AMCS 394E: FEM

Release 0.1.1

A.J.J. Lagerweij

Oct 19, 2021

HOMEWORK ASSIGNMENTS

CHAPTER

ONE

AMCS 394E: FEM

In this repository you can find my homework for Contemporary Topics in Computational Science: Computing with the Finite Element Method. The course is hosted from AMCS 394E: Computing with the Finite Element Method Git.

The folder */src/* contains the actual functions wherase the homework functions contain the homework assignments and the scripts that are used to run tha problems.

$$physics * argmin$$

1.1 Homework 1

Topic

Homework regarding the first week. The goal is to work with basic numerical approximation of PDE's' and functions.

Bram Lagerweij 18 Feb 2021

Table of Contents

1.1.1 1 Method of Lines

Consider the one-dimensional advection diffusion equation:

$$u_t + cu_x - \mu u_{xx} = 0 \qquad \forall\, x \in \Omega = [0, 1] \quad \& \quad t > 0$$

where $\mu > 0$ is the diffusion coefficient and c the wave speed. Consider periodic boundary conditions and the following initial condition:

$$u(x, 0) = \sin(2\pi x)$$

What do we expect the exact solution to do? Due to the advective part, the initial condition travels at constant speed to the right. At the same time, due to the diffusive term, the initial condition is dissipated at a rate that depends on μ.

Consider the following discretization. Use second-order central finite differences to approximate u_x and u_{xx}. Use forward and backward Euler to obtain full discretization (write down the schemes). Consider a fixed mesh with of Δx.

1.1 Advective Diffusive PDE

Consider a final time of $t = 1$, $c = 1$ and $\mu = 0.01$. For each full discretization proceed as follows:

1. Experiment using the following time step sizes: $\Delta t = 10^4$, 10^3 and 10^2.
2. How do the explicit and implicit methods behave for these time steps?

There is a so called Courant-Friedrichs-Lewy condition that formulates a condition of stability on the model:

$$C = \frac{c\Delta t}{\Delta x} \leq C_{\max}$$

Where $C_{\max}$ is a constant, which for explicit schemes, such as forward Euler, is around 1. If the condition is violated the method becomes unstable, that does not mean that the results are unstable from the first iteration.

Fig. 1.1: : The forward difference scheme is unstable for $dt = 10^{-2}$, the backward scheme behaves as expected. Click here for an animated version.

Fig. 1.2: : With a timestep of $dt = 10^{-3}$ both the forward and backward Euler scheme are stable. Click here for an animated version.

Fig. 1.3: : As expected with a timestep of $dt = 10^{-4}$ both time integrations behave stable. Click here for an animated version.

```
r"""
Solving an Advective and Diffusive PDE with finite differences.

The PDE described by

.. math::
    u_{t} + u_{x} = \mu u_{xx}  \quad \forall x \in\Omega = [0, 1]  \;\; \& \;\;  t > 0

With a periodic boundary condition. It will show a combination of diffusive
and advective behaviour. The approximation used is a second order finite
```

(continues on next page)

(continued from previous page)

```python
difference scheme in space with both a forward and backward Euler method of
lines implementation to handle the time direction.

The goal is to implement the code in python and not rely on existing solvers.

Bram Lagerweij
COHMAS Mechanical Engineering KAUST
2021
"""

# Importing External modules
import sys
import matplotlib.pyplot as plt
import numpy as np

# Importing my own scripts
sys.path.insert(1, '../src')
from finitedifference import advectivediffusive
from solvers import forwardEuler, backwardEuler


if __name__ == '__main__':
    # Define properties.
    dx = 1e-2
    dt = 1e-4
    t_end = 1
    mu = 0.01  # Diffusive term
    c = 1  # Advective term

    # Define discrete ranges.
    dof = int(1 / dx) + 1
    x, dx = np.linspace(0, 1, dof, retstep=True)
    t = np.arange(0, t_end + dt, step=dt)

    # Prepare solver.
    u0 = np.sin(2 * np.pi * x)  # Initial condition

    # Solve the problem using method of lines.
    u_forw = forwardEuler(advectivediffusive, u0, dt, t_end, args=(dof, dx, mu, c))
    u_back = backwardEuler(advectivediffusive, u0, dt, t_end, args=(dof, dx, mu, c))

    # Plotting the results.
    plt.xlim(0, 1)
    plt.xlim(0, 1)
    plt.ylim(-1, 1)
    plt.xlabel('$x$ location')
    plt.ylabel('$u(x)$')
    plt.annotate('time t={}'.format(t[-1]), xy=(0.5, 0.9), ha='center')
    plt.tight_layout()

    plt.plot(x, u_forw, label='forward')
    plt.plot(x, u_back, label='backward')
```

(continues on next page)

(continued from previous page)

```python

    plt.legend()
    plt.show()
```

1.2 Advective PDE

Consider $\mu = 0$ and $c = 2$ and solve the PDE using the explicit and the implicit methods. Use $\Delta t = 10^4$ and solve the problem for the following final times $t = 1, 5, 10, 15$ and 20. Comment on the behaviour of each full discretization as the final time increases.

Fig. 1.4: : Even with small time steps this type of hyperbolic like equation can become unstable when using a forward Euler method. Click here for an animated version.

Due to the region of convergence of the forward Euler method such a hyperbolic PDE with no dissipation will always be unstable. In the animation the instabilities become only clear after 14 seconds. Nevertheless, even at $t = 1$ the method should be considered unstable. Similarly the backward Euler is inaccurate as well, it is too dissipative, after 20 seconds around 20% of our, wave magnitude has disappeared.

```python
r"""
Solving an Advective PDE with finite differences.

The PDE described by

.. math::
    u_{t} + u_{x} = 0  \quad \forall x \in\Omega = [0, 1]  \;\; \& \;\;  t > 0

With a periodic boundary condition. The approximation used is a second order
finite difference scheme in space with both a forward and backward Euler method
of lines implementation to handle the time direction.

The goal is to implement the code in python and not rely on existing solvers.

Bram Lagerweij
COHMAS Mechanical Engineering KAUST
2021
"""

# Importing External modules
import sys
import matplotlib.pyplot as plt
import numpy as np

# Importing my own scripts
sys.path.insert(1, '../src')
from finitedifference import advective
from solvers import forwardEuler, backwardEuler


if __name__ == '__main__':
    # Define properties.
```

(continues on next page)

(continued from previous page)

```
    dx = 1e-2
    dt = 1e-4
    t_end = 20
    c = 2  # Advective term

    # Define discrete ranges.
    dof = int(1 / dx) + 1
    x, dx = np.linspace(0, 1, dof, retstep=True)
    t = np.arange(0, t_end + dt, step=dt)

    # Prepare solver.
    u0 = np.sin(2 * np.pi * x)  # Initial condition

    # Solve the problem using method of lines.
    u_forw = forwardEuler(advective, u0, dt, t_end, args=(dof, dx, c))
    # u_back = backwardEuler(advective, u0, dt, t_end, args=(dof, dx, c))

    # Plotting the results.
    plt.xlim(0, 1)
    plt.ylim(-1, 1)
    plt.annotate('time t={}'.format(t[-1]), xy=(0.5, 0.9), ha='center')
    plt.tight_layout()

    plt.plot(x, u_forw, label='forward')
    # plt.plot(x, u_back, label='backward')

    plt.legend()
    plt.show()
```

1.1.2 2 Approximation of functions

Consider the function:

$$f(x) = \sin^4(2\pi x) \qquad \forall\, x \in \Omega = [0, 1]$$

for which we have to find multiple global and local approximations. Let $f_h(x)$ be such an approximation for a given grid. We consider the following errors:

$$E_1 := \int_\Omega |f(x) - f_h(x)| dx \quad \text{and} \quad E_2 := \int_\Omega \big(f(x) - f_h(x)\big)^2 dx$$

2.1 Global Approximations

Consider the following approximations all with N terms:

1. the Taylor series around $x = 0.5$,
2. the Fourier series,
3. a global polynomial interpolation on the closed interval given by:

$$f_h(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{N-1} x^{N-1}$$

Consider different levels of refinement, $N = 4, 5, 6, \ldots, 10$ and for each approximation report both E_1 and E_2.

2.1.1 Taylor series

The Taylor series till the order N is defined through:

$$f_h(x) = \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$$

Which immediately got me into problems, analyzing the n-th derivative of a function is a numerically a pain. Quickly the round off errors become significant, and from the 5th derivative onward the basic *scipy* Taylor series function became useless. As a result I decided to hardcode the weighting constants in our expansion, these are obtained from manual derivatives.

Fig. 1.5: : Approximating $f(x)$ with a Taylor series centered around $x_0 = 0.5$ till order 10.

From Fig. 1.5 it can be observed that the Taylor series is not a very efficient approximation. At the boundary of our domain the error is very high.

2.1.2 Fourier series

The Fourier series, which we assume to be real, approximates the equation with:

$$f_h(x) = \sum_{n=0}^{N} c_n \exp^{\frac{2\pi n x}{P} i} + \bar{c}_n \exp^{-\frac{2\pi n x}{P} i}$$

where P is the period of the function $f(x)$ and c_n are complex valued coefficients that can be found through a Fourier Transform. In our case I used a FFT algorithm to find these coefficients from our discrete dataset, essentially the real-FFH tries to solve:

$$c_n = \sum_{n=0}^{K} x_k \exp^{\frac{2\pi k n}{K-1}} \qquad n = 0, \ldots, N$$

in a highly efficient manner. Notice that for each unknown c_n consists of a real and imaginary part. This does mean that this approximation for any given N is more complex. The resulting approximation is shown in Fig. 1.6. Which show that this series is highly efficient in the approximation of our function. This is not to surprising, after all we are approximation a trigonometric functions with a series of trigonometric functions it is likely that we find the exact function somewhere in our series.

Fig. 1.6: : Approximating $f(x)$ with a Fourier series seems to be exact from the fourth order.

2.1.3 Polynomial series

The polynomial series

$$f_h(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{N-1} x^{N-1}$$

was to be found with a fitting through N evenly spaced points x_i throughout this interval. It should be noted that this type of fitting can be rewritten as an minimization:

$$_{a_0,\ldots,a_{N-1}} \sum_{i=0}^{N} \left(f(x_i) - f_h(x_i)\right)^2$$

$$thatmeans: finda_0, \ldots a_{N-1} suchthat f(x_i) - f_h(x_i) = 0 \quad \forall x_i$$

This minimization can efficiently be casted to a system of equations and subsequently be solved. This system of equations has N unknowns and N functions, and because each of these functions is linearly independent a solution exists. Simply said we construct a polynomial that goes exactly through these N points.

Fig. 1.7: : Approximating $f(x)$ with a polynomials of order $N-1$ using N sample points.

One can also choose to use more sample points to evaluate the minimization problem, lets consider that we use M sample points. It is not generally possible to find a $N-1$ order polynomial to fit exactly through more then N points. But we can find the best polynomial, to be specific one that minimizes:

$$_{a_0,\ldots,a_{N-1}} \sum_{i=0}^{M} \big(f(x_i) - f_h(x_i)\big)^2$$

Which is as if we are minimizing our error E_2 at only discrete points, instead of solving the integral itself. Anyway, Fig. 1.8 shows this fit would look like. The results seems closer, because we're not just minimizing the error at N points but at $5N$ points.

Fig. 1.8: : Approximating $f(x)$ with a polynomials of order $N-1$ using $M = 5N$ sample points.

2.1.4 Comparison

For the comparison of these different approximations I've plotted the errors on a log scale. Please do note that the Fourier series has 2 times as many unknowns for the N compared to the other methods.

Fig. 1.9: : The error E_1 for our different approximations where the approximation order ranges from 1 to 20.

I assume that the error of the Taylor series is increasing because the higher order terms will cause higher errors at the boundaries of our domain. But all in all it is my opinion that the Taylor series is a bad approximation for this purpose, it is difficult to calculate due to the derivatives and the result is inaccurate. This is not so surprising however, Taylor series are meant to approximate the behaviour of a function around a given point x_0 to characterize the local behaviour. We are here using it on a relatively large domain.

The script used for these computations can be found at 3 GlobalApproximation.py.

2.2 Local Approximations

Split the domain Ω into N cells. For each cell K, compute linear and quadratic approximations $f_K(x)$ where $f_K(x_i) = f(x_i)$ where x_i are evenly spaced gridpoints, including the boundaries of the cell. Compute and report both E_1 and E_2 for a different numbers of cells $N = 4, 5, 6, \ldots, 10$.

The approximation by linear elements is created by scaling hat (shape) functions appropriately. These functions are chosen in such a way that:

1. The sum of all the shape functions together equals one, $\sum_{n=1}^{N} \varphi_i(x) = 1$ This is called the Partition of Unity Method.
2. There where a single function reaches its maximum all the other functions equal zero.

Fig. 1.10: : The error E_2 for our different approximations where the approximation order ranges from 1 to 20.

Then our approximation is defined by:

$$f_h(x) = \sum_{n=1}^{N} w_n \varphi_n(x)$$

where the weights w_n are unknown. But because the shape function where chosen smartly these weights are independent. After all at the point where a single shape function reaches its maximum (1) the other functions are zero. As a result the weight of this shape function equals the value of the function we are trying to approximate at the center point of the shape:

$$w_n = f(X_n)$$

where X_n denotes the point where shape function $\varphi_n(x)$ reaches its maximum.

2.2.1 Linear Elements

In the case of linear elements these shape functions are defined as:

$$\varphi_n(x) = \begin{cases} 0 & \forall \quad 0 \\ \leq x \leq & X_{n-1} \\ \frac{x-X_{n-1}}{X_n-X_{n-1}} & \forall \quad X_{n-1} \\ \leq x \leq & X_n \\ 1-\frac{x-X_n}{X_{n+1}-X_n} & \forall \quad X_n \\ \leq x \leq & X_{n+1} \\ 0 & \forall \quad X_{n+1} \\ \leq x \leq & L \end{cases}$$

where X_n is the node of this shape function, X_{n-1} and X_{n+1} the nodes surrounding ours.

A more efficient formulation includes the creation of a unit function that is rescaled depending on the locations of the nodes. But I haven't yet implemented such an function yet.

Fig. 1.11: : The function $4\sin(\pi x) + 1$ approximated with four elements. The first element contain the orange and half of the green shape function.

Fig. 1.12: : The function $4\sin(\pi x) + 1$ approximated more and more linear elements.

2.2.2 Quadratic Elements

In the case of quadratic elements there are two different types of shape function. One of these function extents into two elements, similar to what the linear element does. The second shape function is only inside a single element, and on an interior node. This node is placed exactly in the middle between the start and end of the element. I'll give these nodes

Fig. 1.13: : The approximation of $f(x)$ with linear elements.

the subscripts $n-\frac{1}{2}$ and $n+\frac{1}{2}$. Now the shape functions are defined by:

$$\varphi_n(x) = \begin{cases} 0 & \forall \quad 0 \\ \leq x \leq & X_{n-1} \\ \frac{2}{(X_n-X_{n-1})^2}(x-X_{n-1})(x-X_{n-\frac{1}{2}}) & \forall \quad X_{n-1} \\ \leq x \leq & X_n \\ \frac{2}{(X_{n+1}-X_n)^2}(x-X_{n+1})(x-X_{n+\frac{1}{2}}) & \forall \quad X_n \\ \leq x \leq & X_{n+1} \\ 0 & \forall \quad X_{n+1} \\ \leq x \leq & L \end{cases}$$

$$\varphi_{n-\frac{1}{2}}(x) = \begin{cases} 0 & \forall \quad 0 \\ \leq x \leq & X_{n-1} \\ -\frac{4}{(X_n-X_{n-1})^2}(x-X_{n-1})(x-X_n) & \forall \quad X_{n-1} \\ \leq x \leq & X_n \\ 0 & \forall \quad X_{n+1} \\ \leq x \leq & L \end{cases}$$

Again a more efficient formulation includes the creation of a unit function that is rescaled depending on the locations of the nodes. But I haven't yet implemented such an function yet.

Fig. 1.14: : The function $4\sin(\pi x)+1$ approximated with four elements. The first element contain the orange and half of the green shape function.

Fig. 1.15: : The function $4\sin(\pi x)+1$ approximated more and more quadratic elements.

It is important to notice from Fig. 1.16 that the resulting curve is not smooth. for example at $x = 0.5$ one can see that the red approximation (6 elements) is non-smooth.

2.2.3 Comparison

For the comparison of these different approximations I've plotted the errors on a log scale. Please do note that the quadratic elements have $(N+1)N$ unknowns where the linear elements have $N+1$ weights to be determined. Nevertheless there is no interdependency between these weights, which as mentioned before means that these can be determined independently.

The script used for these computations can be found at 4 LocalApproximation.py.

$$physics * argmin$$

Fig. 1.16: : The approximation of $f(x)$ with quadratic elements.

Fig. 1.17: : The error E_1 for our element based approximations with 1 to 20 elements.

1.2 Homework 2

Topic

Homework regarding the third week. The goal is to work with simple 1D FEM methods. We'll be solving several PDEs and project function on a FEM space.

Bram Lagerweij 2 Mar 2021

Table of Contents

1.2.1 1 Project the Navier-Stokes equations

Consider the incompressible Navier-Stokes equations in non-conservative form:

$$\begin{aligned} \partial_t \vec{u} + \vec{u}\,\vec{u} + \frac{1}{\rho} p - \mu^2 \vec{u} &= \vec{f} \qquad \forall \quad \vec{x} \in \Omega \\ \vec{u} &= 0 \qquad \forall \quad \vec{x} \in \Omega \\ \vec{u}\vec{n} &= 0 \qquad \forall \quad \vec{x} \in \Omega \end{aligned}$$

where $\vec{u}, \vec{x}, \vec{f}, \vec{n} \in \mathbb{R}^d$ are the speed, location, external forces and surface normal, ρ the density, μ the viscosity and p the pressure. The original Chorin's projection method considers the following discretziation in time:

$$\frac{\vec{u}^* - \vec{u}^n}{\Delta t} + \vec{u}^n\,\vec{u}^n - \mu^2 \vec{u}^* = \vec{f}$$

Fig. 1.18: : The error E_2 for our element based approximations with 1 to 20 elements.

where we ingore the pressure as a kind of operation splitting. The non-linear term is treated explicitely to avoid the non-linearity and we treat the viscouse term implicitely to avoid extreme small time step restrictions. However this does not ensure that $\vec{u}^* = 0$. To fix this, the projection method considers:

$$\frac{\vec{u}^{n+1} - \vec{u}^*}{\Delta t} = -\frac{1}{\rho} p^{n+1}$$

When we take the divergence we impose $\vec{u}^{n+1} = 0$ to get:

$$\Delta p^{n+1} = \frac{\rho}{\Delta t} \vec{u}^*$$

Finaly, the updated divergence-free velocity is given by:

$$\vec{u}^{n+1} = \vec{u}^* - \frac{\Delta t}{\rho} p^{n+1}$$

1.1 The shape functions

Consider two discrete spaces. For the velocity and pressure use continuous piecewise bi-quadratic and bilinear polynomials (in 2D)

$$\begin{aligned} p_1(x,y) &= c_0 x + c_1 y + c_2 xy + c_3 \\ p_2(x,y) &= c_0 x^2 + c_1 x^2 y + c_2 x^2 y^2 + c_3 y^2 + c_4 xy^2 + c_5 x + c_6 y + c_7 xy + c_8 \end{aligned}$$

respectively. How many shape function do we have for each space in the reference element? Derive the shape functions for the reference element (hint: use tensor products). The code used to plot these two figures is available in 1 shape2D.py.

Fig. 1.19: : Quadrilateral elements with linear shape functions.

Fig. 1.20: : Quadrilateral elements with quadratic shape functions.

1.2 Weak form of Chorin's projection

Consider the previously described NS-equations and the Chorin'n projection method and obtain:

1. **Weak formulation,** From what I understand the this approach goes in three steps:

    1. **Solve PDE 1,**

    $$\frac{\vec{u}^* - \vec{u}^n}{\Delta t} + \vec{u}^n \ \vec{u}^n - \mu^2 \vec{u}^* = \vec{f}$$

    where $\vec{u}^*$ is the unknown and all other variables are known. This is PDE can be written into the following format:

    $$\alpha \vec{u}^* - \beta^2 \vec{u}^* = \vec{b}_1$$

    which is a non-homogeneous diffusion equation, but vector valued, as $\vec{u}$ is a vector.

    2. **Solve PDE 2:**

    $$^2 p^{n+1} = \frac{\rho}{\Delta t} \vec{u}^*$$

    where p is the variable to be determined, through the a Poisson equation.

    $$^2 p^{n+1} = \vec{b}_2$$

3. **Obtain new primal $\vec{u}$ by updating it through:**

$$\vec{u}^{n+1} = \vec{u}^{*} - \frac{\Delta t}{\rho} p^{n+1}$$

This is simply an update, there is no PDE to be solved and everything on the right hand side is known.

2. **Discrete weak form and,** Incomplete.
3. **The linear algebra representation of the problem.** I'll first need to find the answer to the previous question, nevertheless it is clear that we need at least the mass and the stiffness matrix for the first PDE. The transport matrix is also required to compute the right hand side in the first PDE.

1.2.2 2 Project a smooth function to FE space

From HW1 we consider the following function again:

$$f(x) = \sin^4(2\pi x) \quad \forall \quad 0 \leq x \leq 1$$

and project it on the finite element space.

2.1 Projection

Perform the projection through the following steps.

1. **Consider piecewise linear and quadratic continuous polynomials.** Done, see `element.shape1d()`.
2. **Consider the reference element $[0, 1]$ and interpolatory basis functions to derive the shape functions for each space.** Done, see `element.shape1d()`.
3. **What is the weak formulation and the linear algebra problem associated with the projection?** The derivation of the weak form is described at `pde.projection()`.
4. **Compute the entries of the mass matrix for each space.** Done, see `fem.element_mass()`.
5. **Solve the system to obtain the DoF associated with the projection.** Done, see `solvers.solve()`.
6. **Plot the projected functions considering $N = 25, 50, 100$ and 200 cells.** Done, the main code, in 2 ProjectionFE.py, was used to create Fig. 1.21 and Fig. 1.22.

Fig. 1.21: : Approximating $f(x)$ with a finite element projections with N linear elements.

Fig. 1.22: : Approximating $f(x)$ with a finite element projections with N quadratic elements.

2.2 Evaluate Projection

For both projections compute the following two errors

$$E_1 = \int_0^1 \|f(x) - f_h(x)\| x E_2 = \sqrt{\int_0^1 (f(x) - f_h(x))^2 x}$$

where $f_h(x)$ is the projection of $f(x)$ on our FE space.

Fig. 1.23: : Comparing error 1 to the number of elements shows faster convergence of the quadratic elements. The order seems to be 2 and 3 respectively.

Fig. 1.24: : Comparing error 2 to the number of elements shows faster convergence of the quadratic elements. The order seems to be 2 and 3 respectively.

Estimate the order of convergence for each space. That is assume that the error behaves as:

$$E = ch^p$$

where c is a constant and $h = 1/N$ is the mesh size. When is the value of p? Does this error behave different for the different spaces and norms?

Table 1.1: : The convergence power of different approximations to the smooth projection.

	Linear				Quadratic			
N	E1 p(1/N)	E2 p(1/N)	E1 p(1/DoFs)	E2 p(1/DoFs)	E1 p(1/N)	E2 p(1/N)	E1 p(1/DoFs)	E2 p(1/DoFs)
4	1.93	1.90	2.61	2.57	1.98	1.93	2.33	2.28
8	1.17	1.16	1.38	1.37	2.09	2.02	2.28	2.20
16	1.83	1.67	1.99	1.82	2.80	2.82	2.92	2.95
32	2.27	2.20	2.37	2.30	2.75	2.75	2.82	2.81
64	2.09	2.05	2.13	2.10	2.95	2.91	2.98	2.95
128	2.02	2.01	2.04	2.04	2.99	2.98	3.01	2.99
256	2.01	2.00	2.02	2.01	3.01	2.99	3.01	3.00
512	2.00	2.00	2.01	2.01	3.00	3.00	3.00	3.00
1024	2.00	1.99	2.00	2.00	3.00	3.00	3.00	3.00
2048	2.00	2.00	2.00	2.00	3.00	3.00	3.00	3.00
4096	2.00	2.00	2.00	2.00	3.00	3.00	3.00	3.00
8192	2.00	2.00	2.00	2.00	3.01	3.00	3.01	3.00
16384	2.00	1.99	2.00	1.99	3.00	3.00	3.00	3.00
32768	2.00	2.01	2.00	2.01	3.00	3.00	3.00	3.00
65536	2.00	2.00	2.00	2.00	3.00	3.00	3.00	3.00
131072	2.00	2.00	2.00	2.00	3.00	3.00	3.00	3.00

Fig. 1.25: : Comparing error 1 to the amount of degrees of freedom still shows faster convergence of the quadratic elements. Clearly the difference is less pronounced, because the quadratic elements have more unknowns per element. The order seems to be 2 and 3 respectively.

Fig. 1.26: : The result for error 2 is again similar to that for error 1. The order seems to be 2 and 3 respectively.

1.2.3 3 Project a non-smooth function to FE space

Preform the same projection for the following non-smooth function:

$$f(x) = \begin{cases} 1 & 0.35 \leq x \leq 0.65 \\ 0 & \text{otherwise} \end{cases}$$

For which the the main code can be found in 3 ProjectionFE.py

Fig. 1.27: : Approximating the discrete function with a finite element projections with N linear elements is not improving with a refined mesh. The spikes around the step change keep the same height, although the width is reducing.

Fig. 1.28: : Moving to quadratic elements make it even worse, the spikes at the step change get higher.

Table 1.2: : The convergence power of different approximations of the non smooth projection.

	Linear				Quadratic			
N	E1 p(1/N)	E2 p(1/N)	E1 p(1/DoFs)	E2 p(1/DoFs)	E1 p(1/N)	E2 p(1/N)	E1 p(1/DoFs)	E2 p(1/DoFs)
4	0.95	0.53	1.29	0.72	0.72	0.37	0.85	0.44
8	0.27	1.39	0.31	1.64	0.21	1.02	0.22	1.11
16	1.41	0.34	1.53	0.37	0.70	-0.01	0.73	-0.01
32	0.47	1.64	0.49	1.71	0.30	1.01	0.30	1.04
64	1.51	0.35	1.55	0.36	0.70	-0.01	0.71	-0.01
128	0.49	1.65	0.49	1.66	0.30	1.01	0.30	1.02
256	1.52	0.35	1.53	0.35	0.70	-0.01	0.71	-0.01
512	0.48	1.65	0.48	1.65	0.29	1.01	0.29	1.01
1024	1.52	0.35	1.52	0.35	0.71	-0.01	0.71	-0.01
2048	0.48	1.64	0.48	1.65	0.29	1.02	0.29	1.02
4096	1.52	0.36	1.52	0.36	0.71	-0.01	0.71	-0.01
8192	0.48	1.64	0.48	1.64	0.29	1.01	0.29	1.01
16384	1.51	0.35	1.51	0.35	0.70	-0.02	0.70	-0.02
32768	0.49	1.64	0.49	1.64	0.31	1.01	0.31	1.01
65536	1.53	0.38	1.53	0.38	0.72	0.02	0.72	0.02
131072	0.46	1.64	0.46	1.64	0.25	1.06	0.25	1.06

1.2.4 4 Solve Advection-Diffusion PDE with FE

Consider the one-dimensional advection diffusion equation:

$$u_t + u_x - \mu u_{xx} = 0 \qquad \forall \qquad x \in \Omega = [0, 2\pi]$$

where $\mu > 0$ is a coefficient. Consider periodic boundary conditions and the following initial conditions:

$$u(x, 0) = \sin^4(x)$$

Fig. 1.29: : Comparing error 1 to the number of elements shows faster convergence for the linear elements. The order seems to be 1 and less then 1 respectively.

Fig. 1.30: : Both approximations seem to be equally bad. The order seems to be less then 1.

The exact solution to this equation is given by:

$$u(x,t) = \frac{3}{8} - \frac{1}{2}\exp^{-4\mu t}\cos(2(x-t)) + \frac{1}{8}\exp^{-16\mu t}\cos(4(x-t))$$

4.1 Solve through FEM

Solve this problem using a FEM implementation with the following steps:

1. **Consider continuous piecewise linear polynomials and interpolatory basis functions.** Done, see `element.shape1d()`.
2. **Obtain the discrete weak formulation.** We need two steps here, firstly we need to project the initial condition, for which the weak form is derived in *`pde.projection()`*. Secondly the PDE will be solved using the method of lines, see *`solvers.forwardEuler()`* and *`solvers.backwardEuler()`*, which needs to be fed with the weak form of the PDE, avalible at *`pde.advectivediffusive()`*.
3. **Identify the different matrices associated with the finite element discretization.** For these functions the Mass (fem.element_mass), Transport (fem.element_transport) and Stiffness (fem.element_stiffness) matrices need to be obtained.
4. **Implement and solve the equation via finite elements up to $t = 2\pi$.** Done, the code in Done, the main code, in 4_AdvectionDiffusion.py, produces Fig. 1.33, Fig. 1.34, Fig. 1.35 and Fig. 1.36.

4.2 Compute the error

Compute the errors E_1 and E_2 and compare the results to those of previous weeks homework, in which the same PDE was solved using a Finite Difference approach. Preform a convergence test as described in *2.2 Evaluate Projection*.

Table 1.3: : The convergence power of the different approximation schemes.

		FD forward		FD backward		FE forward		FE backward	
N	dt	p E1	p E2	p E1	p E2	p E1	p E2	p E1	p E2
4	6.25E-04	-1.65	-1.87	-1.65	-1.87	2.53	2.52	2.53	2.52
8	1.56E-04	-2.06	-1.86	-2.06	-1.85	-1.55	-1.66	-1.55	-1.66
16	3.91E-05	0.56	0.35	0.56	0.35	2.08	2.19	2.08	2.19
32	9.77E-06	1.66	1.61	1.66	1.61	3.52	3.40	3.50	3.39
64	2.44E-06	1.90	1.84	1.90	1.84	2.33	2.29	2.33	2.28
128	6.10E-07	1.99	1.99	1.99	1.99	2.02	2.02	2.02	2.02

Fig. 1.31: : The linear approximation is better then the quadratic one. The order seems to be 1 and less then 1 respectively.

Fig. 1.32: : Both approximations seem to be equally bad. The order seems to be less then 1.

Fig. 1.33: : In this course grid large differences between the FD and FE methods can be observed. The forward and backward scheme preform nearly the same.

$$physics$$

1.3 About

Toppic

The reason for me is to solve classical problems in Solid Mechanics. This section, and those below will introduce the typical equations that are encountered in solid mechanics. This section is not exhaustive and it might be extended in the future to discuss more details.

Bram Lagerweij 11 Feb 2020

The examples will become gradually more complex. It starts with the simplest problem, the Laplace equation:

$$^2u(\vec{m}) = 0 \qquad \forall \vec{m} \in \Omega$$

In here one can imagine various levels of complication:

1. With a simple geometry, no sharp corners, and a combination of Neuman and Diriclet boundary conditions.
2. With a more complex geometry, sharp corners, cracks and inclusions.
3. With a 'non-linear' stiffness, $(\vec{C}\,u(\vec{m}))$ adding a non-constant variable $\vec{C}$ which is a function depending somehowe on u.
4. Where $\vec{C}$ is non-linear and history dependent, aka $\vec{C}^{(n+1)}$ is a function af all previous timesteps.
5. With softening in the non-linear stiffness C, that is the tangent of $\vec{C}\,u$ will become negative at some point.
6. Versions in 3D

Moving on to solids where we solve elasticity and plasticity equations:

$$\begin{aligned} \sigma + \vec{b} &= 0 \qquad \forall \vec{m} \in \Omega \\ where \sigma &= C : \varepsilon \\ \varepsilon &= \frac{1}{2}\left(\vec{u} + (\vec{u})^T\right) \end{aligned}$$

The simplest problem would be linear elasticity, but more complicated versions can be build as well.

1. With a simple geometry, no sharp corners, and a combination of Neuman and Diriclet boundary conditions.
2. With a more complex geometry, sharp corners, cracks and inclusions.

Fig. 1.34: : With a finer grid and time step the differences become smaller.

Fig. 1.35: : And smaller.

Fig. 1.36: : At the finished mesh and time step the results become quite close to the exact solution.

3. Large displacements (geometrically non-linear) and deformations (this might require a different strain measure).
4. Softening and possbily fracture.
5. Self Contact.

$physics$

1.4 Poisson Equation

Toppic

The Poisson equation is the simplest example of the PDE's considerd in Solid Mechanics. It is an eliptical PDE, and is simplified compared to linear elasticity in the sense that its solution is a scalar field, instead fo the vector field found in elasticity problems. This makes Poisson's equation a good start to explore numerical solving strategies for Solid Mechanics problems.

Bram Lagerweij 11 Feb 2020

Table of Contents

Fig. 1.37: : The finite element method seems to converge faster with respect to error 1. This must come from the change in mass matrix, as the stiffness and transport matrix don't differ from the FD method. There does not seem to be a difference between the forward and backwards methods, because the time steps are small enough for the forward method to be stable.

Fig. 1.38: : The behaviour of E_2 is similar to that of $E1$.

1.4.1 1 Laplace Equation

The most basic description of the Laplace equation is given by:

$$
\begin{aligned}
{}^2u(\vec{m}) &= [2]ux + [2]uy = 0 \\
\forall \vec{m} \in \Omega \\
\text{s.t.:} \quad & u(\vec{m}) = \tilde{\vec{u}}(\vec{m}) \\
\forall \vec{m} \in \mathcal{S}_u \\
& u(\vec{m}) = \tilde{\vec{t}}(\vec{m}) \\
\forall \vec{m} \in \mathcal{S}_t
\end{aligned}
$$

Where the entirety of the boundary $\partial\Omega$ is the union of these to boundary conditions that do not intersect.

$$
\begin{aligned}
\partial\Omega &= \mathcal{S}_u \cup \mathcal{S}_t \\
0 &= \mathcal{S}_u \cap \mathcal{S}_t
\end{aligned}
$$

The following images summarizes this.

Fig. 1.39: A domain Ω subjected to the Laplace equation with combined boundary conditions.

1.4.2 2 Poisson equation

In case of non-homogeneous formulations the Laplace equations is called the Poisson equation.

$$
\begin{aligned}
{}^2u(\vec{m}) &= [2]ux + [2]uy = \vec{b}(\vec{m}) \\
\forall \vec{m} \in \Omega \\
\text{s.t.:} \quad & u(\vec{m}) = \tilde{\vec{u}}(\vec{m}) \\
\forall \vec{m} \in \mathcal{S}_u \\
& u(\vec{m}) = \tilde{\vec{t}}(\vec{m}) \\
\forall \vec{m} \in \mathcal{S}_t
\end{aligned}
$$

The boundary condition can still be defined in the same way as in the Laplace equation. An example of such a Poisson problem in 1D is a statically determinate Euler-Bernoulli beam problem. Solving a these linear beam problem can be done with finite differences.

The PDE described by

$$
EIu''(x) = M(x) \qquad \forall x \in \Omega = [0, L]
$$

Where M is the internal bending moment of the beam. This beam has a length L and a stiffness EI. In general these kinds of problems can not be solved directly in this way, as it is not always possible to describe the moment explicitly, but because our cantilever beam is statically determinate it can be done. Now we'll be exploring two examples to introduce the different types of boundary conditions.

Example 1: Dirichlet

Fig. 1.40: A beam that is simply supported at $x = 0$ and 250 mm and subjected to a point load.

In this example we consider a beam with a length of 1000mm which is simply supported at $x = 0$ and $x = 250$. Simply supported means that the displacement u at those points is fixed and equals 0. That is our ODE becomes:

$$\begin{aligned} EI\,u''(x) =& M(x) \quad \forall \quad 0 \leq x \leq 1000 \\ \text{where:} \quad & M(x) = \begin{cases} -3Px & 0 \leq x \leq L/4 \\ P(x-L) & L/4 \leq x \leq L \end{cases} \\ \text{s.t.:} \quad & u(0) = 0 \\ & u(L/4) = 0 \end{aligned}$$

where I did compute the moment equation explicitly already. To derive u'' a central difference scheme is used,

$$u''(x) = \frac{u(x-dx) - 2u(x) + u(x+dx)}{dx^2}$$

We'll be evaluating this derivative an N regularly distributed points in our domain. And if we note x_n as the location of one of these points than we can note the derivative as:

$$u''(x_n) = \frac{u(x_{n-1}) - 2u(x_n) + u(x_{n+1})}{dx^2}$$

This is implemented into a matrix format by `finitedifference.Dxx()`, such that:

$$u'' = D_{xx}u$$

where u is a vector with the field at all the discrete points and u'' the derivative that was calculated. This does however not yet specify the way to analyze the derivative at the first and last points. After all that would require the calculation of u outside the domain. As a result the matrix will have an empty first and last row.

This and the right hand side (f) of the Poisson equation are available through `finitedifference.poisson()`. You would expect that we can solve the system of equations:

$$EI\,D_{xx}\,u = f$$

but that is not true, as we'll have to deal with the boundary conditions as well, without those the problem is singular. To be specific we know that $u(0) = 0$ and $u(L/4) = 0$, this can be used to make the problem determinate. Lets say that $x_0 = 0$ and $x_n = L/4$ then we can add the following to equations to our system of equations:

$$u_0 = 0 and u_n = 0$$

these two equations can be placed in the still empty first and last row of our stiffness matrix and right hand side. That is in the first row we make the first element equal to 1 and the rest all equal to 0. Similarly the right hand side of the first degree of freedom is set to 0. In the last row we set the degree of freedom that corresponds to x_n to 1 and the rest to 0, here we do also set the right hand side of the last row equal to zero (see lines 53 to 61 in the code below).

Fig. 1.41: The finite difference solution of the beam problem seems to be in good agreement with the exact result. This simulation was run with 101 degrees of freedom.

```python
# Importing External modules
import sys
import numpy as np
from scipy.sparse.linalg import spsolve

# Importing my own scripts
sys.path.insert(1, '../src')
from finitedifference import poisson


def moment(P, L):
    """
    Moment as a function of :math:`x` of the double simply supported beam.

    Parameters
    ----------
    P : float
        Applied load.
    L : float
        Length of the beam

    Returns
    -------
    callable
        The moment :math:`M(x)` of the beam.
    """

    def fun(x):
        shape = np.shape(x)

        if len(shape) == 0:
            if x < L / 4:
                m = -3 * P * x
            else:
                m = P * (x - L)
        else:
            m = np.zeros_like(x)
            ind = np.where(x < L / 4)  # where x < L/4
            m[ind] = -3 * P * x[ind]

            ind = np.where(L / 4 <= x)  # where L/4 < x
            m[ind] = P * (x[ind] - L)
        return m

    return fun


if __name__ == '__main__':
    # Define properties of the problem.
    L = 1000  # 1000 mm length
    P = 1  # 1 N load
    EI = 187500000  # Beam bending stiffness Nmm^4

```

(continues on next page)

(continued from previous page)

```python
    # Discretion of the space.
    dof = 101  # Number of nodes
    x, dx = np.linspace(0, L, dof, retstep=True)

    # Calculate the internal Moment.
    M = moment(P, L)  # Create a callable for the moment in Nmm

    # Create linear problem.
    K, f = poisson(dof, dx, M, c=EI)

    # Boundary condition u(0) = 0
    K[0, 0] = 1
    f[0] = 0

    # Boundary condition u(L/4) = 0  For this purpose we use
    # the last row of the matrix, this row is not yet used.
    index = int(dof / 4)
    K[-1, index] = 1
    f[-1] = 0

    # Solve the problem.
    u = spsolve(K, f)
```

Example 2: Dirichlet and Neumann

Fig. 1.42: A cantilever beam is fixed in the wall of the left and subjected to a point load at the right. This type of constraint, called an endcast, limits both the displacement and rotation, that is $u(0) = 0$ and $u'(0) = 0$.

The approach follows exactly what was described in example 1, except of course the constraints. Our problem is formulate following:

$$\begin{aligned} EI\, u''(x) &= M(x) \quad \forall \quad 0 \leq x \leq 1000 \\ \text{where:} \quad & M(x) = P(x - L) \\ \text{s.t.:} \quad & u(0) = 0 \\ & u'(0) = 0 \end{aligned}$$

where the moment did change as well because the loading conditions changed. That is after discritization our system of equations is represented by:

$$EI\, D_{xx}\, u = f$$

Now as for the boundary conditions, for the first row we again fill the first element with a 1 and leave the rest 0. In the right hand side we set the value of the forcing term equal to zero. As a result the first row reads:

$$u(x_0) = 0$$

Now for the Neumann boundary it is a bit more tricky. The derivative $u'(x_0)$ can be approximated with a backwards finite difference:

$$u'(x_0) = \frac{-u(x_0) + u(x_1}{dx} = 0$$

I'll put this in the last row as that one is not yet populated. That means that we have to populate the first element of the last row with a -1, the second element of that row with a 1 and set the last element of the right hand side to zero as well. (see lines 64 to 72 below)

Fig. 1.43: The finite difference solution of the beam problem seems to be in good agreement with the exact result. This simulation was run with 101 degrees of freedom.

```
 1  # Importing External modules
 2  import sys
 3  import numpy as np
 4  from scipy.sparse.linalg import spsolve
 5
 6  # Importing my own scripts
 7  sys.path.insert(1, '../src')
 8  from finitedifference import poisson
 9
10
11  def moment(P, L):
12      """
13      Moment as a function of :math:`x` of the cantilever beam.
14
15      Parameters
16      ----------
17      P : float
18          Applied load.
19      L : float
20          Length of the beam
21
22      Returns
23      -------
24      callable
25          The moment :math:`M(x)` of the beam.
26      """
27      def fun(x):
28          return P*(x-L)
29      return fun
30
31
32  if __name__ == '__main__':
33      # Define properties of the problem.
34      L = 1000  # 1000 mm length
35      P = 1  # 1 N load
36      EI = 187500000  # Beam bending stiffness Nmm^4
37
38      # Discretion of the space.
39      dof = 101  # Number of nodes
40      x, dx = np.linspace(0, L, dof, retstep=True)
41
42      # Calculate the internal Moment.
43      M = moment(P, L)  # Create a callable for the moment in Nmm
44
45      # Create linear problem.
```

(continues on next page)

(continued from previous page)

```
    K, f = poisson(dof, dx, M, c=EI)

    # Boundary condition u(0) = 0
    K[0, 0] = 1
    f[0] = 0

    # Boundary condition u'(0) = 0 with a finite difference.
    # For this purpose we use the last row of the matrix
    # This row is not yet used
    K[-1, 0] = -1 / dx
    K[-1, 1] = 1 / dx
    f[-1] = 0

    # Solve the problem.
    u = spsolve(K, f)
```

1.5 Partial Differential Equations

Storing various PDEs that can be will be solved in this course. This includes:

- Diffusive 1D

$$u_t - \mu u_{xx} = 0 \qquad \forall x \in \Omega = [0, L] \quad \& \quad t > 0$$

- Advective 1D

$$u_t + c u_x = 0 \qquad \forall x \in \Omega = [0, L] \quad \& \quad t > 0$$

- Diffusive-Advective 1D

$$u_t + c u_x - \mu u_{xx} = 0 \qquad \forall x \in \Omega = [0, L] \quad \& \quad t > 0$$

- Poisson in 1D

$$u_{xx} = f(x) \qquad \forall x \in \Omega = [0, L]$$

The goal is to implement the code in python and not rely on existing methods.

Bram Lagerweij COHMAS Mechanical Engineering KAUST 2021

`pde.`**`advective`**(*x*, *connect*, *c*, *num_q*, *order*)

Time derivative of the PDE for advective diffusive problems.

$$u_t + c u_x = 0 \qquad \forall x \in \Omega = [0, L] \quad \& \quad t > 0$$

Which, is converted into a weak form through:

$$\int_\Omega (\tilde{u}_t + c\tilde{u}_x)\phi_i(x)dV = 0 \quad \forall \quad \phi_i \in V_h$$

Where $\tilde{u}$ is our approximation:

$$\tilde{u}(x) = \sum_{n=1}^{N} \bar{u}_n \phi_n(x)$$

in which $\bar{u}_n$ are the unknowns, socalled degrees of freedom and $\phi_n(x)$ are the basisfunctions of FE approximation space V_h. Substituting this approximation leads to:

$$\int_\Omega (\partial_t \sum_{j=1}^{N} \bar{u}_j\phi_j(x) + c\partial_x \sum_{j=1}^{N} \bar{u}_j\phi_j(x))\phi_i(x)dV = 0 \quad \forall \quad \phi_i \in V_h$$

Which we split into different integrals:

$$\int_\Omega \tilde{u}_t dV = \int_\Omega (\partial_t \sum_{j=1}^{N} \bar{u}_j\phi_j(x))\phi_i(x)dV = \sum_{j=1}^{N} \partial_t\bar{u}_j \int_\Omega \phi_j(x)\phi_i(x)dV \quad \forall \quad \phi_i \in V_h$$
$$\Rightarrow \qquad M\bar{u}$$

For the first integral we notice that the basis functions are constant through time, only the degrees of freedom $\bar{u}_j$ vary through time. Similarly these degrees of freedom does not affect the integral over space $\int_\Omega dV$. Thus we can write:

$$\begin{aligned}\int_\Omega \tilde{u}_t dV &= \int_\Omega (\partial_t \sum_{j=1}^{N} \bar{u}_j\phi_j(x))\phi_i(x)dV \quad \forall \quad \phi_i \in V_h \\ &= \sum_{j=1}^{N} \partial_t\bar{u}_j \int_\Omega \phi_j(x)\phi_i(x)dV \\ &\Rightarrow \qquad M\bar{u}\end{aligned}$$

where M is the mass matrix which combines the integral for all different basis functions. For the second term we aknoledge that the degrees of freedom have no spatial and temporal effects thus we can take them out of the integral and derivatives.

$$\begin{aligned}\int_\Omega u_x dV &= \int_\Omega (\partial_x \sum_{j=1}^{N} \bar{u}_j\phi_j(x))\phi_i(x)dV \quad \forall \quad \phi_i \in V_h \\ &= \sum_{j=1}^{N} \bar{u}_j \int_\Omega \partial_x\phi_j(x)\phi_i(x)dV \\ &\Rightarrow \qquad T\bar{u}\end{aligned}$$

where T is the socalled transport matrix, wich only depends on the basis functions.

Now we can write our PDE in terms of linear algabra objects:

$$M\bar{u}_t + cT\bar{u} = 0$$

which we modify to be in the format is expected by the temporal solvers:

$$M\bar{u}_t = K\bar{u}$$

Parameters

- **x** (*array_like(float)*) – Global coordinates of all degrees of freedom.
- **connect** (*array_like(int), shape((num_ele, dofe/ele))*) – Element to degree of freedom connectivety map.
- **c** (*float*) – Advective constant.
- **num_q** (*int*) – Number of Gausian quadrature points.
- **order** (*int*) – Order of the polynomial used by our element.

Returns

- **M** (*matrix, (sparse csr format)*) – The mass matrix.
- **K** (*matrix, (sparse csr format)*) – The combination of stiffness and transport matrix matrix scaled with the approprate constants $K = -cT$.
- **b** (*vector, (dense array)*) – The right hand side, because we consider a homogeneous PDE with diriclet conditions it is a zero vector.

`pde.`**`advectivediffusive`**(*mesh*, *c*, *mu*)

Time derivative of the PDE for advective diffusive problems.

$$u_t + cu_x - \mu u_{xx} = 0 \qquad \forall\, x \in \Omega = [0, L] \quad \& \quad t > 0$$

Which, is converted into a weak form through:

$$\int_\Omega (\tilde{u}_t + c\tilde{u}_x - \mu\tilde{u}_{xx})\phi_i(x)dV = 0 \quad \forall \quad \phi_i \in V_h$$

Where $\tilde{u}$ is our approximation:

$$\tilde{u}(x) = \sum_{n=1}^{N} \bar{u}_n \phi_n(x)$$

in which $\bar{u}_n$ are the unknowns, socalled degrees of freedom and $\phi_n(x)$ are the basisfunctions of FE approximation space V_h. Substituting this approximation leads to:

$$\int_\Omega (\partial_t \sum_{j=1}^{N} \bar{u}_j \phi_j(x) + c\partial_x \sum_{j=1}^{N} \bar{u}_j \phi_j(x) - \mu\partial_{xx} \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\phi_i(x)dV = 0 \quad \forall \quad \phi_i \in V_h$$

Which we split into different integrals:

$$\int_\Omega (\partial_t \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\phi_i(x)dV + \int_\Omega (c\partial_x \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\phi_i(x)dV - \int_\Omega (\mu\partial_{xx} \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\phi_i(x)dV = 0 \quad \forall \quad \phi_i \in V_h$$

For the first integral we notice that the basis functions are constant through time, only the degrees of freedom $\bar{u}_j$ vary through time. Similarly these degrees of freedom does not affect the integral over space $\int_\Omega dV$. Thus we can write:

$$\begin{aligned}\int_\Omega \tilde{u}_t dV &= \int_\Omega (\partial_t \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\phi_i(x)dV \quad \forall \quad \phi_i \in V_h \\ &= \sum_{j=1}^{N} \partial_t \bar{u}_j \int_\Omega \phi_j(x)\phi_i(x)dV \\ &\Rightarrow \qquad M\bar{u}\end{aligned}$$

where M is the mass matrix which combines the integral for all different basis functions. For the second term we aknoledge that the degrees of freedom have no spatial and temporal effects thus we can take them out of the integral and derivatives.

$$\begin{aligned}\int_\Omega u_x dV &= \int_\Omega (\partial_x \sum_{j=1}^N \bar{u}_j \phi_j(x))\phi_i(x) dV \quad \forall \quad \phi_i \in V_h \\ &= \sum_{j=1}^N \bar{u}_j \int_\Omega \partial_x \phi_j(x)\phi_i(x) dV \\ &\Rightarrow \qquad T\bar{u}\end{aligned}$$

where T is the socalled transport matrix, wich only depends on the basis functions. For the thrird part we apply integration by parts, while assuming that Neumann boundary conditions:

$$\begin{aligned}\int_\Omega \tilde{u}_{xx} dV \quad &= \int_\Omega (\partial_{xx} \sum_{j=1}^N \bar{u}_j \phi_j(x))\phi_i(x) dV \quad \forall \quad \phi_i \in V_h \\ &= -\int_\Omega (\partial_x \sum_{j=1}^N \bar{u}_j \phi_j(x))\partial_x \phi_i(x) dV \\ &= \sum_{j=1}^N \bar{u}_j \int_\Omega -\partial_x \phi_j(x)\partial_x \phi_i(x) dV \\ &\Rightarrow S\bar{u}\end{aligned}$$

where S is the stiffness matrix, which can be computed independently from the actual unknowns. Now we can write our PDE in terms of linear algebra objects:

$$M\bar{u}_t + cT\bar{u} - \mu S\bar{u} = 0$$

which we modify to be in the format is expected by the temporal solvers:

$$M\bar{u}_t = K\bar{u}$$

Parameters

- **x** (*array_like(float)*) – Global coordinates of all degrees of freedom.
- **connect** (*array_like(int), shape((num_ele, dofe/ele))*) – Element to degree of freedom connectivety map.
- **c** (*float*) – Advective constant.
- **mu** (*float*) – Diffusive constant.
- **num_q** (*int*) – Number of Gausian quadrature points.
- **order** (*int*) – Order of the polynomial used by our element.

Returns

- **M** (*matrix, (sparse csr format)*) – The mass matrix.
- **K** (*matrix, (sparse csr format)*) – The combination of stiffness and transport matrix matrix scaled with the approprate constants $K = \mu S - cT$.
- **b** (*vector, (dense array)*) – The right hand side, because we consider a homogeneous PDE with diriclet conditions it is a zero vector.

`pde.`**`diffusive`**(*x*, *connect*, *mu*, *num_q*, *order*)

Time derivative of the PDE for diffusion problems.

$$u_t - \mu u_{xx} = 0 \qquad \forall\, x \in \Omega = [0, L] \quad \& \quad t > 0$$

Which, is converted into a weak form through:

$$\int_\Omega (\tilde{u}_t - \mu \tilde{u}_{xx})\phi_i(x) dV = 0 \quad \forall \quad \phi_i \in V_h$$

Where $\tilde{u}$ is our approximation:

$$\tilde{u}(x) = \sum_{n=1}^{N} \bar{u}_n \phi_n(x)$$

in which $\bar{u}_n$ are the unknowns and $\phi_n(x)$ are the basisfunctions of FE approximation space V_h. Substituting this approximation leads to:

$$\int_\Omega (\partial_t \sum_{j=1}^{N} \bar{u}_j \phi_j(x) - \mu \partial_{xx} \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\phi_i(x) dV = 0 \quad \forall \quad \phi_i \in V_h$$

Which we split into different integrals:

$$\int_\Omega (\partial_t \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\phi_i(x) dV - \int_\Omega (\mu \partial_{xx} \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\phi_i(x) dV = 0 \quad \forall \quad \phi_i \in V_h$$

For the first integral we notice that the basis functions are constant through time, only the degrees of freedom $\bar{u}_j$ vary through time. Similarly these degrees of freedom does not affect the integral over space $\int_\Omega dV$. Thus we can write:

$$\begin{aligned}\int_\Omega \tilde{u}_t dV &= \int_\Omega (\partial_t \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\phi_i(x) dV \quad \forall \quad \phi_i \in V_h \\ &= \sum_{j=1}^{N} \partial_t \bar{u}_j \int_\Omega \phi_j(x)\phi_i(x) dV \\ &\Rightarrow \qquad M\bar{u}\end{aligned}$$

where M is the mass matrix which combines the integral for all different basis functions. For the second term we apply integration by parts, while assuming that Neumann boundary conditions:

$$\begin{aligned}\int_\Omega \tilde{u}_{xx} dV \quad &= \int_\Omega (\partial_{xx} \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\phi_i(x) dV \quad \forall \quad \phi_i \in V_h \\ &= -\int_\Omega (\partial_x \sum_{j=1}^{N} \bar{u}_j \phi_j(x))\partial_x \phi_i(x) dV \\ &= \sum_{j=1}^{N} \bar{u}_j \int_\Omega -\partial_x \phi_j(x)\partial_x \phi_i(x) dV \\ &\Rightarrow S\bar{u}\end{aligned}$$

where S is the stiffness matrix, which can be computed independently from the actual unknowns. Now we can write our PDE in terms of linear algabra objects:

$$M\bar{u}_t - \mu S\bar{u} = 0$$

which we modify to be in the format is expected by the temporal solvers:

$$M\bar{u}_t = K\bar{u}$$

Parameters

- **x** (*array_like(float)*) – Global coordinates of all degrees of freedom.
- **connect** (*array_like(int), shape((num_ele, dofe/ele))*) – Element to degree of freedom connectivety map.
- **mu** (*float*) – Diffusive constant.
- **num_q** (*int*) – Number of Gausian quadrature points.
- **order** (*int*) – Order of the polynomial used by our element.

Returns

- **M** (*matrix, (sparse csr format)*) – The mass matrix.
- **K** (*matrix, (sparse csr format)*) – The stiffeness matrix scaled with the diffusivity constant $K = \mu S$.
- **b** (*vector, (dense array)*) – The right hand side, because we consider a homogeneous PDE with diriclet conditions it is a zero vector.

`pde.projection(`*mesh*, *fun*`)`

Projecting a 1D function $f(x)$ on a finite element basis.

Lets create our approximation function,

$$f_h(x) = \sum_{n=0}^{N} \bar{u}_n \phi_n(x)$$

as a weighted summation of the basisfunctions of approximation. Where phi_n are the basisfunctions of our FE space V_h. The unknows here are the weights $\bar{u}_n$, these we call degrees of freedom. To find these DOFs we formulate a weak form:

$$\int_\Omega \left(f_h(x) - f(x)\right) \phi_i(x)\, dV = 0 \quad \forall \quad \phi_i \in V_h$$

in which we substitute our approximation function and separate the knowns from the unknowns. We find:

$$\int_\Omega \phi_i(x) \sum_{j=0}^{N} \bar{u}_j \phi_j(x)\, dV = \int_\Omega \phi_i(x) f(x)\, dV \quad \forall \quad \phi_i \in V_h$$

As the weights $\bar{u}_n$ are independent of location, we can take them out of the integral:

$$\sum_{j=0}^{N} \bar{u}_j \int_\Omega \phi_i(x)\phi_j(x)\, dV = \int_\Omega \phi_i(x) f(x)\, dV \quad \forall \quad \phi_i \in V_h$$

Which can be rewritten as a system of linear equations, which is:

$$M\, \bar{u} = b$$

Where M is a matrix and $\bar{u}$ and b are vectors.

Parameters

- **mesh** (`Mesh`) – The mesh object which specifies all discretization.
- **fun** (*callable*) – Function that acts as our right hand side (nonhomogeneous term).

Returns

- **M** (*matrix, (sparse csr format)*) – The mass matrix.
- **b** (*vector, (dense array)*) – The right hand side, caused by the non-homogeneous behavior.

1.6 FEM Kernel

The main FEM loop.

This main FEM loop will assample the differenc matrices that are required in a FEM solver. It takes the following steps:

1. Loop over all elements.
2. **Compute the element based integrals.**
    1. calculate the quantities in the reference element.
    2. integrate using Quadrature rules and include the mapping from reference to global axis system.
3. Assamble these element based contributions into a global operator.

Bram Lagerweij COHMAS Mechanical Engineering KAUST 2021

`fem.`**`element_mass`**(*phi_xq*, *wq_detJ*)

Compute the elmement mass matrix.

This matrix is defined as:

$$
\begin{aligned}
M &= \int_\Omega \phi_j(x)\ \phi_i(x)\ dV \quad \forall \quad \phi_i, \phi_j \in V_h \\
&= \sum_{e=0}^{N} \int_{\Omega_e} \phi_j(x)\ \phi_i(x)\, dV \quad \forall \quad \phi_i, \phi_j \in V_h^e \\
&= \sum_{e=0}^{N} M_e
\end{aligned}
$$

but here we only compute the portion contributed by our current element. Hence we only need to consider the trial functions within each elements. when integrated in reference element coordinates, ξ this is:

$$M = \int_0^1 \phi_j(\xi)\phi_i(\xi) det(J) dV$$

To evaluate these integras Gaussian quadrature is used such that thi integal becomes:

$$M = \sum_{q=0}^{N_q} \phi_j(\xi_q)\phi_i(\xi_q) det(J) w_q$$

Parameters

- **`phi_xq`** (*`array_like(float), shape((dofe, num_q))`*) – For each shape function the value at the quadrature points.
- **`wq_detJ`** (*`array_like(float), shape((dofe, num_q))`*) – Integration weight including the mapping from local to global coordinates.

Returns **me** – Element mass matrix.

Return type array_like(float), shape((dofe, dofe))

`fem.`**`element_rhs`**(*phi_xq*, *wq_detJ*, *f_xq*)

Compute the elmement right hand side vector.

Parameters

- **`phi_xq`** (*`array_like(float), shape((dofs, num_q))`*) – For each shape function the value at the quadrature points.

- **f_xq** (*array_like(float), shape(num_q)*) – The value of the right hand side equation evaluated at the quadrature points.
- **wq_detJ** (*array_like(float), shape((dofs, num_q))*) – Integration weight including the mapping from local to global coordinates.

Returns **fe** – Element right hand side in our system of equations.

Return type array_like(float), shape(dofe)

`fem.`**`element_stiffness`**(*invJ_dphi_xq*, *wq_detJ*)

Compute the elmement stiffness matrix.

This matrix is defined as:

$$\begin{aligned} S &= \int_\Omega -\partial_x\phi_j(x)\ \partial_x\phi_i(x)\ dV \quad \forall \quad \phi_i, \phi_j \in V_h \\ &= \sum_{e=0}^{N} \int_{\Omega_e} -\partial_x\phi_j(x)\ \partial_x\phi_i(x)\ dV \quad \forall \quad \phi_i, \phi_j \in V_h^e \\ &= \sum_{e=0}^{N} S_e \end{aligned}$$

but here we only compute the portion contributed by our current element. Hence we only need to consider the trial functions within each elements. When integrated in reference element coordinates, ξ this is:

$$S_e = \int_0^1 J^{-1}\,\partial_\xi\phi_j(\xi)\ J^{-1}\,\partial_\xi\phi_i(\xi)\ det(J)dV$$

To evaluate these integras Gaussian quadrature is used such that thi integal becomes:

$$S_e = \sum_{q=0}^{N_q} J^{-1}\,\partial_\xi\phi_j(\xi_q)\ J^{-1}\,\partial_\xi\phi_i(\xi_q)\ det(J)w_q$$

Parameters

- **invJ_dphi_xq** (*array_like(float), shape((dofs, num_q))*) – For each shape function its derivative value at the quadrature points times the inverse Jacobian.
- **wq_detJ** (*array_like(float), shape((dofe, num_q))*) – Integration weight including the mapping from local to global coordinates.

Returns **Se** – Element mass matrix.

Return type array_like(float), shape((dofe, dofe))

`fem.`**`element_transport`**(*phi_xq*, *invJ_dphi_xq*, *wq_detJ*)

Compute the elmement transport matrix.

This matrix is defined as:

$$\begin{aligned} T &= \int_\Omega \partial_x\phi_j(x)\ \phi_i(x)\ dV \quad \forall \quad \phi_i, \phi_j \in V_h \\ &= \sum_{e=0}^{N} \int_{\Omega_e} \partial_x\phi_j(x)\ \phi_i(x)\,dV \quad \forall \quad \phi_i, \phi_j \in V_h^e \\ &= \sum_{e=0}^{N} T_e \end{aligned}$$

but here we only compute the portion contributed by our current element. Hence we only need to consider the trial functions within each elements. when integrated in reference element coordinates, ξ this is:

$$T_e = \int_0^1 J^{-1}\,\partial_\xi \phi_j(\xi)\ \phi_i(\xi)\ det(J)dV$$

To evaluate these integras Gaussian quadrature is used such that thi integal becomes:

$$T_e = \sum_{q=0}^{N_q} J^{-1}\,\partial_\xi \phi_j(\xi_q)\ \phi_i(\xi_q)\ det(J)w_q$$

Parameters

- **phi_xq** (*array_like(float), shape((dofe, num_q))*) – For each shape function the value at the quadrature points.
- **invJ_dphi_xq** (*array_like(float), shape((dofs, num_q))*) – For each shape function its derivative value at the quadrature points times the inverse Jacobian.
- **wq_detJ** (*array_like(float), shape((dofe, num_q))*) – Integration weight including the mapping from local to global coordinates.

Returns **Te** – Element mass matrix.

Return type array_like(float), shape((dofe, dofe))

fem.**interpolate**(*mesh*, *u*, *x_inter*)

Obtain the field $u(x)$ any points *x_inter* following the FE interpolation.

Parameters

- **mesh** (Mesh) – The mesh class specifying all discretization.
- **u** (*array_like(float), shape(dofs)*) – The field *u* at the degrees of freedom.
- **x_inter** (*array_like(float)*) – The location where we want to obtain our interpolated field.

Returns The field *u* at the interpolation points *x_inter*.

Return type array_like(float)

fem.**kernel1d**(*mesh*, *rhs=None*, *mass=False*, *transport=False*, *stiffness=False*)

Create the global FEM system by looping over the elements.

Parameters

- **mesh** (Mesh) – The mesh class specifying all discretization.
- **rhs** (*callable*) – Function that acts as our right hand side (nonhomogeneous term), set equal to *None* if the rhs is zero valued.
- **mass** (*bool, optional*) – Return a mass matrix. Default is *False*.
- **transport** (*bool, optional*) – Return the transport matrix. Default is *False*.
- **stiffness** (*bool, optional*) – Return the stiffness matrix. Default is *False*.

Returns

- **f** (*array_like(float), shape(dofe)*) – Global right hand side in our system of equations. Only when *rhs != None*, *None* otherwise.
- **M** (*COO (value, (row, column))*) – Global mass matrix, ready to be converted to COO. Repeating indices do exist. Only *mass == True*, *None* otherwise.

- **T** (*COO (value, (row, column))*) – Global transport matrix, ready to be converted to COO. Repeating indices do exist. Only ‘transport == True`, *None* otherwise.
- **S** (*COO (value, (row, column))*) – Global stiffness matrix, ready to be converted to COO. Repeating indices do exist. Only ‘stiffness == True`, *None* otherwise.

1.7 Elements and Meshes

Discretization objects, containing both the meshing and the solution space approximation.

That is inside this object is both the h and p discretization. These are in this code orginized together as no local p refinement is expeced. There is a base class, Mesh, specifying the interface to the main kernel and solver, and there are the following interited classes specifying:

1. Mesh1D for a 1D mesh of different approximation orders p.
2. Mesh2Dtri for a 2D mesh of triangles.
3. Mesh2Dqua for a 2D mesh of quadralatirals.

And although these have the name ‘mesh’ they do describe the elements as well.

Bram Lagerweij COHMAS Mechanical Engineering KAUST 2021

1.7.1 Base Mesh

`class element.`**`Mesh`**

Specify a base mesh object, and it’s behaviour.

This base class is not an actual usefull mesh but provides the basics outline that should be in all other mesh classes. All other meshes should be childeren from this base class. But inheritance works badly with the just in time compilation of *numba*. Hence all function have to be redefined in the child classes, while the class requires the *@nb.jitclass(spec)* decorator.

`num_ele`

Number of elements in the entire mesh.

Type int

`num_dofe`

Number of degrees of freedom per element.

Type int

`num_dofs`

Number of degrees of freedom in the problem, this property depends on the element type and the mesh.

Type int

`num_q`

Number of quadrature point in integration approximations.

Type int

`nodes`

For each node in each element the coordinates.

Type array_like(float), shape(n+1, order+1)

`connectivity`

Elements to node connectivity array.

Type array_like(int), shape(n+1, order+1)

`dshape`(*xi*)
Shape functions derivatives at locations ξ in element coordinates system.

Parameters **xi** (*array_like(float)*) – Locations ξ where the shape functions are evaluated.

Returns **dphi_xq** – Shape functions derivatives ϕ_i at locations ξ.

Return type array_like(float)

`get_element`(*ele*, *rhs=None*)
Get the element properties of element *ele*.

Parameters

- **ele** (*int*) – Number of the element of which the properties should be obtained.
- **rhs** (*callable(float)*) – The righthandside function of the pde in terms of global coordinates.

Returns

- **dofe** (*array_like(int), shape(num_dofe)*) – The degrees of freedom that belong to this element.
- **phi_xq** (*array_like(float), shape((dofs, num_q))*) – For each shape function the value at the quadrature points.
- **invJ_dphi_xq** (*array_like(float), shape((dofs, num_q))*) – For each shape function its derivative value at the quadrature points times the inverse Jacobian.
- **f_xq** (*array_like(float), shape(num_q)*) – The value of the right hand side equation evaluated at the quadrature points.
- **wq_detJ** (*array_like(float), shape((dofs, num_q))*) – For the local determinant times quadrature weight at each of the quadrature points.

`jacobian`(*ele*, *xi*)
The jacobian and mapping for the local to global coordinates system (ξ to x).

Parameters

- **ele** (*int*) – Element for which the jacobian has to be calculated.
- **xi** (*array_like(float)*) – Location where the jacobians should be measured.

Returns

- **jac** (*array_like(float)*) – The Jacobian at ξ.
- **invJ** (*array_like(float)*) – The inverse Jacobian at ξ.
- **detJ** (*array_like(float)*) – The derivative of the Jacobian at ξ.

`shape`(*xi*)
Shape functions at locations ξ in element coordinates system.

Parameters **xi** (*array_like(float)*) – Locations ξ where the shape functions are evaluated.

Returns **phi_xq** – Shape functions ϕ_i at locations ξ.

Return type array_like(float)

`x_to_xi`(*ele*, *x*)
Converting global into local coordinates $x \rightarrow \xi$.

Parameters

- **ele** (*int*) – Element in which the transformation has to take place.
- **x** (*array_like(float)*) – Global coordinates, these must be within the element.

Returns **xi** – The local, element, coordinates.

Return type array_like(float)

`xi_to_x`(*ele*, *xi*)

Converting local coordinates into global ones $\xi \to x$.

Parameters

- **ele** (*int*) – Element in which the transformation has to take place.
- **xi** (*array_like(float)*) – Local coordinates within the element.

Returns **x** – The global coordinates.

Return type array_like(float)

1.7.2 Implemented Meshes

class element.`Mesh1D`(*x_start*, *x_end*, *num_ele*, *order*, *num_q*, *periodic=False*)

Specify a 1D mesh object, and it's behaviour.

This is a 1D mesh object with Lagransian basis functions.

Parameters

- **x_start** (*float*) – Start coordinate of the domain.
- **x_end** (*float*) – End coordinate of the domain.
- **num_ele** (*int*) – Number of elements in the mesh.
- **order** (*int*) – Polynomial order of the Lagransian basis functions.
- **num_q** (*int*) – Number of quadrature points per element.
- **periodic** (*bool, optional*) – Whether the domain is periodic, default is *False*.

`num_ele`

Number of elements in the entire mesh.

Type int

`order`

Order of the polynomaial approximation.

Type int

`num_dofe`

Number of degrees of freedom per element.

Type int

`num_dofs`

Number of degrees of freedom in the problem, this property depends on the element type and the mesh.

Type int

`num_q`

Number of quadrature point in integration approximations.

Type int

`nodes`
For each node in each element the coordinates.

Type array_like(float), shape(n+1, order+1)

`connectivity`
Elements to node connectivity array.

Type array_like(int), shape(n+1, order+1)

`get_element`(*ele*, *rhs=None*)
Get the element properties of element *ele*.

Parameters

- `ele` (`int`) – Number of the element of which the properties should be obtained.
- `rhs` (`callable(float), optional`) – The righthandside function of the pde in terms of global coordinates.

Returns

- **dofe** (*array_like(int), shape(num_dofe)*) – The degrees of freedom that belong to this element.
- **phi_xq** (*array_like(float), shape((dofs, num_q))*) – For each shape function the value at the quadrature points.
- **invJ_dphi_xq** (*array_like(float), shape((dofs, num_q))*) – For each shape function its derivative value at the quadrature points times the inverse Jacobian.
- **f_xq** (*array_like(float), shape(num_q)*) – The value of the right hand side equation evaluated at the quadrature points.
- **wq_detJ** (*array_like(float), shape((dofs, num_q))*) – For the local determinant times quadrature weight at each of the quadrature points.

`x_to_xi`(*ele*, *x*)
Converting local coordinates into global ones $x \to \xi$.

Parameters

- `ele` (`int`) – Element in which the transformation has to take place.
- `x` (`array_like(float)`) – Global coordinates, these must be within the element.

Returns **xi** – The local, element, coordinates.

Return type array_like(float)

`xi_to_x`(*ele*, *xi*)
Converting local coordinates into global ones $\xi \to x$.

Parameters

- `ele` (`int`) – Element in which the transformation has to take place.
- `xi` (`array_like(float)`) – Local coordinates within the element.

Returns **x** – The global coordinates.

Return type array_like(float)

`jacobian`(*ele*)
The jacobian and mapping for the local to global coordinates system (ξ to x).

Because the jacobian is a constant for 1D meshes, the objectes that are returned are constant floats instead of arrays.

Parameters **`ele`** (*`int`*) – Element for which the jacobian has to be calculated.

Returns

- **jac** (*float*) – The Jacobian at ξ.
- **invJ** (*float*) – The inverse Jacobian at ξ.
- **detJ** (*float*) – The derivative of the Jacobian at ξ.

`shape`(*xi*)

Shape functions at locations ξ in element coordinates system.

Parameters **`xi`** (*`array_like(float)`*) – Locations ξ where the shape functions are evaluated.

Returns **phi_xq** – Shape functions ϕ_i at locations ξ.

Return type array_like(float)

`dshape`(*xi*)

Shape functions derivatives at locations ξ in element coordinates system.

Parameters **`xi`** (*`array_like(float)`*) – Locations ξ where the shape functions are evaluated.

Returns **dphi_xq** – Shape functions derivatives ϕ_i at locations ξ.

Return type array_like(float)

1.8 Solvers and Time Integration

Various implementations of the method of lines to progress through time. The goal is to implement the code in python and not rely on existing solvers.

Bram Lagerweij COHMAS Mechanical Engineering KAUST 2021

`solvers.`**`backwardEuler`**(*pde*, *u*, *dt*, *t_end*)

Itterate a through time with the backward Eurler method.

Lets assume that, through any type of discretization, the time derivative was obtained. This time derivative can be represented through linear algabra as:

$$M\,u_t = K\,u + b \qquad \text{that is} \qquad u_t = M^{-1}(K\,u + b)$$

where M is the mass matrix, K the siffness and transport matrix and vector b the right hand side. these are obtained from approximations of the spatial derivatives defined by the functien provided to *func*

The backward Euler method predicts the field of our function based upon information of the previous timestep only. Imagine that we are at timestep n and want to predict our field at timestep $u^{(n+1)}$. Now a backward finite difference approximation used the time derivative of the next timestep, wich is not yet known:

$$u_t^{(n+1)} = \frac{-u^{(n)} + u^{(n+1)}}{dt}$$

That is we can predict our field in the future timestep as:

$$u^{(n+1)} = u^{(n)} + dt\,u_t^{(n+1)}$$

in which we substitute the linear algabra representation of our PD.

$$u^{(n+1)} = u^{(n)} + dt\,M^{-1}(Ku^{n+1} + b)$$

It is important to notic that there is a term with an unknown, as that is at time step :math:`n+1' on both sides of the equation. Now we rewrite it into a system of equations where we find all unknowns on the left hand side and all knownn on the right hand side.

$$(M - dt\,K)\,u^{(n+1)} = M\,u^{(n)} + dtb$$

This is a system of equations which can be solved.

Parameters

- **pde** (*tuple*) – The linear algabra objects of the pde $M\,u_t = K\,u + b$.
- **u** (*array_like*) – The field at the start $u(t=0)$.
- **dt** (*float*) – The size of the time step.
- **t_end** (*float*) – Time at termination.

Returns The function for all time steps.

Return type array_like

solvers.**forwardEuler**(*pde*, *u*, *dt*, *t_end*)

Itterate a through time with the forward Eurler method.

Lets assume that, through any type of discretization, the time derivative was obtained. This time derivative can be represented through linear algabra as:

$$M\,u_t = K\,u + b \qquad \text{that is} \qquad u_t = M^{-1}(Ku + b)$$

where M is the mass matrix, K the siffness and transport matrix and vector b the right hand side. these are obtained from approximations of the spatial derivatives defined by the functien provided to *func*.

The backward Euler method predicts the field of our function based upon information of the previous timestep only. Imagine that we are at timestep n and want to predict our field at timestep $u^{(n+1)}$. Now a forward finite difference approximation is used:

$$u_t^{(n)} = \frac{-u^{(n)} + u^{(n+1)}}{dt}$$

That is we can predict our field in the future timestep as:

$$u^{(n+1)} = u^{(n)} + dt\,u_t^{(n)}$$

Now from our linear algebra implementation we substitute u_t

$$u^{(n+1)} = u^{(n)} + dt\,M^{-1}(Ku^{(n)} + b)$$

most important of all is to see that everything on the right hand side is exactly known. Thus the updated field can be calculated directly. However For this purpouse we would have to invert the mass matrix. If the mass matrix is the identity matrix this is simple, but in generally this is not the case. As we don't want to invert large matrices, we multiply all terms by M.

$$Mu^{(n+1)} = Mu^{(n)} + dt\,(Ku^{(n)} + b)$$

Which is a system of equations as everything on the right hand side is known and can be calculated directly.

Notes

This code will recognize if M is the identity matrix and, in that case it will solve the problem directly, avoiding the need to solve a sytem of equations.

Parameters

- **pde** (*tuple*) – The linear algebra objects of the pde $M\,u_t = K\,u + b$.
- **u** (*array_like*) – The field at the start $u(t = 0)$.
- **dt** (*float*) – The size of the step.
- **t_end** (*float*) – Time at termination.

Returns The function for all time steps.

Return type array_like

`solvers.solve(K, b)`

Solve a time independed problem.

Parameters

- **func** (*callable*) – The linear algebra problem that we want to solve $K\,u = b$.
- **args** (*tuple, optional*) – The parameters into the PDE approximation. Defealts to an empty tuple.

Returns The vector containing u.

Return type array_like

1.9 Helper Scripts

Minor helper functions for FEM problems.

Bram Lagerweij COHMAS Mechanical Engineering KAUST 2021

`helper.E1(fun, fun_h, x)`

Calculate the E_1 error.

$$E_1 := \int_\Omega |f(x) - f_h(x)| dx$$

Parameters

- **fun** (*array_like*) – The solution of the exact equation at location x.
- **fun_h** (*array_like*) – The solution of the approximation equation at location x.
- **x** (*array_like*) – The locations where the function is analyzed.

Returns Error of the approximation.

Return type float

`helper.E2(fun, fun_h, x)`

Calculate the E_2 error.

$$E_2 := \sqrt{\int_\Omega \big(f(x) - f_h(x)\big)^2 dx}$$

Parameters

- **fun** (*array_like*) – The solution of the exact equation at location x.
- **fun_h** (*array_like*) – The solution of the approximation equation at location x.
- **x** (*array_like*) – The locations where the function is analyzed.

Returns Error of the approximation.

Return type float

`helper.gauss(num)`

Gaussian integration points and weights for *num* sample points.

Computes the sample points and weights for Gauss-Legendre quadrature. These sample points and weights will correctly integrate polynomials of degree $2 \cdot num - 1$ or less over the interval $[0, 1]$ with the weight function $f(x) = 1$.

Parameters **num** (*int*) – Number of sample points and weights. It must be 1 <= num <= 5.

Returns

- **xi** (*array_like(float)*) – 1D array containing the sample points.
- **w** (*array_like(float)*) – 1D array containing the weights at the sample points.

`helper.quadtri(num)`

Symetric quadrature points and weights for *num* sample points in a triangle.

Computes the sample points and weights through the Dunavant unnit trianglue rule[1]. These sample points and weights will correctly integrate polynomials of:

Table 1.4: : Quadrature with *num* points results in exact integrals for polynomial of order p.

num	1	3	4	7
p	1	2	3	4

Parameters **num** (*int*) – Number of sample points and weights. It must be 1, 3, 4, or 7.

Returns

- **xi** (*array_like(float)*) – 1D array containing the sample points in local coordinates.
- **w** (*array_like(float)*) – 1D array containing the weights at the sample points.

References

1.10 Finite Differences

Finite difference example problems.

- Diffusive 1D

$$u_t - \mu u_{xx} = 0 \qquad \forall x \in \Omega = [0, L] \quad \& \quad t > 0$$

- Advective 1D

[1] Dunavant, D.A. (1985), High degree efficient symmetrical Gaussian quadrature rules for the triangle. Int. J. Numer. Meth. Engng., 21: 1129-1148. DOI:10.1002/nme.1620210612

$$u_t + cu_x = 0 \qquad \forall\, x \in \Omega = [0, L] \quad \& \quad t > 0$$

- Diffusive-Advective 1D

$$u_t + cu_x - \mu u_{xx} = 0 \qquad \forall\, x \in \Omega = [0, L] \quad \& \quad t > 0$$

- Poisson in 1D

$$-cu_{xx} = f(x) \qquad \forall\, x \in \Omega = [0, L]$$

Bram Lagerweij COHMAS Mechanical Engineering KAUST 2021

`finitedifference.`**`Dx`**`(`*dof*, *dx*, *bc='periodic'*`)`

Return the central differences matrix for the first derivative. That is the matrix D_x represents the central difference approximation of ∂_x in 1D axis systems.

Parameters

- **dof** (*int*) – Number of spacial degrees of freedom.
- **dx** (*float*) – Spacial step size.
- **bc** (*str, optional*) – The type of boundary condition to be used. The default is ‘periodic’.

Raises **NotImplementedError** – Is raised when the requested boundary condition is not implemented.

Returns The central difference approximation of the first derivative.

Return type matrix (sparse csr format)

Notes

The following boundary conditions are possible:

- ‘periodic’ (default) that the first and last dofs are representing the same point. As a result the derivative of the first point depends on the second last point and the derivative of the last point will depend on the second point as well.
- ‘none’ means that the row of the first and last degree of freedom are left empty. This will result in a singular matrix, thus extra constraints will have to be defined before solving a system with such a matrix.

`finitedifference.`**`Dxx`**`(`*dof*, *dx*, *bc='periodic'*`)`

Return the central differences matrix for the second derivative. That is the matrix D_{xx} represents the central difference approximation of ∂_{xx} in 1D axis systems.

Parameters

- **dof** (*int*) – Number of spacial degrees of freedom.
- **dx** (*float*) – Spacial step size.
- **bc** (*str, optional*) – The type of boundary condition to be used. The default is ‘periodic’.

Raises **NotImplementedError** – Is raised when the requested boundary condition is not implemented.

Returns The central difference approximation of the first derivative.

Return type matrix (sparse csr format)

Notes

The following boundary conditions are possible:

- 'periodic' (defeat) that the first and last dofs are representing the same point. As a result the derivative of the first point depends on the second last point and the derivative of the last point will depend on the second point as well.
- 'none' means that the row of the first and last degree of freedom are left empty. This will result in a singular matrix, thus extra constraints will have to be defined before solving a system with such a matrix.

`finitedifference.`**`advective`**`(`*dof*, *dx*, *c*`)`

Time derivative of the PDE for advective diffusive problems.

$$u_t + cu_x = 0 \qquad \forall\, x \in \Omega = [0, L] \quad \& \quad t > 0$$

Thus this returns:

$$u_t = -cu_x$$

Because we use finite difference based matrix products we can convert this into a matrix vector product, where D_x is the central difference approximation of ∂_x:

$$u_t = -cD_x u = Ku$$

This function calculates the matrix K. Because it should be compatible with general, non-homogeneous formulation, a part that is independent of u is also included.

Parameters

- **`dof`** (*`int`*) – Number of degrees of freedom.
- **`dx`** (*`float`*) – Step size in the of spatial discretization.
- **`c`** (*`float`*) – The advective coefficient.

Returns

- **M** (*matrix (sparse csr format)*) – The mass matrix, which will equal the identity matrix in finite differenc problems.
- **K** (*matrix (sparse csr format)*) – The time derivative part of the pde obtained from the spatial part.
- **b** (*vector (dense array)*) – The remaining term, in this homogeneous case it is a zero array.

`finitedifference.`**`advectivediffusive`**`(`*dof*, *dx*, *mu*, *c*`)`

Time derivative of the PDE for advective diffusive problems.

$$u_t + cu_x = \mu u_{xx} \qquad \forall\, x \in \Omega = [0, L] \quad \& \quad t > 0$$

Thus this returns:

$$u_t = -cu_x + \mu u_{xx}$$

Because we use finite difference based matrix products we can convert this into a matrix vector product, where D_x is the central difference approximation of ∂_x and similarly D_{xx} the central difference approximation of ∂_{xx}:

$$u_t = -cD_x u + \mu D_{xx} u = (-cD_x + \mu D_{xx})\, u = Ku$$

This function calculates the matrix K. Because it should be compatible with general, non-homogeneous formulation, a part that is independent of u is also included.

Parameters

- **dof** (*int*) – Number of degrees of freedom.
- **dx** (*float*) – Step size in the of spatial discretization.
- **mu** (*float*) – The diffusive coefficient.
- **c** (*float*) – The advective coefficient.

Returns

- **M** (*matrix (sparse csr format)*) – The mass matrix, which will equal the identity matrix in finite differenc problems.
- **K** (*matrix (sparse csr format)*) – The time derivative part of the pde obtained from the spatial part.
- **b** (*vector (dense array)*) – The remaining term, in this homogeneous case it is a zero array.

`finitedifference.`**`diffusive`**`(`*dof*, *dx*, *mu*`)`

Time derivative of the PDE for advective diffusive problems.

$$u_t = \mu u_{xx} \qquad \forall x \in \Omega = [0, L] \quad \& \quad t > 0$$

Thus this returns:

$$u_t = \mu u_{xx}$$

Because we use finite difference based matrix products we can convert this into a matrix vector product, where $D_x x$ is the central difference approximation of ∂_{xx}:

$$u_t = \mu D_{xx} u = K u$$

This function calculates the matrix K. Because it should be compatible with general, non-homogeneous formulation, a part that is independent of u is also included.

Parameters

- **dof** (*int*) – Number of degrees of freedom.
- **dx** (*float*) – Step size in the of spatial discretization.
- **mu** (*float*) – The diffusive coefficient.

Returns

- **M** (*matrix (sparse csr format)*) – The mass matrix, which will equal the identity matrix in finite differenc problems.
- **K** (*matrix (sparse csr format)*) – The time derivative part of the pde obtained from the spatial part.
- **b** (*vector (dense array)*) – The remaining term, in this homogeneous case it is a zero array.

`finitedifference.`**`poisson`**`(`*dof*, *dx*, *f*, *c=1*`)`

Problem formulation of a Poisson equation.

$$-c u_{xx} = f(x) \qquad \forall x \in \Omega = [0, L]$$

Because we use finite difference based matrix products we can convert this into a matrix vector product, where D_{xx} the is the central difference approximation of ∂_{xx}:

$$D_{xx}u = Ku = f/c$$

This function calculates the matrix K and the forcing vector f. The matrix is however singular as no boundary conditions are specified.

Parameters

- **dof** (*int*) – Number of degrees of freedom.
- **dx** (*float*) – Step size in the of spatial discretization.
- **f** (*callable*) – A function to calculate the forcing term for any location x.
- **c** (*float, optional*) – A scalar multiplying the derivative.

Returns

- **K** (*matrix (sparse csr format)*) – The stiffness matrix.
- **b** (*vector (dense array)*) – The right hand side, caused by the non-homogeneous behavior.

1.11 Mozilla Public License Version 2.0

1.11.1 1. Definitions

1.1. "Contributor" means each individual or legal entity that creates, contributes to the creation of, or owns Covered Software.

1.2. "Contributor Version" means the combination of the Contributions of others (if any) used by a Contributor and that particular Contributor's Contribution.

1.3. "Contribution" means Covered Software of a particular Contributor.

1.4. "Covered Software" means Source Code Form to which the initial Contributor has attached the notice in Exhibit A, the Executable Form of such Source Code Form, and Modifications of such Source Code Form, in each case including portions thereof.

1.5. "Incompatible With Secondary Licenses" means

(a) that the initial Contributor has attached the notice described in Exhibit B to the Covered Software; or

(b) that the Covered Software was made available under the terms of version 1.1 or earlier of the License, but not also under the terms of a Secondary License.

1.6. "Executable Form" means any form of the work other than Source Code Form.

1.7. "Larger Work" means a work that combines Covered Software with other material, in a separate file or files, that is not Covered Software.

1.8. "License" means this document.

1.9. "Licensable" means having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently, any and all of the rights conveyed by this License.

1.10. "Modifications" means any of the following:

(a) any file in Source Code Form that results from an addition to, deletion from, or modification of the contents of Covered Software; or

(b) any new file in Source Code Form that contains any Covered Software.

1.11. "Patent Claims" of a Contributor means any patent claim(s), including without limitation, method, process, and apparatus claims, in any patent Licensable by such Contributor that would be infringed, but for the grant of the License, by the making, using, selling, offering for sale, having made, import, or transfer of either its Contributions or its Contributor Version.

1.12. "Secondary License" means either the GNU General Public License, Version 2.0, the GNU Lesser General Public License, Version 2.1, the GNU Affero General Public License, Version 3.0, or any later versions of those licenses.

1.13. "Source Code Form" means the form of the work preferred for making modifications.

1.14. "You" (or "Your") means an individual or a legal entity exercising rights under this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty percent (50%) of the outstanding shares or beneficial ownership of such entity.

1.11.2 2. License Grants and Conditions

2.1. Grants

Each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license:

(a) under intellectual property rights (other than patent or trademark) Licensable by such Contributor to use, reproduce, make available, modify, display, perform, distribute, and otherwise exploit its Contributions, either on an unmodified basis, with Modifications, or as part of a Larger Work; and

(b) under Patent Claims of such Contributor to make, use, sell, offer for sale, have made, import, and otherwise transfer either its Contributions or its Contributor Version.

2.2. Effective Date

The licenses granted in Section 2.1 with respect to any Contribution become effective for each Contribution on the date the Contributor first distributes such Contribution.

2.3. Limitations on Grant Scope

The licenses granted in this Section 2 are the only rights granted under this License. No additional rights or licenses will be implied from the distribution or licensing of Covered Software under this License. Notwithstanding Section 2.1(b) above, no patent license is granted by a Contributor:

(a) for any code that a Contributor has removed from Covered Software; or

(b) for infringements caused by: (i) Your and any other third party's modifications of Covered Software, or (ii) the combination of its Contributions with other software (except as part of its Contributor Version); or

(c) under Patent Claims infringed by Covered Software in the absence of its Contributions.

This License does not grant any rights in the trademarks, service marks, or logos of any Contributor (except as may be necessary to comply with the notice requirements in Section 3.4).

2.4. Subsequent Licenses

No Contributor makes additional grants as a result of Your choice to distribute the Covered Software under a subsequent version of this License (see Section 10.2) or under the terms of a Secondary License (if permitted under the terms of Section 3.3).

2.5. Representation

Each Contributor represents that the Contributor believes its Contributions are its original creation(s) or it has sufficient rights to grant the rights to its Contributions conveyed by this License.

2.6. Fair Use

This License is not intended to limit any rights You have under applicable copyright doctrines of fair use, fair dealing, or other equivalents.

2.7. Conditions

Sections 3.1, 3.2, 3.3, and 3.4 are conditions of the licenses granted in Section 2.1.

1.11.3 3. Responsibilities

3.1. Distribution of Source Form

All distribution of Covered Software in Source Code Form, including any Modifications that You create or to which You contribute, must be under the terms of this License. You must inform recipients that the Source Code Form of the Covered Software is governed by the terms of this License, and how they can obtain a copy of this License. You may not attempt to alter or restrict the recipients' rights in the Source Code Form.

3.2. Distribution of Executable Form

If You distribute Covered Software in Executable Form then:

(a) such Covered Software must also be made available in Source Code Form, as described in Section 3.1, and You must inform recipients of the Executable Form how they can obtain a copy of such Source Code Form by reasonable means in a timely manner, at a charge no more than the cost of distribution to the recipient; and

(b) You may distribute such Executable Form under the terms of this License, or sublicense it under different terms, provided that the license for the Executable Form does not attempt to limit or alter the recipients' rights in the Source Code Form under this License.

3.3. Distribution of a Larger Work

You may create and distribute a Larger Work under terms of Your choice, provided that You also comply with the requirements of this License for the Covered Software. If the Larger Work is a combination of Covered Software with a work governed by one or more Secondary Licenses, and the Covered Software is not Incompatible With Secondary Licenses, this License permits You to additionally distribute such Covered Software under the terms of such Secondary License(s), so that the recipient of the Larger Work may, at their option, further distribute the Covered Software under the terms of either this License or such Secondary License(s).

3.4. Notices

You may not remove or alter the substance of any license notices (including copyright notices, patent notices, disclaimers of warranty, or limitations of liability) contained within the Source Code Form of the Covered Software, except that You may alter any license notices to the extent required to remedy known factual inaccuracies.

3.5. Application of Additional Terms

You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Software. However, You may do so only on Your own behalf, and not on behalf of any Contributor. You must make it absolutely clear that any such warranty, support, indemnity, or liability obligation is offered by You alone, and You hereby agree to indemnify every Contributor for any liability incurred by such Contributor as a result of warranty, support, indemnity or liability terms You offer. You may include additional disclaimers of warranty and limitations of liability specific to any jurisdiction.

1.11.4 4. Inability to Comply Due to Statute or Regulation

If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Software due to statute, judicial order, or regulation then You must: (a) comply with the terms of this License to the maximum extent possible; and (b) describe the limitations and the code they affect. Such description must be placed in a text file included with all distributions of the Covered Software under this License. Except to the extent prohibited by statute or regulation, such description must be sufficiently detailed for a recipient of ordinary skill to be able to understand it.

1.11.5 5. Termination

5.1. The rights granted under this License will terminate automatically if You fail to comply with any of its terms. However, if You become compliant, then the rights granted under this License from a particular Contributor are reinstated (a) provisionally, unless and until such Contributor explicitly and finally terminates Your grants, and (b) on an ongoing basis, if such Contributor fails to notify You of the non-compliance by some reasonable means prior to 60 days after You have come back into compliance. Moreover, Your grants from a particular Contributor are reinstated on an ongoing basis if such Contributor notifies You of the non-compliance by some reasonable means, this is the first time You have received notice of non-compliance with this License from such Contributor, and You become compliant prior to 30 days after Your receipt of the notice.

5.2. If You initiate litigation against any entity by asserting a patent infringement claim (excluding declaratory judgment actions, counter-claims, and cross-claims) alleging that a Contributor Version directly or indirectly infringes any patent, then the rights granted to You by any and all Contributors for the Covered Software under Section 2.1 of this License shall terminate.

5.3. In the event of termination under Sections 5.1 or 5.2 above, all end user license agreements (excluding distributors and resellers) which have been validly granted by You or Your distributors under this License prior to termination shall survive termination.

1.11.6 6. Disclaimer of Warranty

Covered Software is provided under this License on an “as is” basis, without warranty of any kind, either expressed, implied, or statutory, including, without limitation, warranties that the Covered Software is free of defects, merchantable, fit for a particular purpose or non-infringing. The entire risk as to the quality and performance of the Covered Software is with You. Should any Covered Software prove defective in any respect, You (not any Contributor) assume the cost of any necessary servicing, repair, or correction. This disclaimer of warranty constitutes an essential part of this License. No use of any Covered Software is authorized under this License except under this disclaimer.

1.11.7 7. Limitation of Liability

Under no circumstances and under no legal theory, whether tort (including negligence), contract, or otherwise, shall any Contributor, or anyone who distributes Covered Software as permitted above, be liable to You for any direct, indirect, special, incidental, or consequential damages of any character including, without limitation, damages for lost profits, loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses, even if such party shall have been informed of the possibility of such damages. This limitation of liability shall not apply to liability for death or personal injury resulting from such party’s negligence to the extent applicable law prohibits such limitation. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so this exclusion and limitation may not apply to You.

1.11.12 Exhibit B - “Incompatible With Secondary Licenses” Notice

1.12 Indices and Tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX