (e.g. use more gridpoints, a larger timestep). Notice how the numerical solution becomes unstable when the timestep is increased beyond a certain value (what does this value depend on?). This is a major drawback of explicit finite difference codes such as the one presented here.
- Record and plot the temperature evolution versus time at a distance of 5 meter from the dike/country rock contact. What is the maximum temperature the country rock experiences at this location and when is it reached?
- Bonus question: Derive a finite-difference approximation for variable k and variable dx.

3.6 Example 2: Cooling dike - implicit version

In the previous session, we have learned how to solve for temperature using finite differences. However, our solution was only stable for certain parameter choices. In fact, we found that the finite differences formulation is only stable for $\beta < 0.5$:

$$\begin{aligned} T_i^{n+1} &= \frac{\kappa \Delta t}{\Delta x^2}\left(T_{i-1}^n - 2T_i^n + T_{i+1}^n\right) + T_i^n \\ \beta &= \frac{\kappa \Delta t}{\Delta x^2} \\ T_i^{n+1} &= \begin{bmatrix}\beta & (1-2\beta) & \beta\end{bmatrix} \begin{bmatrix} T_{i-1}^n \\ T_i^n \\ T_{i+1}^n \end{bmatrix} \end{aligned} \tag{3.29}$$

Today we will explore how to formulate a finite differences model that is always stable.

3.6.1 Stability analysis

Before progressing let us explore when and why the explicit scheme is unstable. Let us assume a temperature profile with a single peak:

$$T = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{3.29}$$

and let's further assume that $\beta = 0.2$. Then after one time step the temperature profile looks like this:

$$\begin{aligned} T_2 &= \begin{bmatrix}0.2 & 0.6 & 0.2\end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 0.2 \\ T_3 &= \begin{bmatrix}0.2 & 0.6 & 0.2\end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = 0.6 \\ T_4 &= \begin{bmatrix}0.2 & 0.6 & 0.2\end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0.2 \end{aligned}$$

A nice diffusion profile. By further try-and-error (do it for different values for beta!), we find that the scheme is only stable for $\beta < 0.5$

3.6.2 Radioactive decay analog

A good test problem to explore the stability and accuracy of different finite element schemes is radioactive decay. The change in concentration is controlled by this equation:

$$\frac{\partial c}{\partial t} = -\lambda c \tag{3.31}$$

equation (3.31) states that the rate of decay is controlled by how much of the material is still there. To solve equation (3.31), we can use our standard finite differences approach:

$$\frac{c^{n+1} - c^n}{\Delta t} = -\lambda c^n \tag{3.32}$$

This standard implementation is called the explicit form because you can write the new concentration directly as a function of the old one. The assumption here is that the concentration at the beginning of the time step controls

the decay rate. An alternative formulation, the implicit form, is to assume that the decay rate is controlled by the (unknown) concentration at the end of the time step c^{n+1}:. The implicit from is therefore:

$$\frac{c^{n+1} - c^n}{\Delta t} = -\lambda c^{n+1} \tag{3.33}$$

We will explore the differences between both formulations with a little python script.

3.6.3 FDM notebook

FD calculation of radioactive decay

We will solve for radioactive using explicit and implicit FD schemes.

Load libraries and set resolution of output figures

[18]:
```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rcParams['figure.dpi']= 300
```

Define physics constants, here we use arbitrary values without any physical meaning

[23]:
```
c0      = 1      # initial concentration
k       = 0.1    # decay constant
```

Setup numerics

[24]:
```
dt      = 2                              # time step
steps   = 41                             # number of time steps
Cana    = np.zeros(steps)                # analytical solution
Cana[0] = c0                             # initial condition

Cexp    = Cana.copy()                    # make a copy
Cimp    = Cana.copy()

Time    = np.linspace(0, (steps-1)*dt, steps)  # time stepping, and Time vector
```

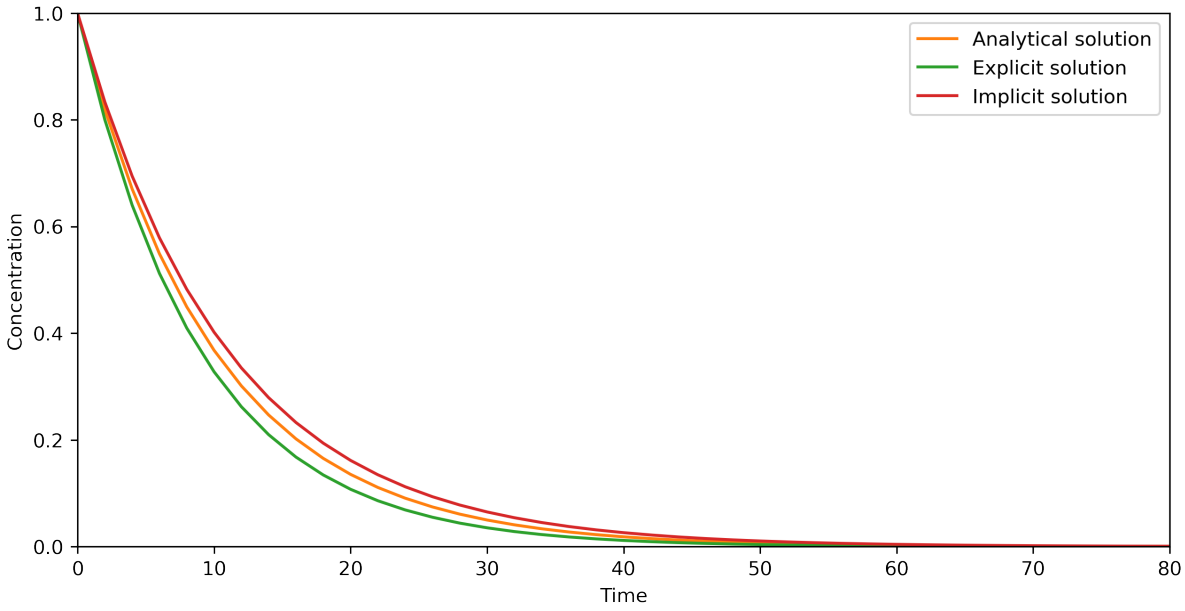
Perform calculation

[25]:
```
for n in range(0,len(Cana)-1):

    Cana[n+1] = np.exp(-k*Time[n+1])            #analytical
    Cexp[n+1] = Cexp[n]*(1-k*dt)                #explicit
    Cimp[n+1] = Cimp[n]/(1+k*dt)                #implicit
```

Plot different solutions

[26]:
```
# First set up the figure, the axis, and the plot element we want to animate
fig = plt.figure(figsize=(10,5))
fig.clf()
ax  = plt.axes(xlim=(0, dt*(steps-1)), ylim=(0, c0))
line, = ax.plot([], [], lw=1)
ax.set_xlabel('Time')
ax.set_ylabel('Concentration')
plt.plot(Time, Cana, label='Analytical solution')
plt.plot(Time, Cexp, label='Explicit solution')
plt.plot(Time, Cimp, label='Implicit solution')
plt.legend()

plt.show
```

[26]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



[ ]:

3.6.4 Excercise - radioactive decay

Try out the notebook and explore under which conditions the solutions are stable. What's the difference between the explicit and the implicit solution?

Add another method to the script that takes the concentration at the center of the time step:

$$\frac{c^{n+1} - c^n}{\Delta t} = -\lambda c^{n+\frac{1}{2}} \tag{3.34}$$

3.6.5 Implicit Heat Diffusion

The previous exercise on radioactive decay has shown that the fully implicit method is always stable. We will now rewrite our dike cooling model using the implicit formulation. Here is the implicit finite differences form:

$$\begin{aligned} \frac{T_i^{n+1} - T_i^n}{\Delta t} &= \frac{\kappa}{\Delta x^2}\left(T_{i-1}^{n+1} - 2T_i^{n+1} + T_{i+1}^{n+1}\right) \\ \begin{bmatrix} -\beta & (1+2\beta) & -\beta \end{bmatrix} &\begin{bmatrix} T_{i-1}^{n+1} \\ T_i^{n+1} \\ T_{i+1}^{n+1} \end{bmatrix} = T_i^n \end{aligned} \tag{3.35}$$

It is characteristic for the implicit form that the solution for T_i^{n+1} depends on the solution of the neighboring nodes ($i-1$ and $i+1$). As a consequence we cannot simply solve for one node after the other anymore but need to solve a system of equations. In fact, the way forward is to state the problem in matrix form $A\vec{x} = \vec{b}$ and solve all equations simultaneously:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\beta & (1+2\beta) & -\beta & 0 & 0 & 0 & 0 \\ 0 & -\beta & (1+2\beta) & -\beta & 0 & 0 & 0 \\ 0 & 0 & -\beta & (1+2\beta) & -\beta & 0 & 0 \\ 0 & 0 & 0 & -\beta & (1+2\beta) & -\beta & 0 \\ 0 & 0 & 0 & 0 & -\beta & (1+2\beta) & -\beta \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_1^{n+1} \\ T_2^{n+1} \\ T_3^{n+1} \\ T_4^{n+1} \\ T_5^{n+1} \\ T_6^{n+1} \\ T_7^{n+1} \end{bmatrix} = \begin{bmatrix} T_{top} \\ T_2^n \\ T_3^n \\ T_4^n \\ T_5^n \\ T_6^n \\ T_{bottom} \end{bmatrix} \tag{3.36}$$

All we need to do is set up the matrix A and then solve for $T^{n+1} = A \backslash T^n$!

3.6.6 FDM notebook

Implicit descretization

[2]:
```
import numpy as np
import numpy.matlib
from scipy.linalg import solve
from scipy.sparse import spdiags
import matplotlib.pyplot as plt
from matplotlib import animation
from IPython.display import HTML
import matplotlib as mpl
```

[3]:
```
L       = 100           # Length of modeled domain [m]
Tmagma  = 1200          # Temperature of magma [C]
Trock   = 300           # Temperature of country rock [C]
kappa   = 1e-6          # Thermal diffusivity of rock [m2/s]
W       = 5;            # Width of dike [m]
day     = 60*60*24       # # seconds per day
dt      = 10*day         # Timestep [s]
```

[14]:
```
nx      = 101 # Number of gridpoints in x-direction
nt      = 500                                          # Number of timesteps to␣
↪compute
Xvec,dx = np.linspace(-L/2, L/2, nx,  retstep=True)    # X coordinate vector,␣
↪constant spacing
beta    = kappa*dt/dx**2
print(beta)
```

```
0.864
```

Now we need an array to store the coefficient matrix. We use the scipy spdiags command for this, which puts the coefficients in "data" onto the diagonals defined in "diags". The result is a sparse matrix, which only stores the coefficients and not all the zeros. Boundary conditions are done as outlined in the script. Note how the setting of boundary conditions "destroys" the symmetry of the matrix. This is numerically not very smart; later we will learn how to do this better.

[5]:
```
# build the coefficient matrix
data  = (np.ones((nx,1))*np.array([-beta, (1+2*beta), -beta ])).T
diags = np.array([-1, 0, 1])
A     = spdiags(data, diags, nx, nx).toarray()

# and add boundary conditions
A[0,0]       = 1
A[0,1]       = 0
A[nx-1,nx-1] = 1
A[nx-1,nx-2] = 0
```

[6]:
```
# make initial conditions
T_init  = np.ones(nx)*Trock;              # everything is cold initially
T_init[np.nonzero(np.abs(Xvec) <= W/2)] = Tmagma    # and hot where the dike is
time    = 0                                      # track the run time
```

[7]:
```
# We only store the latest time step
def fdm_solve(Told):
    Tnew=solve(A,Told)
    return Tnew
```

[13]:
```
# First set up the figure, the axis, and the plot element we want to animate
fig = plt.figure(figsize=(10,5))
ax = plt.axes(xlim=(-L/2, L/2), ylim=(0, Tmagma))
line, = ax.plot([], [], lw=1)
timeLabel=ax.text(0.02,0.98,'Time: ',transform=ax.transAxes,va='top')
ax.set_xlabel('X (m)')
ax.set_ylabel('Temperature ($^{\circ}$C)')

# Initialization function: plot the background of each frame
def init():
    line.set_data(Xvec, T_init)
    return line,

# Initialize Tnew
Tnew=T_init
# Animation function which updates figure data.  This is called sequentially
def animate(i):
    timeLabel._text='Time: %.1f day'%(i*dt/day)

    # use global keyword to store the latest solution and update it using fdm_
↪solve function
    global Tnew
    Tnew=fdm_solve(Tnew)
    line.set_data(Xvec, Tnew)

    return line,

# Call the animator.  blit=True means only re-draw the parts that have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=nt, interval=30, blit=True)

plt.close(anim._fig)

# Call function to display the animation
#HTML(anim.to_html5_video())  # lower resolution
HTML(anim.to_jshtml())  # higher resolution
```

[13]:
```
<IPython.core.display.HTML object>
```

[ ]:
```

```

3.6.7 Excercise - implicit dike cooling

Get the notebook to work and programm the implicit solution. Explore if you find any stability limits.

Bonus: Plot the explicit and implicit solutions together. Do you find the same behavior as in the radioactive decay example?

3.7 Example 3: Periodic variations in seafloor temperature

3.7.1 Flux boundary conditions

So far we have always assumed that the boundary condition for the temperature equation was a fixed temperature. This condition is also called a *Dirichlet boundary condition*. We can however, also assume a case where the boundary has a constant gradient (e.g. heat flux). This is called a *Neumann boundary conditions* and an example would be a given flux of heat from the mantle at the base of the lithosphere or the top basement in the case of sedimentary basin

analysis. A flux boundary condition at the bottom could be written like this:

$$\frac{\partial T(y=-L,t)}{\partial y} = c_1 \tag{3.36}$$

where c_1 is the specified gradient. We could also write it as a heat flow boundary conditions

$$\frac{-q_{base}}{k} = \frac{\partial T(y=-L,t)}{\partial y} \tag{3.37}$$

We can program these conditions by using a forward or a backward finite difference expression. However, this is not so good, since these finite difference approximations are only first order accurate in space. Moreover they would yield the first derivative at the location $T_{\frac{1}{2}}$ or at $T_{ny-\frac{1}{2}}$, and not at T_1 and T_y. A better way to incorporate a flux boundary conditions is therefore to use a central finite difference approximation, which is given (at $i=1$) by:

$$\frac{T_2^{n+1} - T_0^{n+1}}{2\Delta y} = -\frac{q_{base}}{k} \tag{3.38}$$

The problem is that the expression above involves a point that is not part of the numerical grid (T_0). A way around this can be found by noting that the equation for the center nodes is given by:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \kappa \frac{T_{i+1}^{n+1} - 2T_i^{n+1} + T_{i-1}^{n+1}}{\Delta y^2} \tag{3.39}$$

Writing this expression for the first node gives

$$\frac{T_1^{n+1} - T_1^n}{\Delta t} = \kappa \frac{T_2^{n+1} - 2T_1^{n+1} + T_0^{n+1}}{\Delta y^2} \tag{3.40}$$

An explicit expression for T_0 can be obtained from equation (3.38):

$$T_0^{n+1} = T_2^{n+1} + \frac{2\Delta y q_{base}}{k} \tag{3.41}$$

Substituting equation (3.41) into equation (3.40) gives:

$$\frac{T_1^{n+1} - T_1^n}{\Delta t} = \kappa \frac{2T_2^{n+1} - 2T_1^{n+1} + 2\Delta y \frac{q_{base}}{k}}{\Delta y^2} \tag{3.42}$$

Again we can rearrange this equation to bring known terms to the right-hand side:

$$\begin{aligned} (1+2\beta)T_1^{n+1} - 2\beta T_2^{n+1} &= T_1^n + 2\beta \Delta y \frac{q_{base}}{k} \\ \beta &= \frac{\kappa \Delta t}{\Delta y^2} \end{aligned} \tag{3.43}$$

This equation only involves grid points that are part of the computational grid and equation equation (3.43) can be incorporated into the matrix A and the right-hand side b.

3.7.2 Example: seafloor temperature variations

Oceanic heat flow measurements provide important insights into cooling and alteration processes of oceanic plates. Such measurements are usually done with devices that measure heat flow within the first few meters of sediment. A natural question to ask is to which degree such measurements may be perturbed by seasonal variations in bottom water temperatures. Let's set up a simple model for this.

Assume a vertical modeling domain of \$30unt{m}\$. At the top of the domain a sinusoidal change in surface temperature ($\pm 2°C$ around $4°C$) over a one year period is applied and at the bottom a constant heat flow of $60 \frac{mW}{m^2}$ is assumed. The sediments have a constant diffusivity of $1e^{-6} m^2/s$ and a thermal conductivity of $1.5 W/m/K$.

3.7.3 FDM notebook

Periodic variations in seafloor temperature

We will explore how periodic variations in seafloor temperature affect the sub-seafloor temperature structure. For this we will learn about flux boundary conditions

[37]:
```
import numpy as np
import numpy.matlib
from scipy.linalg import solve
from scipy.sparse import spdiags
import matplotlib.pyplot as plt
from matplotlib import animation
from IPython.display import HTML
import matplotlib as mpl
```

[51]:
```
# Physical parameters
L       = 30            # Length of modeled domain [m]
Ttop    = 4             # Temperature at seafloor [C]
kappa   = 1e-6          # Thermal diffusivity of rock [m2/s]
day     = 60*60*24      # seconds per day
hf      = 0.06           # basement heat flow in W/m2
k       = 1.5            # thermal conductivity of sediment
ampl    = 2              # surface temperature variation
zeval   = -5
r_time  = 10*365*day

# Numerical parameters
ny      = 51            # Number of gridpoints in y-direction
steps   = 301;
Time,dt    = np.linspace(0, r_time, steps,  retstep=True)
Yvec,dy = np.linspace(-L, 0, ny,  retstep=True)    # Y coordinate vector, constant␣
↪spacing
```

[52]:
```
# Boundary and initial conditions
Tsurf   = Ttop + ampl*np.sin(2*np.pi/(6/12*365*day)*Time);
gradT   = -hf/k;
T_init  = Ttop + gradT*Yvec;
time    = 0;
beta    = kappa*dt/dy**2
```

[53]:
```
# setup coefficient matrix with flux boundary condition at the bottom
data  = (np.ones((ny,1))*np.array([-beta, (1+2*beta), -beta ])).T
diags = np.array([-1, 0, 1])
A     = spdiags(data, diags, ny, ny).toarray()

# Boundary conditions
# constant temperature at seafloor
A[ny-1,ny-1] = 1
A[ny-1,ny-2] = 0

# constant gradient at the bottom
A[0,(0,1)]  = [(1+2*beta), -2*beta]
```

[54]:
```
# Implicit FD solve with flux boundary conditions
def fdm_solve(Rhs):
    Tnew=solve(A,Rhs)
    return Tnew
```

[59]:
```
# First set up the figure, the axis, and the plot element we want to animate
fig = plt.figure(figsize=(10,5))
ax = plt.axes(xlim=(Ttop-ampl-1, Ttop-L*gradT+1), ylim=(-L, 0))
line, = ax.plot([], [], lw=1)
timeLabel=ax.text(0.02,0.98,'Time: ',transform=ax.transAxes,va='top')
ax.set_xlabel('Temperature ($^{\circ}$C)')
ax.set_ylabel('Depth (m)')

# Initialization function: plot the background of each frame
def init():
    line.set_data([], [])
    return line,

# Initialize Tnew
Tnew=T_init
# Animation function which updates figure data.  This is called sequentially
def animate(i):
    timeLabel._text='Time: %.1f day'%(i*dt/day)

    # use global keyword to store the latest solution and update it using fdm_
↪solve function
    global Tnew
    Rhs = Tnew.copy()
    Rhs[0]= Rhs[0] + 2*beta*dy*hf/k;
    Rhs[-1]    = Tsurf[i];
    Tnew=fdm_solve(Rhs)
    line.set_data(Tnew, Yvec)

    return line,

# Call the animator.  blit=True means only re-draw the parts that have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=steps, interval=50, blit=True)
plt.close(anim._fig)

# Call function to display the animation
#HTML(anim.to_html5_video())  # lower resolution
HTML(anim.to_jshtml())  # higher resolution
```

[59]:
```
<IPython.core.display.HTML object>
```

3.7.4 Excercise - periodic changes in seafloor temperature

tbc