Lecture 13

Contents

- Time-dependent variational methods
- Weekly assignments

Time-dependent variational methods

In lecture 11 and lecture 12 we have focused on the spatial discretization of steady (time-independent) differential equations. We will now add time-dependence and thus consider equations like

$$\frac{\partial u}{\partial t} = \mathcal{L}(u) + f,$$
$$\frac{\partial^2 u}{\partial t^2} = \mathcal{L}(u) + f.$$

The solution $u(\boldsymbol{x}, t)$ is now a function of both space $\boldsymbol{x}$ and time t, and naturally, the approximation will be as well. However, we will put all the time dependence in the expansion coefficients

$$u(\boldsymbol{x}, t) \approx u_N(\boldsymbol{x}, t) = \sum_{j=0}^{N} \hat{u}_j(t)\psi_j(\boldsymbol{x}),$$

and the spatial dependence in the basis functions. As such, we still use the same (time-independent) function space $V_N = \text{span}\{\psi_j\}_{j=0}^N$, and the above sum is equivalent to the expansion $u_N(\boldsymbol{x}, t) \in V_N$.

The most common approach for handling time-dependent problems with variational methods is to use a finite difference method for the time-derivative, and then subsequently apply the method of weighted residuals. For example, in order to solve

Skip to main content

$$\frac{\partial u}{\partial t} = \mathcal{L}(u) + f,$$

with the forward Euler method in time, we discretize first as

$$\frac{u^{n+1} - u^n}{\Delta t} = \mathcal{L}(u^n) + f^n, \tag{67}$$

where $u^n(\boldsymbol{x}) = u(\boldsymbol{x}, t_n)$, $f^n(\boldsymbol{x}) = f(\boldsymbol{x}, t_n)$ and $t_n = n\Delta t$.

> **Note**
>
> For simplicity we normally drop the spatial dependence and write only u^n and f^n, even though we mean $u^n(\boldsymbol{x})$ and $f^n(\boldsymbol{x})$.

The solution u^n is known and u^{n+1} is the unknown we are after. As always the Euler method needs to initialize the solution u^0 in order to get started.

A residual can now be defined as

$$\mathcal{R}(u^{n+1}; u^n) = \frac{u^{n+1} - u^n}{\Delta t} - \mathcal{L}(u^n) - f^n,$$

and as always we want this residual to be zero.

> **Note**
>
> The notation $\mathcal{R}(u^{n+1}; u^n)$ is used to indicate that u^{n+1} is the unknown and u^n (right of the semicolon) is known.

We now introduce the approximations to $u^k(\boldsymbol{x}) = u(\boldsymbol{x}, t_k)$, which for given integer k is found in the space V_N

$$u(\boldsymbol{x}, t_k) \approx u_N^k = \sum_{j=0}^{N} \hat{u}_j^k \psi_j(\boldsymbol{x}).$$

Inserted into the residual we get

Skip to main content

$$\mathcal{R}_N = \mathcal{R}(u_N^{n+1}; u_N^n) = \frac{u_N^{n+1} - u_N^n}{\Delta t} - \mathcal{L}(u_N^n) - f^n.$$

The method of weighted residuals is to find $u_N^{n+1} \in V_N$ such that

$$(\mathcal{R}_N, v) = 0, \quad \forall\, v \in W,$$

and for the Galerkin method $W = V_N$.

In order to solve the time-dependent problem we need to solve this Galerkin problem many times. The procedure is

1. Initialize $u_N^0(\boldsymbol{x})$
2. For $n = 0, 1, \ldots, N_t - 1$ find $u_N^{n+1} \in V_N$ such that

$$(\mathcal{R}_N, v) = 0, \quad \forall\, v \in V_N.$$

Here N_t is the number of time steps such that $N_t \Delta t = T$ and $t \in (0, T]$.

> **Note**
>
> By initializing $u_N^0(\boldsymbol{x}) = u^0(\boldsymbol{x})$, we need to project $u^0(\boldsymbol{x})$ to V_N and store the expansion coefficients $\hat{\boldsymbol{u}}^0$. This is a regular function approximation, like in lectures 8 and 9. For the finite difference method, on the other hand, initialization is simply to evaluate (interpolate) the solution on the computational mesh: $(u^0(x_i))_{i=0}^N$. The finite element method can do either project or interpolate, because the FEM is using Lagrange polynomials such that $u_N(x_i) = \hat{u}_i$.

Stability

Just like for the finite difference method we need to be very careful with the time step Δt, in order to obtain a stable solution. A stable solution is one that remains bounded as $t \longrightarrow \infty$. However, there is one major difference from the finite difference or collocation methods. The Galerkin method described above does not involve a mesh size Δx! So the Courant number is not defined. How do we overcome this problem? In order to describe this problem it is helpful to first revisit the stability of the finite difference method.

Skip to main content

Stability of the forward Euler method for finite difference spatial discretizations

We start by considering (67) with finite difference discretizations for both temporal and spatial derivatives:

$$u_j^{n+1} - u_j^n = \Delta t \mathcal{L}(\boldsymbol{u}^n)_j,$$

where, for example, $\mathcal{L}(\boldsymbol{u}^n)_j = (u_{j+1}^n - 2u_j^n + u_{j-1}^n)/\Delta x^2$ if $\mathcal{L}$ represents the second derivative $\partial^2/\partial x^2$. For simplicity we have assumed only one spatial dimension and the mesh function is thus $u_j^n = u(x_j, t_n)$, where $x_j = j\Delta x$ and $t_n = n\Delta t$. The solution vector at a given time step n is given as $\boldsymbol{u}^n = (u_j^n)_{j=0}^N$.

With the finite difference method we assume that the solution can be written as an exponential decay in time and periodic in space (the Fourier wave e^{ikx_j} is periodic in space)

$$u_j^n = (e^{a\Delta t})^n e^{ikx_j} \quad \text{and} \quad \boldsymbol{u}^n = (e^{a\Delta t})^n e^{ik\boldsymbol{x}},$$

where $i = \sqrt{-1}$ and k is a wavenumber. The spatial mesh is $\boldsymbol{x} = (x_j)_{j=0}^N$ and we will write this ansatz as

$$u_j^n = g^n e^{ikx_j} = g^n u_j^0 \quad \text{and} \quad \boldsymbol{u}^n = g^n e^{ik\boldsymbol{x}} = g^n \boldsymbol{u}^0,$$

where $g = e^{a\Delta t}$ is the amplification factor, which is independent of space and time, and only depends on Δt and Δx. We now let the operator $\mathcal{L}$ be the finite difference m'th derivative operator $D^{(m)} = (d_{ij}^{(m)})_{i,j=0}^N$, such that

$$u_j^{n+1} - u_j^n = \Delta t \sum_k d_{jk}^{(m)} u_k^n.$$

For example, if $m = 2$ and we use second order accuracy, then $\sum_k d_{jk}^{(2)} u_k^n = (u_{j+1}^n - 2u_j^n + u_{j-1}^n)/\Delta x^2$. With matrix notation we can write

$$\boldsymbol{u}^{n+1} - \boldsymbol{u}^n = \Delta t D^{(m)} \boldsymbol{u}^n.$$

Now apply the ansatz $\boldsymbol{u}^n = g^n \boldsymbol{u}^0$, such that

Skip to main content

$$g^{n+1}\boldsymbol{u}^0 - g^n\boldsymbol{u}^0 = \Delta t g^n D^{(m)}\boldsymbol{u}^0.$$

Divide by g^n to obtain

$$(g-1)\boldsymbol{u}^0 = \Delta t D^{(m)}\boldsymbol{u}^0. \tag{68}$$

Stability requires that $|g| \leq 1$. However, unlike the decay equation discussed in lectures 1 and 2, it is not always a requirement that $g > 0$, because the solution may be oscillatory.

In order to say something more specific about stability, we need to define the finite difference matrix $D^{(m)}$. So lets assume for now that $m = 2$ and use the derivative matrix with periodic boundary conditions

$$D^{(2)} = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 \\ \vdots & & & \ddots & & & & \dots \\ 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{bmatrix} \tag{69}$$

where $h = \Delta x$ and $D^{(2)} = h^{-2}\tilde{D}^{(2)}$, where $\tilde{D}^{(2)}$ is the matrix in (69) without the h^{-2} scaling. We get

$$(g-1)\boldsymbol{u}^0 = \frac{\Delta t}{\Delta x^2}\tilde{D}^{(2)}\boldsymbol{u}^0,$$

and inserting for $u_j^0 = e^{ikj\Delta x}$ we get

$$(g-1)e^{ikj\Delta x} = \frac{\Delta t}{\Delta x^2}(e^{ik(j+1)\Delta x} - 2e^{ikj\Delta x} + e^{ik(j-1)\Delta x}).$$

Divide by $e^{ikj\Delta x}$ to obtain

$$g - 1 = \frac{\Delta t}{\Delta x^2}(e^{ik\Delta x} - 2 + e^{-ik\Delta x}),$$

and use $e^{ix} + e^{-ix} = 2\cos x$, $\cos(2x) = \cos^2 x - \sin^2 x$ and $1 = \cos^2 x + \sin^2 x$ to obtain

Skip to main content

$$g = 1 - \frac{4\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right).$$

We still want $|g| \leq 1$, so we need

$$|1 - \frac{4\Delta t}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)| \leq 1, \tag{70}$$

and since $\sin^2\left(k\Delta x/2\right) \leq 1$ and positive, we get that $1 - \frac{4\Delta t}{\Delta x^2}$ is always less than 1. However, $1 - \frac{4\Delta t}{\Delta x^2}$ can be smaller than -1, and that gives us the limit

$$1 - \frac{4\Delta t}{\Delta x^2} \geq -1, \tag{71}$$

which is satisfied if $\Delta t/\Delta x^2 \leq 1/2$. This is the absolute stability limit for the forward Euler method for the diffusion equation. Note that it involves both Δt and Δx.

We have now found the absolute stability limit for the diffusion equation using the forward Euler method. However, it was quite complicated to compute since we had to discretize in both space and time. You might ask if there is an easier way, and the answer for once is yes. There is actually a generic approach we can take for any differentiation matrix (any value of m in $D^{(m)}$), and that is to compute its eigenvalues λ, such that

$$D^{(m)}\boldsymbol{u}^0 = \lambda \boldsymbol{u}^0.$$

We can insert this into (68) to obtain

$$(g-1)\boldsymbol{u}^0 = \Delta t \lambda \boldsymbol{u}^0.$$

which indicates that $g - 1 = \Delta t\lambda$. So we can compute the amplification factor g from the eigenvalues of the difference matrix! Since we require that $|g| \leq 1$, we get that

$$|1 + \Delta t\lambda| \leq 1,$$

which is a generic result for the forward Euler method with any difference matrix $D^{(m)}$. Note that λ are the eigenvalues of $D^{(m)}$, which is a matrix that includes Δx. In the case of $D^{(2)}$, we could also find the eigenvalues of $\tilde{D}^{(2)}$, which would be the same as for $D^{(2)}$, but divided by Δx^2. With λ the eigenvalues of $\tilde{D}^{(2)}$, we get the more specific result for the diffusion equation

Skip to main content

$$|1 + \frac{\Delta t}{\Delta x^2}\lambda| \leq 1.$$

We can now compute the eigenvalues of $\tilde{D}^{(2)}$

```
import numpy as np
from scipy import sparse

N = 11
D2 = sparse.diags((1, 1, -2, 1, 1), (-N, -1, 0, 1, N), shape=(N+1, N+1))
Lambda = np.linalg.eig(D2.toarray())[0]
print(f"Minimum eigenvalue = {min(Lambda)}")
```

```
Minimum eigenvalue = -4.0
```

We find that the minimum eigenvalue is -4! And we get the same result using any odd N. Hence we get

$$|1 - 4\frac{\Delta t}{\Delta x^2}| \leq 1.$$

which is the same result as (70) if we set $\sin^2(k\Delta x/2)$ to its maximum value of 1.

Stability of the forward Euler method for variational forms

As described in the start of Stability, the variational methods do not include a mesh size Δx. However, we can use the eigenvalue approach described above without defining a mesh size and we will now use this to find stability limits for variational forms.

First of all we make the same ansatz and assume that the solution u_N^{n+1} can be computed from u_N^n as

$$u_N^{n+1} = gu_N^n \quad \text{and thus} \quad u_N^n = g^n u_N^0, \tag{72}$$

where g is an amplification factor independent of spatial coordinate x. As always a stable method is one where $|g| < 1$.

For the forward Euler method we now consider the discretized Galerkin equation

$$(u_N^{n+1}, v) = (u_N^n, v) + \Delta t\,(\mathcal{L}(u_N^n), v) \tag{73}$$

Skip to main content

and insert for Eq. (72)

$$\left(g^{n+1}u_N^0, v\right) = \left(g^n u_N^0, v\right) + \Delta t \left(\mathcal{L}(g^n u_N^0), v\right)$$

Divide by g^n (g is not dependent on x) to obtain

$$g\left(u_N^0, v\right) = \left(u_N^0, v\right) + \Delta t \left(\mathcal{L}(u_N^0), v\right).$$

We now insert for $u_N^0 = \sum_{j=0}^N \hat{u}_j^0 \psi_j$ and $v = \psi_i$ to obtain

$$g\sum_{j=0}^N (\psi_j, \psi_i)\hat{u}_j^0 = \sum_{j=0}^N (\psi_j, \psi_i)\hat{u}_j^0 + \Delta t \sum_{j=0}^N (\mathcal{L}(\psi_j), \psi_i)\hat{u}_j^0$$

On matrix form this is

$$gA\hat{\boldsymbol{u}}^0 = A\hat{\boldsymbol{u}}^0 + \Delta t M\hat{\boldsymbol{u}}^0$$

where A is the mass matrix and $m_{ij} = (\mathcal{L}(\psi_j), \psi_i)$ and $M = (m_{ij})_{i,j=0}^N$. Multiply from the left by A^{-1} to obtain

$$g\hat{\boldsymbol{u}}^0 = \hat{\boldsymbol{u}}^0 + \Delta t A^{-1} M\hat{\boldsymbol{u}}^0 \tag{74}$$

We can compute the eigenvalues λ of the matrix $H = A^{-1}M$ such that

$$H\hat{\boldsymbol{u}}^0 = \lambda\hat{\boldsymbol{u}}^0,$$

which can be inserted into (74)

$$g\hat{\boldsymbol{u}}^0 = \hat{\boldsymbol{u}}^0 + \lambda\Delta t\hat{\boldsymbol{u}}^0.$$

Since the vector $\hat{\boldsymbol{u}}^0$ is the same in all three terms, we get that

$$g = 1 + \lambda\Delta t$$

and for stability $|g| \leq 1$, we need to have $|1 + \lambda\Delta t| \leq 1$.

So with the Galerkin method we do not need the mesh size Δx in order to compute stability limits. On the other hand, we do need to solve an eigenvalue problem to find the

Skip to main content

eigenvalues λ. In the stability computation, we use the largest eigenvalue. Lets try this out with a complete example, using the diffusion equation.

The diffusion equation

Consider the diffusion equation with Dirichlet boundary conditions and a prescribed initial condition

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \quad x, t \in (0, L) \times (0, T],\ u(0,t) = p(0), u(L,t) = p(L), u(x,0) = \quad (75)$$

In order to solve (75) we choose a global Galerkin method with mapped Legendre polynomials and basis functions $\psi_i(x) = P_i(X) - P_{i+2}(X)$ such that $V_N = \text{span}\{\psi_i\}_{i=0}^N$. We then use Eq. (73) and create a linear algebra problem by inserting for u_N^{n+1}, u_N^n and $v = \psi_i$ and using integrating by parts on the diffusion term:

$$\sum_{j=0}^{N}(\psi_j, \psi_i)\hat{u}_j^{n+1} = \sum_{j=0}^{N}(\psi_j, \psi_i)\hat{u}_j^n - \Delta t \left(\sum_{j=0}^{N}(\psi_j', \psi_i')\hat{u}_j^n - [u_N^n \psi_i]_{x=0}^{x=L}\right).$$

Since we are using basis functions that are all zero on the boundaries, the last term disappears. In order to incorporate the Dirichlet boundary condition we add a boundary term $B(x)$ and solve for $\tilde{u}_N^{n+1} = u_N^{n+1} - B(x)$, where $\tilde{u}_N^{n+1}(0,t) = \tilde{u}_N^{n+1}(L,t) = 0$. Note that we get exactly the same equation for u_N^{n+1} as for $\tilde{u}_N^{n+1}$ since all terms with $B(x)$ will be eliminated when inserting $u_N^{n+1} = \tilde{u}_N^{n+1} + B(x)$ and $u_N^n = \tilde{u}_N^n + B(x)$. But we solve the equation for $\tilde{u}_N^{n+1}$, with the zero boundary conditions, and then to get u_N^{n+1} we simply add $B(x)$.

We use Legendre basis functions with a mapping from $x \in [0, L]$ to $X \in [-1, 1]$ such that

$$x = L(1+X)/2 \quad \text{and} \quad \frac{dx}{dX} = L/2.$$

The stiffness matrix $s_{ij} = (\psi_j', \psi_i')$ was given in Eq. (51), but it needs to be modified for the mapping

Skip to main content

$$\begin{aligned}\int_0^L \psi_j'(x)\psi_i'(x)dx &= \int_{-1}^1 (P_j'(X) - P_{j+2}'(X))\frac{dX}{dx}(P_i'(X) - P_{i+2}'(X))\frac{dX}{dx}\frac{dx}{dX}dX,\\ &= \frac{2}{L}\int_{-1}^1 (P_j'(X) - P_{j+2}'(X))(P_i'(X) - P_{i+2}'(X))dX,\\ &= \frac{2}{L}(P_j' - P_{j+2}', P_i' - P_{i+2}')_{L^2(-1,1)},\\ &= \frac{2}{L}(4i+6)\delta_{ij}.\end{aligned}$$

The mass matrix $a_{ij} = (\psi_j, \psi_i)$ is similarly

$$\begin{aligned}\int_0^L \psi_j(x)\psi_i(x)dx &= \int_{-1}^1 (P_j(X) - P_{j+2}(X))(P_i(X) - P_{i+2}(X))\frac{dx}{dX}dX,\\ &= \frac{L}{2}(P_j - P_{j+2}, P_i - P_{i+2})_{L^2(-1,1)},\\ &= \frac{L}{2}((P_j, P_i) - (P_{j+2}, P_i) - (P_j, P_{i+2}) + (P_{j+2}, P_{i+2})).\end{aligned}$$

The mass matrix is symmetric and we know that $(P_j, P_i) = \|P_i\|^2\delta_{ij}$, where $\|P_i\|^2 = \frac{2}{2i+1}$ is the squared L^2 norm of P_i. Hence with $j = i$ we get

$$(\psi_i, \psi_i) = \frac{L}{2}((P_i, P_i) + (P_{i+2}, P_{i+2})) = \frac{L}{2}(\|P_i\|^2 + \|P_{i+2}\|^2),$$

and with $j = i + 2$ we get

$$(\psi_{i+2}, \psi_i) = -\frac{L}{2}\|P_{i+2}\|^2.$$

The mass matrix is as such

$$a_{ij} = a_{ji} = \frac{L}{2}\begin{cases}\|P_i\|^2 + \|P_{i+2}\|^2, & i = j,\\ -\|P_{i+2}\|^2, & j = i+2,\\ 0, & \text{otherwise.}\end{cases}$$

On matrix form the equation to solve is

$$A\hat{\boldsymbol{u}}^{n+1} = A\hat{\boldsymbol{u}}^n - \Delta t S\hat{\boldsymbol{u}}^n.$$

This equation is solved for $n = 0, 1, \ldots, N_t - 1$, after first initializing $\hat{\boldsymbol{u}}^0$. A solver is

Skip to main content

assignment 2. In the code we assemble the mass and stiffness matrices A and S, and compute the eigenvalues λ of $H = A^{-1}S$. We then use that $|1 - \lambda\Delta t| \leq 1$ (where the minus is because we have used integration by parts) and the fact that here all eigenvalues are positive real numbers, to determine that

$$0 \leq \lambda\Delta t \leq 2.$$

This means that for stability we need to choose a time step $\Delta t \leq 2/\max(\lambda)$.

```
import numpy as np
import sympy as sp
from scipy import sparse
from galerkin import TrialFunction, TestFunction, DirichletLegendre, Dirich
    project, inner

x, t, c, L = sp.symbols('x,t,c,L')

class FE_Diffusion:
    """Class for solving the diffusion equation with

    Parameters
    ----------
    N : int
        Number of basis functions
    L0 : number
        The extent of the domain, which is [0, L0]
    bc : 2-tuple of numbers
        The Dirichlet boundary conditions at the two edges
    u0 : Sympy function of x, t, c and L
        Used for specifying initial condition
    """
    def __init__(self, N, L0=1, u0=sp.cos(sp.pi*x/L)+sp.cos(10*sp.pi*x/L)):
        self.N = N
        self.L = L0
        self.u0 = u0.subs({L: L0})
        self.bc = (self.u0.subs({x: 0, t: 0}).n(), self.u0.subs({x: L0, t: (
        self.uh_np1 = np.zeros(N+1) # expansion coefficients \hat{u}^{n+1}
        self.uh_n = np.zeros(N+1)
        self.V = self.get_function_space()
        self.A, self.S = self.get_mats()
        self.lu = sparse.linalg.splu(self.A.tocsc())

    def get_function_space(self):
        return DirichletLegendre(self.N, domain=(0, self.L), bc=self.bc)

    def get_mats(self):
        """Return mass and stiffness matrices
        """
        N = self.N
        u = TrialFunction(self.V)
        v = TestFunction(self.V)
        a = inner(u, v)
        s = 2/self.L*sparse.diags([4*np.arange(N+1)+6], [0], shape=(N+1, N+1
```

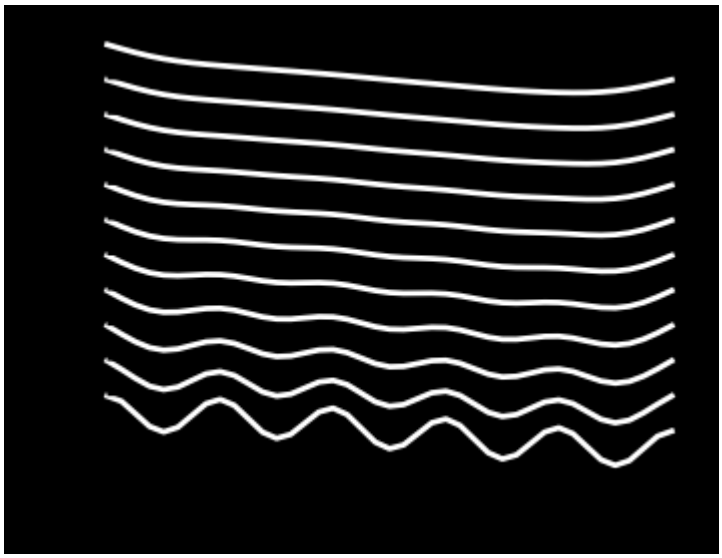
Skip to main content

```
    def get_eigenvalues(self):
        """Return eigenvalues of H=A^{-1}S"""
        return np.linalg.eig(np.linalg.inv(self.A.toarray()) @ self.S.toarra

    def __call__(self, Nt, dt=0.1, save_step=100):
        """Solve diffusion equation

        Parameters
        ----------
        Nt : int
            Number of time steps
        dt : number
            timestep
        save_step : int, optional
            Save solution every save_step time step

        Returns
        -------
        Dictionary with key, values as timestep, array of solution
        The number of items in the dictionary is Nt/save_step, and
        each value is an array of length N+1

        """
        # Initialize
        self.uh_n[:] = project(self.u0, self.V) # unm1 = u(x, 0)
        xj = self.V.mesh()
        plotdata = {0: self.V.eval(self.uh_n, xj)}

        # Solve
        f = np.zeros(self.N+1)
        for n in range(2, Nt+1):
            f[:] = self.A @ self.uh_n - dt*(self.S @ self.uh_n)
            self.uh_np1[:] = self.lu.solve(f)
            self.uh_n[:] = self.uh_np1
            if n % save_step == 0: # save every save_step timestep
                plotdata[n] = self.V.eval(self.uh_np1, xj)
        return plotdata
```

Lets solve the problem using initial condition $u(x,0) = \cos(\pi x/L) + \cos(10\pi x/L)$ and $dt = 2/\max(\lambda)$:

```
from utilities import plot_with_offset
sol = FE_Diffusion(40, L0=2)
dt = 2/max(sol.get_eigenvalues())
data = sol(1000, dt=dt, save_step=100)
plot_with_offset(data, sol.V.mesh(), figsize=(4, 3))
```

Skip to main content



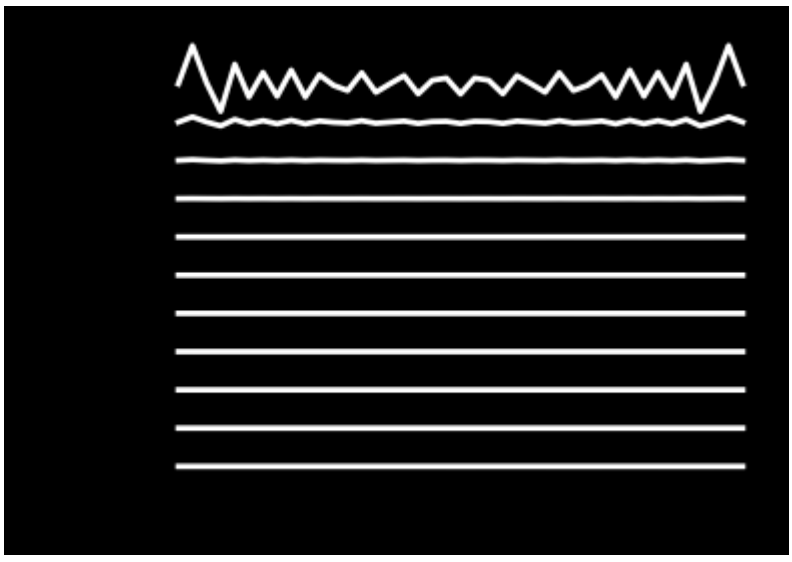
The solution is nice and smooth and the time step is

```
print(f'Maximum time step = {dt}')
```

```
Maximum time step = 2.1980578790345177e-05
```

Lets solve it using a slightly larger dt to see if this really is the largest possible time step:

```
data = sol(1000, dt=1.01*dt, save_step=100)
plot_with_offset(data, sol.V.mesh(), figsize=(4, 3))
```



Obviously, the solver is now unstable and the solution diverges.

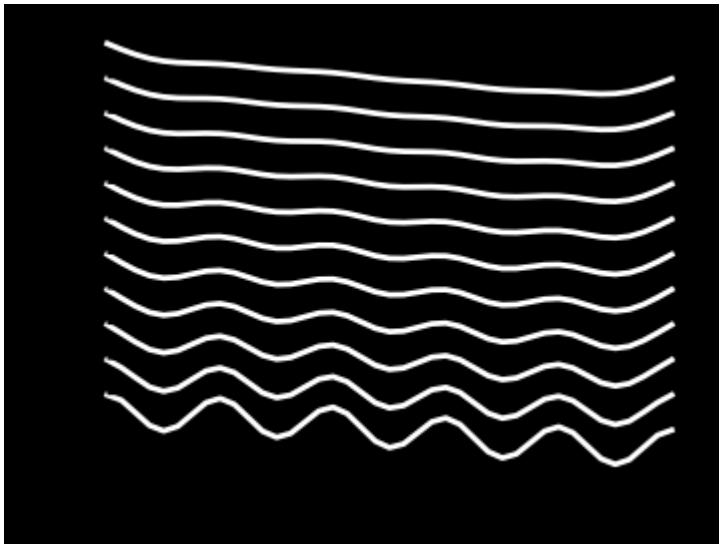
Skip to main content

> **Note**
>
> The diffusion equation is “stiff” and requires a very short time step with the forward Euler method in order to remain stable.

Changing from Legendre to Chebyshev makes the required time step even shorter.

```
class FE_Diffusion_Cheb(FE_Diffusion):

    def get_function_space(self):
        return DirichletChebyshev(self.N, domain=(0, self.L), bc=self.bc)

    def get_mats(self):
        """Return mass and stiffness matrices
        """
        N = self.N
        u = TrialFunction(self.V)
        v = TestFunction(self.V)
        a = inner(u, v)
        #s2 = -inner(u.diff(2), v) # slow
        k = np.arange(N+1, dtype=float)
        def diag(i):
            if i == 0:
                return 2*np.pi*(k+1)*(k+2)
            return 4*np.pi*(k[:-(i)]+1)
        diags = np.arange(0, N+1, 2)
        vals = [diag(i) for i in diags]
        s = 2/self.L * sparse.diags(vals, diags, shape=(N+1, N+1), format='
        return a, s
```

```
sol = FE_Diffusion_Cheb(40, L0=2)
dt = 2/max(sol.get_eigenvalues())
data = sol(1000, dt=dt, save_step=100)
plot_with_offset(data, sol.V.mesh(), figsize=(4, 3))
```



Skip to main content

```
print(f'Maximum time step = {dt}')
```

```
Maximum time step = 1.2332249161314778e-05
```

We can compare the absolute stability limits with the finite difference method, where $\Delta t/\Delta x^2 \leq 0.5$ for stability. Using a uniform mesh with $N = 40$, we get $\Delta x = 2/N = 0.05$ and $\Delta t \leq 0.5 \cdot 0.05^2 = 1.25 \cdot 10^{-3}$. Hence, the Galerkin method with Chebyshev basis functions requires approximately 100 times shorter time steps for stability!

Stiffness matrix for Chebyshev basis

Note that in the implementation we have used $s_{ij} = -(\psi_j'', \psi_i)_\omega$ and

$$(\psi_j'', \psi_i)_\omega = \frac{2}{L}\begin{cases} 2\pi(i+1)(i+2), & i = j, \\ 4\pi(i+1), & j = i+2, i+4, \ldots, N \\ 0, & j < i \text{ or } i+j \text{ odd} \end{cases} \tag{76}$$

because it is quite expensive to compute the matrix using the `inner(u.diff(2), v)` function that loops over all $i, j \in \mathcal{I}_N^2$. Equation (76) can be derived using

$$T_n'' = \sum_{\substack{j=0 \\ j+n \text{ even}}}^{n-2} \frac{1}{c_j} n(n^2 - j^2) T_j,$$

but it is a lot of work and only necessary in order to save some time on computing the stiffness matrix.

Also note that because of the non-constant weight function $\omega(x) = 1/\sqrt{1-x^2}$ it is not common to use integration by parts with the Chebyshev basis.

Use a stencil matrix for the implementation

It is rather messy to work with composite basis functions. A very smart trick is thus to use a **stencil matrix** S, such that (summation on repeated j index)

$$\psi_i = P_i - P_{i+2} = s_{ij} P_j.$$

Skip to main content

This defines $S \in \mathbb{R}^{(N+1)\times(N+3)}$ for the Dirichlet problem as

$$s_{ij} = \begin{cases} 1 & j = i \\ -1 & j = i+2' \end{cases} \quad S = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & \cdots & \\ 0 & 1 & 0 & -1 & 0 & \cdots & \\ 0 & 0 & 1 & 0 & -1 & \cdots & \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \\ 0 & 0 & 0 & \cdots & 1 & 0 & -1 \end{bmatrix}.$$

With $\boldsymbol{\psi} = \{\psi_i\}_{i=0}^{N}$ and $\boldsymbol{P} = \{P_i\}_{i=0}^{N+2}$ (note the longer vector) we get all basis functions in matrix form:

$$\boldsymbol{\psi} = S\boldsymbol{P}.$$

But why is the stencil matrix so smart? Consider the mass matrix

$$\begin{aligned}(\psi_j, \psi_i) &= (P_j - P_{j+2}, P_i - P_{i+2}) \\ &= (P_j, P_i) - (P_j, P_{i+2}) - (P_{j+2}, P_i) + (P_{j+2}, P_{i+2}).\end{aligned}$$

You need to sort out the four (diagonal) matrices on the right. This is not very difficult, since they are all diagonal, but you need to watch out for the indices in order for the mass matrix to turn out correctly. Now use stencil matrix instead (with summation on k and l):

$$\begin{aligned}(\psi_j, \psi_i) &= (s_{jk}P_k, s_{il}P_l) \\ &= s_{il}(P_l, P_k)s_{jk}\end{aligned}.$$

Here you only need the one diagonal matrix $P = ((P_l, P_k))_{k,l=0}^{N+2,N+2}$, where $(P_l, P_k) = \frac{2}{2l+1}\delta_{kl}$. The great advantage only becomes apparent in matrix form though

$$A = ((\psi_j, \psi_i))_{i,j=0}^{N,N} = SPS^T$$

And this works for any stencil matrix, not just the Dirichlet one. That is the main advantage of the stencil matrix approach. For the Neumann basis (see lecture 11)

$$\psi_i(x) = P_i - \frac{i(i+1)}{(i+2)(i+3)}P_{i+2},$$

the stencil matrix is

Skip to main content

$$s_{ij} = \begin{cases} 1 & j = i \\ -i(i+1)/((i+2)(i+3)) & j = i+2' \end{cases} \quad S = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1/6 & 0 \\ 0 & 0 & 1 & 0 & -3/10 \\ \vdots & \vdots & \vdots & \ddots & \ddots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

where $\alpha = N(N+1)/((N+2)(N+3))$. The stiffness matrix can also be computed through the stencil matrix:

$$\begin{aligned}(\psi_j'', \psi_i) &= (s_{jk} P_k'', s_{il} P_l) \\ &= s_{il} (P_l'', P_k) s_{jk}\end{aligned}$$

Again you need only the single matrix using orthogonal polynomials $q_{kl} = (P_l'', P_k) = -(P_l', P_k')$ and the matrix form is again much simpler:

$$M = ((\psi_j'', \psi_i))_{i,j=0}^{N,N} = SQS^T$$

Backward Euler

The forward Euler method requires a very short time step for the diffusion equation. How about the backward Euler method

$$\left(u_N^{n+1}, v\right) = (u_N^n, v) + \Delta t \left(\mathcal{L}(u_N^{n+1}), v\right), \tag{77}$$

where the only difference from forward is that $\left(\mathcal{L}(u_N^{n+1}), v\right)$ is making use of the unknown u_N^{n+1} and not the known u_N^n? The linear algebra problem is now

$$\sum_{j=0}^{N} \left((\psi_j, \psi_i) + \Delta t (\psi_j', \psi_i')\right) \hat{u}_j^{n+1} = \sum_{j=0}^{N} (\psi_j, \psi_i) \hat{u}_j^n.$$

If we continue with the ansatz $u_N^{n+1} = g u_N^n$, and thus $u_N^{n+1} = g^{n+1} u_N^0$, we get

$$\sum_{j=0}^{N} g \left((\psi_j, \psi_i) + \Delta t (\psi_j', \psi_i')\right) \hat{u}_j^0 = \sum_{j=0}^{N} (\psi_j, \psi_i) \hat{u}_j^0,$$

Skip to main content

$$g(A + \Delta t S)\hat{\boldsymbol{u}}^0 = A\hat{\boldsymbol{u}}^0$$

If we multiply from left to right by A^{-1} we get

$$g(I + \Delta t A^{-1} S)\hat{\boldsymbol{u}}^0 = \hat{\boldsymbol{u}}^0.$$

Here we can once again use $H = A^{-1}S$ and $H\hat{\boldsymbol{u}}^0 = \lambda\hat{\boldsymbol{u}}^0$ and thus we find

$$g = \frac{1}{1 + \Delta t \lambda}.$$

Since all time steps and eigenvalues are real numbers larger than 0, we find that g is always less than 1 and the backward Euler method is as such *unconditionally* stable.

The Wave equation

Lets consider the wave equation from lecture 5 in one spatial dimension

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \quad (x, t) \in (0, L) \times (0, T].$$

As you may recall, we solved the wave equation with Dirichlet, Neumann and open boundary conditions. The two time derivatives also required that the solution was initialized at two time-steps, or more mathematically precise, that both $u(x, 0)$ and $\frac{\partial u}{\partial t}(x, 0)$ were given.

We now use finite differences in time such that

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = c^2 \frac{\partial^2 u^n}{\partial x^2}.$$

A numerical residual $\mathcal{R}_N = \mathcal{R}(u_N^{n+1}; u_N^n, u_N^{n-1})$ can now be defined as

$$\mathcal{R}_N = u_N^{n+1} - 2u_N^n + u_N^{n-1} - \alpha^2 (u_N^n)'',$$

where we use $\alpha = c\Delta t$ and $u' = \frac{\partial u}{\partial x}$ for simplicity, since there is only one spatial dimension. If we now initialize u_N^0 and u_N^1, the Galerkin method is to find $u_N^{n+1} \in V_N$ such that

Skip to main content

$$\left(u_N^{n+1} - 2u_N^n + u_N^{n-1} - \alpha^2 (u_N^n)'', v\right) = 0, \quad \forall\, v \in V_N,$$

for $n = 1, \ldots, N_t - 1$. We can also integrate the last term by parts

$$(u_N^{n+1} - 2u_N^n + u_N^{n-1}, v) + \alpha^2 \left(((u_N^n)', v') - \left[(u_N^n)' v\right]_{x=0}^{x=L} \right) = 0, \quad \forall\, v \in V_N.$$

In order to solve the Galerkin problem we need to formulate the linear algebra problem. Hence we insert for $u_N^{n-1}, u_N^n, u_N^{n+1}$ and $v = \psi_i$ to obtain

$$\sum_{j=0}^{N} (\psi_j, \psi_i) \left(\hat{u}_j^{n+1} - 2\hat{u}_j^n + \hat{u}_j^{n-1} \right) + \alpha^2 \left(\sum_{j=0}^{N} (\psi_j', \psi_i') \hat{u}_j^n - \left[(u_N^n)' \psi_i\right]_{x=0}^{x=L} \right) = 0.$$

We next rearrange such that the unknown $\hat{u}_j^{n+1}$ is on the left hand side and the rest of the known terms are on the right hand side

$$\sum_{j=0}^{N} (\psi_j, \psi_i) \hat{u}_j^{n+1} = \sum_{j=0}^{N} (\psi_j, \psi_i)(2\hat{u}_j^n - \hat{u}_j^{n-1}) - \alpha^2 \left(\sum_{j=0}^{N} (\psi_j', \psi_i') \hat{u}_j^n - \left[(u_N^n)' \psi_i\right]_{x=0}^{x=L} \right.$$

With matrix notation, $a_{ij} = (\psi_j, \psi_i)$ and $s_{ij} = (\psi_j', \psi_i')$, we get

$$A\hat{\boldsymbol{u}}^{n+1} = A(2\hat{\boldsymbol{u}}^n - \hat{\boldsymbol{u}}^{n-1}) - \alpha^2 \left(S\hat{\boldsymbol{u}}^n - \boldsymbol{b}^n \right),$$

where $\boldsymbol{b}^n = ((u_N^n)'(L)\psi_i(L) - (u_N^n)'(0)\psi_i(0))_{i=0}^N$. We can write this as

$$A\hat{\boldsymbol{u}}^{n+1} = \boldsymbol{f}^n,$$

where the vector $\boldsymbol{f}^n = A(2\hat{\boldsymbol{u}}^n - \hat{\boldsymbol{u}}^{n-1}) - \alpha^2 \left(S\hat{\boldsymbol{u}}^n - \boldsymbol{b}^n \right)$.

The boundary conditions determine the path forward. Here there are subtile differences between global Galerkin methods and the local finite element method, but the procedures are exactly as detailed in lecture 11 and lecture 12.

For Dirichlet boundary conditions $u(0,t) = a$ and $u(L,t) = b$:

- Global methods make use of basis functions that are all zero at the boundaries $\psi_j(0) = \psi_j(L) = 0$. Hence the vector $\boldsymbol{b}^n$ disappears. Nonzero values for a, b are incorporated using a boundary function, see lecture 11.

Skip to main content

- The finite element method makes use of Lagrange polynomials and $\psi_i(0) = 1$ for $i = 0$ and zero otherwise. Similarly, $\psi_i(L) = 1$ for $i = N$ and zero otherwise. Hence it is only b_0^n and b_N^n that potentially are different from zero. However, since $u(0,t) = a = \hat{u}_0^{n+1}$ and $u(L,t) = b = \hat{u}_N^{n+1}$ we ident A and set $f_0^n = a$ and $f_N^n = b$ (see lecture 12). As such the boundary terms in $\boldsymbol{b}^n$ will never enter the equation system.

For Neumann boundary conditions $u'(0,t) = g_0$ and $u'(L,t) = g_L$:

- Global methods can make use of basis functions that all have $\psi_j'(0) = \psi_j'(L) = 0$, and add a boundary function $B(x)$ that satisfied $B'(0) = g_0$ and $B'(L) = g_L$. However, global methods can also make use of any basis functions (like pure Legendre polynomials) and incorporate the boundary conditions through $\boldsymbol{b}^n$. We get that $b_i^n = g_L\psi_i(L) - g_0\psi_i(0)$. Note that this approach is very easy to use if $g_0 = g_L = 0$, because in that case you only need to neglect the boundary vector $\boldsymbol{b}^n$.
- The finite element method makes use of the weak form and specifies $b_0^n = -g_0$ and $b_N^n = g_L$ since the basis functions are Lagrange polynomials. No more is needed. In the event that $g_0 = g_L = 0$, the boundary vector can simply be neglected.

How about stability? We can make the same ansatz as for the diffusion equation and assume $u_N^{n+1} = g^{n+1}u_N^0$. We get after some trivial manipulations

$$(g^{n+1} - 2g^n + g^{n-1})A\hat{\boldsymbol{u}}^0 = -\alpha^2 g^n S\hat{\boldsymbol{u}}^0,$$

which can be transformed by dividing by g^n and multiplying by A^{-1} from the left

$$(g - 2 + g^{-1})I\hat{\boldsymbol{u}}^0 = -\alpha^2 A^{-1}S\hat{\boldsymbol{u}}^0.$$

Using $H = A^{-1}S$ and $H\hat{\boldsymbol{u}}^0 = \lambda\hat{\boldsymbol{u}}^0$, we get

$$g + g^{-1} = \beta,$$

where $\beta = -\alpha^2\lambda + 2$. This is more difficult to conclude with than for the Euler method, because we have a quadratic equation for g. Nevertheless, we can find

$$g = \frac{\beta \pm \sqrt{\beta^2 - 4}}{2},$$

and in order for $|g| < 1$ (g may be complex) we need $-2 < \beta < 2$ and thus $0 < \alpha^2\lambda < 4$

Skip to main content

$$\Delta t \leq \frac{1}{c}\sqrt{\frac{4}{\max(\lambda)}}.$$

Note

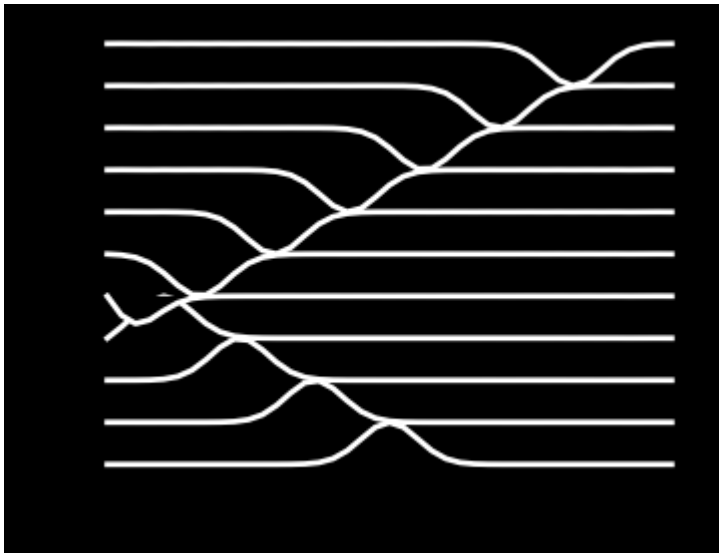
For all $-2 \leq \beta \leq 2$ we get that $|g| = 1$. This makes sense if we consider $g + g^{-1} = \beta$, because if g is a root of the quadratic equation, then g^{-1} must also be a root. Hence, if $g < 1$, then $g^{-1} > 1$, which is unstable. So all the possible β must here have $|g| = 1$. We can also easily compute $|\beta \pm \sqrt{\beta^2 - 4}| = 2$ when the real number $-2 \leq \beta \leq 2$.

Lets implement the wave equation using a global Galerkin method, homogeneous Dirichlet boundary conditions and an initial pulse $u(x,t) = \exp(-200(x - L/2 + ct)^2)$ used for both $t = 0$ and $t = \Delta t$. Since we are using the same matrices as the diffusion problem, we can create a solver by subclassing the `FE_Diffusion` class and thus reuse much code

```
class Wave(FE_Diffusion):
    def __init__(self, N, L0=1, c0=1, u0=sp.cos(sp.pi*x/L)+sp.cos(10*sp.pi*x
        u0 = u0.subs({c: c0})
        FE_Diffusion.__init__(self, N, L0=L0, u0=u0)
        self.c = c0
        self.uh_nm1 = np.zeros_like(self.uh_n)

    def __call__(self, Nt, dt=0.1, save_step=100):
        # Initialize
        self.uh_nm1[:] = project(self.u0.subs(t, 0), self.V) # unm1 = u(x, (
        xj = self.V.mesh()
        plotdata = {0: self.V.eval(self.uh_nm1, xj)}
        self.uh_n[:] = project(self.u0.subs(t, dt), self.V)
        if save_step == 1:
            plotdata[1] = self.V.eval(self.uh_n, xj)

        # Solve
        f = np.zeros(self.N+1)
        alfa = self.c*dt
        for n in range(2, Nt+1):
            f[:] = self.A @ (2*self.uh_n-self.uh_nm1) - alfa**2*(self.S @ s
            self.uh_np1[:] = self.lu.solve(f)
            self.uh_nm1[:] = self.uh_n
            self.uh_n[:] = self.uh_np1
            if n % save_step == 0: # save every save_step timestep
                plotdata[n] = self.V.eval(self.uh_np1, xj)
        return plotdata
```

Skip to main content

```
data = sol(400, dt=dt, save_step=40)
plot_with_offset(data, sol.V.mesh(), figsize=(4, 3))
```



```
print(f"Maximum time step = {dt}")
```

```
Maximum time step = 0.006630321076742087
```

This maximum time step should be compared with the maximum time step of a finite difference method. Remember the Courant number $C = c\Delta t/\Delta x$ and the stability condition $C \leq 1$. For our case, with $c = 1$, this limit is $\Delta t \leq \Delta x$. So with $N = 40$ and $\Delta x = 2/N$, we get $\Delta t \leq 0.05$. Again, the finite difference method can take much longer time steps than the global Galerkin method. But remember, the global Galerkin method is using a much more accurate spatial discretization and can actually get away with using a much lower N.

Weekly assignments

1. Implement the diffusion equation with the backward Euler scheme.
2. Find the absolute stability limit of the diffusion equation discretized with a Crank-Nicolson scheme. Use both a finite difference method $\boldsymbol{u}^{n+1} - \boldsymbol{u}^n = \frac{\Delta t}{2} D^{(2)}(\boldsymbol{u}^{n+1} + \boldsymbol{u}^n)$ and a comparable Galerkin method.
3. Implement the diffusion equation with a Crank-Nicolson scheme and verify the absolute stability limits found in 2.
4. Consider the leap-frog finite difference scheme $\boldsymbol{u}^{n+1} - \boldsymbol{u}^{n-1} = 2\Delta t D^{(2)}\boldsymbol{u}^n$, suggested by Richardson in 1910. Explain why this scheme is unconditionally

Skip to main content

Note

Richardson was one of the great pioneers for using mathematical techniques in weather forecasting, but at the time (1910) he did not know about stability limits for finite difference schemes. The von Neumann stability analysis that made this possible was not discovered until 1947.

Previous
Lecture 12

Next
1. Mandatory assignment due 11/10-2024