Series 3 Warmup

ETH zürich

Numerical methods for PDEs
Last edited: April 7, 2017
Due date: Never at 23:59

Template codes are available on the course's webpage at `https://moodle-app2.let.ethz.ch/course/view.php?id=3089`.

This is a warmup problem. You do **NOT need to hand in** this problem.

Exercise 1 Transient heat equation in 1D

We consider the following one-dimensional, time dependent heat equation:

$$\frac{\partial u}{\partial t}(x,t) - \frac{\partial^2 u}{\partial x^2}(x,t) = 0, \qquad (x,t) \in (0,1) \times (0,T), \tag{1}$$

$$u(0,t) = g_L(t), \quad u(1,t) = g_R(t), \qquad t \in [0,T], \tag{2}$$

$$u(x,0) = u_0(x), \qquad x \in [0,1], \tag{3}$$

where $T > 0$ is the final time, and $g_L, g_R : [0,T] \longrightarrow \mathbb{R}$ are Dirichlet boundary conditions.

We first discretize the above equation with respect to the spatial variable, using *centered finite differences*.

To this aim, we subdivide the interval $[0,1]$ in $N+1$ subintervals of equal length, where N is the number of *interior* grid points $x_1, \dots, x_N$, and $x_0 = 0$, $x_{N+1} = 1$.

The space discretization leads to a *semidiscrete* system of equations associated to (1):

$$\frac{\partial \boldsymbol{u}}{\partial t}(t) + \mathbf{A}\boldsymbol{u}(t) = \boldsymbol{G}(t), \tag{4}$$

where $\mathbf{A} \in \mathbb{R}^{N \times N}$ and $\boldsymbol{u} = \{u_i\}_{i=1}^N$ denotes the approximate values of the solution at the interior grid points. $\boldsymbol{G} : [0,T] \longrightarrow \mathbb{R}^N$ is a source term coming from the boundary conditions.

Hint: $\boldsymbol{G}$ appears from the fact that the discretization for u_1 and u_N includes respectively $u_0 = g_L(t)$ and $u_{N+1} = g_R(t)$

1a)

Denote by h the mesh width, that is $h = \frac{1}{N+1}$. Write down the matrix $\mathbf{A}$ and the vector $\boldsymbol{G}(t)$ explicitly.

Solution: We have

$$\mathbf{A} = \frac{1}{h^2}\begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & -1 & 2 & -1 \\ 0 & \dots & \dots & -1 & 2 \end{pmatrix}, \quad \boldsymbol{G}(t) = \frac{1}{h^2}\begin{pmatrix} g_L(t) \\ 0 \\ \vdots \\ 0 \\ g_R(t) \end{pmatrix}.$$

To fully discretize (1), we still need to apply a time discretization to (4).

1b)

Apply the *forward Euler* scheme to (4), denoting by $\boldsymbol{u}^k = \left\{u_i^k\right\}_{i=1}^N$ the approximate value of the vector $\boldsymbol{u}$ at time k, for $k = 0, \dots, K$, and by $\Delta t = \frac{T}{K}$ the time step. How does the update formula at each time step look like?

Solution: Denote $t_k = k\Delta t$. We obtain

$$\frac{\boldsymbol{u}^{k+1} - \boldsymbol{u}^k}{\Delta t} + \mathbf{A}\boldsymbol{u}^k = \boldsymbol{G}(t_k), \quad k = 0, \dots, K-1,$$

with initial condition $\boldsymbol{u}^0 = \{u_0(x_i)\}_{i=1}^N$. The above system can also be rewritten as

$$\boldsymbol{u}^{k+1} = (\mathbf{I} - \Delta t\mathbf{A})\boldsymbol{u}^k + \Delta t\boldsymbol{G}(t_k) \quad k = 0, \dots, K-1,$$

where $\mathbf{I}$ denotes the identity matrix in $\mathbb{R}^{N\times N}$.

1c)

In the template file `heat_1dfd.cpp`, implement the function

```
void createPoissonMatrix(SparseMatrix& A, int N),
```

where `typedef Eigen::SparseMatrix<double> SparseMatrix`. This function computes the matrix $\mathbf{A}$ from (4). Here the input parameter `N` denotes the number of *interior* grid points. Assume that the size of the input matrix `A` has not been initialized.

Hint: You can copy the routine directly from the solution to an old assignment and do very small modifications to obtain the desired matrix!

Solution: See listing 1 for the code.

Listing 1: Implementation for createPoissonMatrix

```
//! Create the 1D Poisson matrix
//! @param[out] A will contain the Poisson matrix
//! @param[in] N the number of interior points
void createPoissonMatrix(SparseMatrix& A, int N) {
    A.resize(N, N);
    double h=1./(N+1);
    std::vector<Triplet> triplets;
    triplets.reserve(N + 2 * N - 2);
    for (int i = 0; i < N; ++i) {
        triplets.push_back(Triplet(i, i, 2./(h*h)));
        if (i > 0) {
            triplets.push_back(Triplet(i, i - 1, -1./(h*h)));
        }
        if (i < N - 1){
            triplets.push_back(Triplet(i, i + 1, -1./(h*h)));
        }
    }

    A.setFromTriplets(triplets.begin(), triplets.end());
}
```

1d)

In the template file heat_1dfd.cpp, implement the function

```
void explicitEuler(Eigen::MatrixXd  & u, Vector & time, const Vector u0, double dt, double T,
                   int N, const std::function<double(double)>& gL,
                   const std::function<double(double)>& gR)
```

(with `typedef Eigen::VectorXd Vector`). The input and output parameters are specified in the template file.

Solution: See listing 2 for the code.

Listing 2: Implementation for explicitEuler

```
/// Uses the explicit Euler method to compute u from time 0 to time T
///
/// @param[out] u at all time steps up to time T, each column corresponding to a
    ↪ time step (including the initial condition as first column)
```

```
/// @param[out] time the time levels
/// @param[in] u0 the initial data, as column vector
/// @param[in] dt the time step size
/// @param[in] T the final time at which to compute the solution (which we assume
    ↪ to be a multiple of dt)
/// @param[in] N the number of interior grid points
/// @param[in] gL function of time with the Dirichlet condition at left boundary
/// @param[in] gR function of time with the Dirichlet condition at right boundary
///

void explicitEuler(Eigen::MatrixXd & u, Vector & time,
                   const Vector u0, double dt, double T, int N,
                   const std::function<double(double)>& gL,
                   const std::function<double(double)>& gR) {
    const unsigned int nsteps = round(T/dt);
    const double h=1./(N+1);
    u.resize(N,nsteps+1);
    time.resize(nsteps+1);
    /* Initialize A */
    SparseMatrix A;
    createPoissonMatrix(A,N);
    /* Initialize u */
    u.col(0)<<u0;
    time[0]=0.;
    Vector G;
    G.resize(N);
    G.setZero();
    for(unsigned k=0; k<nsteps; k++)
    {
        G[0] = dt*gL(time[k])/(h*h);
        G[N-1] = dt*gR(time[k])/(h*h);

        u.col(k+1)=u.col(k)-dt*A*u.col(k)+G;
        time[k+1]=(k+1)*dt;
    }
}
```

1e)

With the help of the script `sol_movie.m` provided in the handout, observe a movie of the approximate solution to (1) when using the forward Euler scheme. Set the parameters to $T = 0.3$, $\Delta t = 0.0002$, $N = 40$ and $u_0(x) = 1 + \min(2x, 2 - 2x)$ the hat function, $x \in [0, 1]$. Take $g_L(t) = g_R(t) = \exp(-10t)$. What happens to the energy of the system? How does it change if $g_L(t) = 0$, $g_R(t) = 0$? And if $g_L(t) = 1$, $g_R(t) = 0$?

Solution: In the first case, the energy decreases in time until, for $t \to \infty$, the system has no energy anymore. The behavior is analogous for $g_L \equiv g_R \equiv 0$. In the last case, the energy of the system decreases until a linear equilibrium is found.

1f)

We now consider an implicit timestepping. Namely, we derive the Crank-Nicolson scheme. Start with the semidiscrete formulation (4) and integrate over $[t^k, t^{k+1}]$. Use the trapezoidal rule for the integrals involving $\mathbf{A}\boldsymbol{u}$ and $G(t)$, and the approximation $\boldsymbol{u}^k \approx \boldsymbol{u}(t^k)$. Write down the system of equations to be solved at each timestep (this should agree with the Crank-Nicolson scheme stated in the script).

Solution: The semidiscrete formulation is

$$\frac{\partial \boldsymbol{u}}{\partial t}(t) + \mathbf{A}\boldsymbol{u}(t) = G(t).$$

Integration of the first term leads to

$$\int_{t^k}^{t^{k+1}} \frac{\partial \boldsymbol{u}}{\partial t}(t)dt = \boldsymbol{u}(t^{k+1}) - \boldsymbol{u}(t^k) \approx \boldsymbol{u}^{k+1} - \boldsymbol{u}^k,$$

where we have used the approximation $\boldsymbol{u}^k \approx \boldsymbol{u}(t^k)$.

Integration of the second term using the trapezoidal rule leads to

$$\int_{t^k}^{t^{k+1}} \mathbf{A}\boldsymbol{u}(t)dt \approx \frac{\Delta t}{2}\left(\mathbf{A}\boldsymbol{u}(t^{k+1}) + \mathbf{A}\boldsymbol{u}(t^k)\right) \approx \frac{\Delta t}{2}\left(\mathbf{A}\boldsymbol{u}^{k+1} + \mathbf{A}\boldsymbol{u}^k\right).$$

Integration of the right hand side, using the trapezoidal rule, gives

$$\int_{t^k}^{t^{k+1}} \boldsymbol{G}(t)dt \approx \frac{\Delta t}{2}\left(\boldsymbol{G}(t^{k+1}) + \boldsymbol{G}(t^k)\right)$$

Thus, the system of equations reads:

$$\frac{\boldsymbol{u}^{k+1} - \boldsymbol{u}^k}{\Delta t} + \frac{1}{2}\mathbf{A}\boldsymbol{u}^{k+1} + \frac{1}{2}\mathbf{A}\boldsymbol{u}^k = \frac{\Delta t}{2}\left(\boldsymbol{G}(t^{k+1}) + \boldsymbol{G}(t^k)\right), \quad k = 0, \ldots, K-1,$$

with initial condition $\boldsymbol{u}^0 = \{u_0(x_i)\}_{i=1}^N$. The above system can also be rewritten as

$$\left(\mathbf{I} + \frac{\Delta t}{2}\mathbf{A}\right)\boldsymbol{u}^{k+1} = \left(\mathbf{I} - \frac{\Delta t}{2}\mathbf{A}\right)\boldsymbol{u}^k + \frac{\Delta t}{2}\left(\boldsymbol{G}(t^{k+1}) + \boldsymbol{G}(t^k)\right) \quad k = 0, \ldots, K-1,$$

(with $\mathbf{I}$ being the identity matrix in $\mathbb{R}^{N\times N}$).

1g)

In the template file heat_1dfd.cpp, implement the function

```
void CrankNicolson(Eigen::MatrixXd & u, Vector & time, const Vector u0, double dt, double T,
int N)
```

(with `typedef Eigen::VectorXd Vector`). The input and output parameters are specified in the template file.

Hint: In this exercise, you may want to compute $I - M$, where M is a certain sparse matrix and I is the identity. Due to Eigen typecasting, if I is not explicitly defined as a sparse matrix (e.g. it is generated with Eigen::MatrixXd::Identity), $I - M$ will not be a sparse matrix, and sparse solvers will not work. There are several ways to go around this; a simple one is to define I as sparse too with:

```
SparseMatrix I(N,N);
I.setIdentity();
```

Solution: See listing 3 for the code.

Listing 3: Implementation for CrankNicolson

```
/// Uses the Crank-Nicolson method to compute u from time 0 to time T
///
/// @param[out] u at all time steps up to time T, each column corresponding to a
    ↪ time step (including the initial condition as first column)
/// @param[out] time the time levels
/// @param[in] u0 the initial data, as column vector
/// @param[in] dt the time step size
/// @param[in] T the final time at which to compute the solution (which we assume
    ↪  to be a multiple of dt)
/// @param[in] N the number of interior grid points
/// @param[in] gL function of time with the Dirichlet condition at left boundary
/// @param[in] gR function of time with the Dirichlet condition at right boundary
///

void CrankNicolson(Eigen::MatrixXd & u, Vector & time,
                   const Vector u0, double dt, double T, int N,
                   const std::function<double(double)>& gL,
                   const std::function<double(double)>& gR) {
    const unsigned int nsteps = round(T/dt);
    const double h=1./(N+1);
    u.resize(N,nsteps+1);
    time.resize(nsteps+1);
    /* Initialize A */
    SparseMatrix A;
    createPoissonMatrix(A,N);
```

```
    SparseMatrix B(N,N);
    B.setIdentity();
    B+=dt/2.*A;
    /* Initialize u */
    u.col(0)<<u0;
    time[0]=0.;
    /* Initialize solver and compute Cholesky decomposition of B (Note: since dt
        ↪ is constant, the matrix B is the same for all timesteps)*/
    Eigen::SimplicialLDLT<SparseMatrix> solver;
    solver.compute(B);
    Vector G1,G2;
    G1.resize(N);
    G1.setZero();
    G2.resize(N);
    G2.setZero();
    for(unsigned k=0; k<nsteps; k++)
    {
        time[k+1]=(k+1)*dt;
        G1[0] = dt*gL(time[k])/(h*h);
        G1[N-1] = dt*gR(time[k])/(h*h);

        G2[0] = dt*gL(time[k+1])/(h*h);
        G2[N-1] = dt*gR(time[k+1])/(h*h);

        u.col(k+1)=solver.solve(u.col(k)-dt/2*A*u.col(k)
                                +0.5*(G1+G2));

    }
}
```

1h)

With the help of the script `sol_movie.m` provided in the handout, observe a movie of the approximate solution to (1) when using the Crank-Nicolson timestepping scheme. Set the parameters as in subproblem **1e)**. Concerning the energetic behavior of the system, you should observe the same qualitative behavior as in subproblem **1h)**.

1i)

Compute an approximate solution to (1) with both the forward Euler and the Crank-Nicolson schemes. Set the parameters to $T = 0.3$, $N = 20$, $\Delta t = 0.001$ and $u_0(x) = 1 + \min(2x, 2 - 2x)$, $x \in [0, 1]$, $g_L(t) = g_R(t) = \exp(-10t)$. Use now the script `sol_movie.m` provided in the handout to

observe the movie for each of the two solutions. Repeat the experiment with $N = 20$, $\Delta t = 0.01$ and with $N = 5$, $\Delta t = 0.01$. What do you observe?

Solution: With $\Delta t = 0.001$ and $N = 20$, both schemes are stable. With $\Delta t = 0.01$ and $N = 20$, instead, the explicit Euler scheme is unstable (the energy explodes as time passes), while the Crank-Nicholson scheme is stable (the energy is bounded, and more precisely it goes to zero as $t \to \infty$). With $N = 5$ and $\Delta t = 0.01$, both schemes are stable.

1j)

Give an explanation for the observations from subproblem **1i)**. Which condition has to be fulfilled by Δt when using the explicit Euler scheme?

Solution: As we mentioned in the script, the Crank-Nicholson timestepping is unconditionally stable. That is, whatever time step Δt we choose, the energy of the system remains bounded. The explicit schemes (as forward Euler), instead, are conditionally stable, and more precisely, for being stable they need that the time step size fulfills the so-called *CFL* condition. For the heat equation, the CFL condition is:

$$\Delta t \leq \frac{1}{2} h^2,$$

where h denotes the mesh width. Referring to the cases of subproblem **1i)**, the time step must satisfy $\Delta t \leq \frac{1}{2}\left(\frac{1}{N+1}\right)^2$. This leads to $\Delta t \leq 0.0011$ when $N = 20$, and $\Delta t \leq 0.0139$ when $N = 5$. The CFL condition was violated in the second experiment of subproblem **1i)**, and we could observe instability for the forward Euler scheme.