S. Mishra
K. O. Lye
L. Scarabosio

Autumn Term 2015

Computational Methods for Engineering Applications II

ETH Zürich
D-MATH

Homework Problem Sheet 5

This assignment sheet contains some tasks marked as **Core problems**. If you hand them in (see deadline at the end of the problem sheet), these tasks will be corrected. Full mark for the total of the core problems in all assignments will give a 20% bonus on the total points in the final exam. This is really a bonus, which means that at the exam you can still get the highest grade without having the bonus points (of course then you need to score more points at the exam).

The total number of points for the Core problems of this sheet is **3 points**. (The total number of points over all assignments is 20).

Problem 5.1 Transient heat equation in 1D

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

$$\frac{\partial u}{\partial t}(x,t) - \frac{\partial^2 u}{\partial x^2}(x,t) = 0, \qquad (x,t) \in (0,1)\times(0,T), \tag{5.1.1}$$

$$u(0,t) = u(1,t) = 0, \qquad t \in [0,T], \tag{5.1.2}$$

$$u(x,0) = u_0(x), \qquad x \in [0,1], \tag{5.1.3}$$

where $T > 0$ is the final time.

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 (5.1.1):

$$\frac{\partial \boldsymbol{u}}{\partial t}(t) + \mathbf{A}\boldsymbol{u}(t) = 0, \tag{5.1.4}$$

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.

(5.1a) Denote by h the mesh width, that is $h = \frac{1}{N+1}$. Write down the matrix $\mathbf{A}$ explicitly.

Solution: We have

$$\mathbf{A} = \frac{1}{h^2}\begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & -1 & 2 & -1 \\ 0 & \dots & \dots & -1 & 2 \end{pmatrix}.$$

To fully discretize (5.1.1), we still need to apply a time discretization to (5.1.4).

(5.1b) Apply the *forward Euler* scheme to (5.1.4), denoting by $\boldsymbol{u}^k = \{u_i^k\}_{i=1}^N$ 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: We obtain

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

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

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

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

(5.1c) 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 (5.1.4). 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 5.1 for the code.

Listing 5.1: 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());
}
```

(5.1d) (**Core problem**) 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)`

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

Solution: See listing 5.2 for the code.

Listing 5.2: 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
///

void explicitEuler(Eigen::MatrixXd & u, Vector & time,
      const Vector u0, double dt, double T, int N) {
  const unsigned int nsteps = round(T/dt);
  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.;
  for(unsigned k=0; k<nsteps; k++)
    {
     u.col(k+1)=u.col(k)-dt*A*u.col(k);
     time[k+1]=(k+1)*dt;
    }
}
```

(5.1e) With the help of the script `sol_movie.m` provided in the handout, observe a movie of the approximate solution to (5.1.1) when using the forward Euler scheme. Set the parameters to $T = 0.3$, $\Delta t = 0.0002$, $N = 40$ and $u_0(x) = \min(2x, 2 - 2x)$ the hat function, $x \in [0, 1]$. What happens to the energy of the system?

Solution: The energy decreases in time, till, for $t \to \infty$, the system has no energy anymore.

(5.1f) We now consider an implicit timestepping. Namely, we derive the Crank-Nicolson scheme. Start with the semidiscrete formulation (5.1.4) and integrate over $[t^k, t^{k+1}]$. Use the trapezodial rule for the integral involving $\mathbf{A}\boldsymbol{u}$ 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

$$\frac{\partial \boldsymbol{u}}{\partial t}(t) + \mathbf{A}\boldsymbol{u}(t) = 0.$$

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 trapezodial rule leads to

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

Thus, the system of equations reads:

$$\frac{\boldsymbol{u}^{k+1} - \boldsymbol{u}^k}{\Delta t} + \frac{1}{2}\mathbf{A}\boldsymbol{u}^{k+1} + \frac{1}{2}\mathbf{A}\boldsymbol{u}^k = 0, \quad k = 0, \dots, K-1,$$

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

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

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

(5.1g) (**Core problem**) 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.

Solution: See listing 5.3 for the code.

Listing 5.3: Implementation for **CrankNicolson**

```
1 /// Uses the Crank-Nicolson method to compute u from time 0 to time
      T
2 ///
3 /// @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)
4 /// @param[out] time the time levels
5 /// @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
///

void CrankNicolson(Eigen::MatrixXd & u, Vector & time,
      const Vector u0, double dt, double T, int N) {
  const unsigned int nsteps = round(T/dt);
  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);
  for(unsigned k=0; k<nsteps; k++)
    {
      u.col(k+1)=solver.solve(u.col(k)-dt/2*A*u.col(k));
      time[k+1]=(k+1)*dt;
    }
}
```

(5.1h) With the help of the script `sol_movie.m` provided in the handout, observe a movie of the approximate solution to (5.1.1) when using the Crank-Nicolson timestepping scheme. Set the parameters as in subproblem (5.1e). Concerning the energetic behavior of the system, you should observe the same qualitative behavior as in subproblem (5.1h).

(5.1i) Compute an approximate solution to (5.1.1) with both the forward Euler and the Crank-Nicolson schemes. Set the parameters to $T = 0.3$, $N = 20$, $\Delta t = 0.001$ and $u_0(x) = \min(2x, 2-2x)$, $x \in [0,1]$. 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.

(5.1j) Give an explanation for the observations from subproblem (5.1i). 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 (5.1i), 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 (5.1i), and we could observe instability for the forward Euler scheme.

Problem 5.2 Linear transport equation in 1D

Consider the linear transport equation in one dimension with *periodic boundary conditions* and initial data u_0:

$$\frac{\partial u}{\partial t}(x,t) + a\frac{\partial u}{\partial x}(x,t) = 0, \qquad (x,t) \in (0,1) \times \mathbb{R}, \tag{5.2.1}$$

$$u(0,t) = u(1,t), \quad \frac{\partial u}{\partial x}(0,t) = \frac{\partial u}{\partial x}(1,t), \qquad t \in \mathbb{R}, \tag{5.2.2}$$

$$u(x,0) = u_0(x), \qquad x \in [0,1], \tag{5.2.3}$$

with $a \in \mathbb{R}$.

(5.2a) Derive the equation for the characteristics. Assuming $a = \frac{1}{2}$, draw manually or produce a plot of the characteristic lines in the (x,t)-plane.

Solution: The characteristics solve the ODE

$$\begin{aligned} \frac{\mathrm{d}x(t)}{\mathrm{d}t} &= a, x \in (0,1), \\ x(0) &= x_0. \end{aligned}$$

Thus, solving the above problem, we obtain that the characteristic lines are given by $x(t) = x_0 + at$, $t \in \mathbb{R}$. Fig. 5.1 shows the characteristic lines for $a = \frac{1}{2}$. As you can observe, these lines have slope $\frac{1}{a} = 2$ in the (x,t)-plane.

(5.2b) Explain why the solution u to (5.2.1) is constant along the characteristics. Would this still be true if the right-hand side in (5.2.1) is not zero?

Solution: We have that

$$\frac{\mathrm{d}u}{\mathrm{d}t}(x(t),t) = \frac{\partial u}{\partial t} + \frac{\mathrm{d}x(t)}{\mathrm{d}t}\frac{\partial u}{\partial x} = \frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0,$$

where in the second step we have used the equation for the characteristics, and in the last step we have used (5.2.1). The fact that $\frac{\mathrm{d}u}{\mathrm{d}t}(x(t),t) = 0$ means that u is constant when the equation for the characteristic holds, that is u is constant along the characteristics.

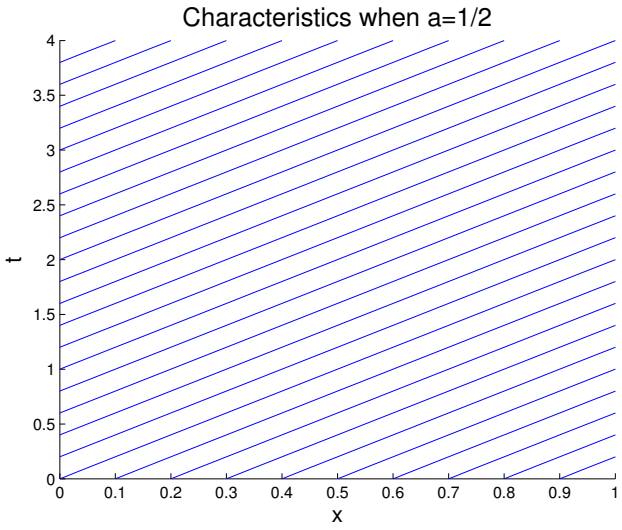

Figure 5.1: Plot for subproblem (5.2a).

If we had a nonzero right-hand side $f = f(x,t)$ in (5.2.1), then we would have that $\frac{\mathrm{d}u}{\mathrm{d}t}(x(t),t) = f(x,t)$. This means that u would not be constant along the characteristics anymore, it would change in a way governed by the *source function* f.

We now want to compute an approximate solution to (5.2.1). For time discretization, we will always use the *forward Euler* scheme, while for space discretization we consider three different finite differences: upwind, central and downwind finite differences.

(5.2c) (**Core problem**) In the template file linear_transport .cpp, implement the function

void UpwindFD(Eigen::MatrixXd & u, Vector & time, **const** Vector u0, **double** dt, **double** T, **int** N, **double** a),

that computes the approximate solution to (5.2.1) using the *forward Euler* scheme for time discretization and *upwind finite differences* for space discretization. The arguments of the function UpwindFD are specified in the template file. Pay attention that this time the input argument N denotes the number of grid points *including the boundary points*.

Solution: See listing 5.4 for the code.

Listing 5.4: Implementation for **UpwindFD**

```
1 /// Uses forward Euler and upwind finite differences to compute u
    from time 0 to time T
2 ///
3 /// @param[out] u solution at all time steps up to time T, each
    column corresponding to a time step (including the initial
    condition as first column)
4 /// @param[out] time the time levels
5 /// @param[in] u0 the initial conditions, as column vector
6 /// @param[in] dt the time step size
7 /// @param[in] T the final time at which to compute the solution
    (which we assume to be a multiple of dt)
8 /// @param[in] N the number of grid points, INCLUDING BOUNDARY
```

```
    POINTS
/// @param[in] a the advection velocity
///

void UpwindFD(Eigen::MatrixXd & u,  Vector & time, const
    Vector u0, double dt, double T, int N, double a)
{
  const unsigned int nsteps = round(T/dt);
  const double dx = 1./(N-1);
  u.resize(N,nsteps+1);
  time.resize(nsteps+1);

  /* Initialize u */
  u.col(0)<<u0;
  time[0]=0;

  for(unsigned k=0; k<nsteps; k++)
    {
      if(a>=0) // take values from the left
     {
       for(unsigned j=1; j<N; j++)
         u(j,k+1) = u(j,k) - dt/dx*a*(u(j,k)-u(j-1,k));
       u(0,k+1) = u(N-1,k+1); // periodic boundary conditions
      }
       else //a<0 => take values from the right
      {
       for(unsigned j=0; j<N-1; j++)
         u(j,k+1) = u(j,k) - dt/dx*a*(u(j+1,k)-u(j,k));
       u(N-1,k+1) = u(0,k+1);
      }
       time[k+1]=(k+1)*dt;
    }
}
```

(5.2d) In the template file linear_transport.cpp, implement the function

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

that computes the approximate solution to (5.2.1) using the *forward Euler* scheme for time discretization and *downwind finite differences* for space discretization. The arguments of the function DownwindFD are specified in the template file.

Solution: See listing 5.5 for the code.

Listing 5.5: Implementation for **DownwindFD**

```
/// Uses forward Euler and downwind finite differences to compute u
    from time 0 to time T
///
```

```cpp
/// @param[out] u solution 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 conditions, 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 grid points, INCLUDING BOUNDARY POINTS
/// @param[in] a the advection velocity
///

void DownwindFD(Eigen::MatrixXd & u, Vector & time, const Vector u0, double dt, double T, int N, double a)
{
  const unsigned int nsteps = round(T/dt);
  const double dx = 1./(N-1);
  u.resize(N,nsteps+1);
  time.resize(nsteps+1);

  /* Initialize u */
  u.col(0)<<u0;
  time[0]=0;

  for(unsigned k=0; k<nsteps; k++)
    {
      if(a<0) // take values from the left
     {
       for(unsigned j=1; j<N; j++)
         u(j,k+1) = u(j,k) - dt/dx*a*(u(j,k)-u(j-1,k));
       u(0,k+1) = u(N-1,k+1); // periodic boundary conditions
     }
      else //a>=0 => take values from the right
     {
       for(unsigned j=0; j<N-1; j++)
         u(j,k+1) = u(j,k) - dt/dx*a*(u(j+1,k)-u(j,k));
       u(N-1,k+1) = u(0,k+1);
     }
      time[k+1]=(k+1)*dt;
    }
}
```

(5.2e) In the template file linear_transport.cpp, implement the function

void CenteredFD(Eigen::MatrixXd & u, Vector & time, **const** Vector u0, **double** dt, **double** T, **int** N, **double** a),

that computes the approximate solution to (5.2.1) using the *forward Euler* scheme for time dis-

cretization and *centered finite differences* for space discretization. The arguments of the function DownwindFD are specified in the template file.

Solution: See listing 5.6 for the code.

Listing 5.6: Implementation for `CenteredFD`

```
/// Uses forward Euler and centered finite differences to compute u from time 0 to time T
///
/// @param[out] u solution 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 conditions, 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 grid points, INCLUDING BOUNDARY POINTS
/// @param[in] a the advection velocity
///

void CenteredFD(Eigen::MatrixXd & u,  Vector & time, const Vector u0, double dt, double T, int N, double a)
{
  const unsigned int nsteps = round(T/dt);
  const double dx = 1./(N-1);
  u.resize(N,nsteps+1);
  time.resize(nsteps+1);

  /* Initialize u */
  u.col(0)<<u0;
  time[0]=0;

  for(unsigned k=0; k<nsteps; k++)
    {
      for(unsigned j=1; j<N; j++)
        u(j,k+1) = u(j,k) - dt/(2.*dx)*a*(u(j+1,k)-u(j-1,k));
      u(0,k+1) = u(0,k) - dt/(2*dx)*a*(u(1,k)-u(N-2,k));
      u(N-1,k+1) = u(0,k+1);
      time[k+1]=(k+1)*dt;
    }
}
```

(5.2f) Run the function main contained in the file linear_transport.cpp. As input parameters, set: $T = 2$, $N = 101$, $\Delta t = 0.02$ and $a = 1$. The initial condition has been set to

$$u_0(x) = \begin{cases} 0 & \text{if } x < 0.25 \text{ or } x > 0.75 \\ 2 & \text{if } 0.25 \leq x \leq 0.75. \end{cases}$$

Use the file `sol_movie.m` to observe movies of the solutions obtained using upwind finite differences, downwind finite differences and centered differences. Repeat the same using now a negative velocity $a = -1$. Answer the following questions:

- The solutions obtained with which finite difference schemes make sense?
- Based on physical considerations, explain the reason why some schemes fail to give a meaningful solution.
- For the schemes that work, what happens to the energy of the system?

Solution: Only the upwind scheme works.

The heuristic reason is that, at each time step, the information about the solution at the previous time step is contained in the upstream direction. To understand this, it is sufficient to observe the characteristic lines. Thus, the downwind and centered finite difference schemes do not work, because they also take information from the downstream direction, which is not physical.

When considering the upwind scheme, the energy remains nearly conserved over time. Indeed, the exact equation (5.2.1) describes energy conservation. Then, since we are using a numerical scheme to solve the equation, the energy is not exactly conserved and since the scheme is stable the energy decreases a bit over time (we can observe some numerical diffusion).

(5.2g) Run the function main contained in the file linear_transport.cpp using $\Delta t = 0.002$, $\Delta t = 0.01$, $\Delta t = 0.011$ and $\Delta t = 0.05$, and the other parameters as in the previous subtask (with $a = 1$). Running the routine `sol_movie.m`, observe the results that you obtain in the four cases when using the upwind finite difference scheme. You can see that in some cases the solution is meaningful, while in the others the energy explodes and the solution is unphysical. Why does this happen? Which condition should the time step Δt fulfill in order to have stability?

Solution: When using $\Delta t = 0.011$ or $\Delta t = 0.05$ the CFL condition is violated. For hyperbolic equations, the CFL condition reads $\frac{\Delta t}{\Delta x} \leq |a|$, where a is the velocity of propagation and Δx the meshwidth. In our experiment $a = 1$ and $\Delta x = \frac{1}{N-1} = 1/100$, which means that the time step must satisfy $\Delta t \leq 0.01$.

Problem 5.3 Upwind finite differences for Burgers' equation in 1D

We consider the so-called Burgers' equation with Dirichlet boundary conditions:

$$\frac{\partial u}{\partial t}(x,t) + \frac{\partial}{\partial x}\left(\frac{u^2(x,t)}{2}\right) = 0, \qquad (x,t) \in (0,1) \times \mathbb{R}, \tag{5.3.1}$$

$$u(0,t) = u_0(0), \qquad t \in \mathbb{R}, \tag{5.3.2}$$

$$u(0,x) = u_0(x), \qquad x \in [0,1], \tag{5.3.3}$$

with some initial data $u_0 = u_0(x) \geq 0$. At the right boundary $x = 1$, we use a non-reflecting boundary condition.

Burgers' equation is an example of a *nonlinear* hyperbolic partial differential equation.

(5.3a) Burgers' equation is a transport equation with nonconstant velocity $a = a(u)$, where u is the solution to (5.3.1). Explain this statement and write explicitly the velocity at which waves travel.

Solution: We have that

$$0 = \frac{\partial u}{\partial t}(x,t) + \frac{\partial}{\partial x}\left(\frac{u^2(x,t)}{2}\right) = \frac{\partial u}{\partial t}(x,t) + u(x,t)\frac{\partial u}{\partial x}(x,t),$$

so Burgers' equation describes the propagation of waves traveling at velocity $a(u) = u$.

(5.3b) Consider an equispaced grid on the interval $[0,1]$. Furthermore, assume that $u_0(x) \geq 0$ for every $x \in [0,1]$. Then the upwind finite difference scheme with forward Euler time stepping reads:

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} + \frac{(u_j^k)^2 - (u_{j-1}^k)^2}{2\Delta x} = 0, \quad j = 1, \ldots, N, k = 0, \ldots, K-1, \tag{5.3.4}$$

(for some $K \in \mathbb{N}$) where Δt denotes the time step, Δx denotes the mesh width, and u_j^k is the approximate value of the solution at time $k\Delta t$ at the point $j\Delta x$, $j = 0, \ldots, N$, $k = 0, \ldots, K$.

Complete the template file burgers_upwind.cpp implementing the function

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

that computes the approximate solution to (5.3.1) using the scheme given in (5.3.4). The arguments of the function BurgersUpwind are specified in the template file.

Solution: See listing 5.7 for the code.

Listing 5.7: Implementation for **BurgersUpwind**

```
1  /// Uses forward Euler and upwind finite differences to compute the
       solution u to Burgers' equation from time 0 to time T
2  ///
3  /// @param[out] u solution at all time steps up to time T, each
       column corresponding to a time step (including the initial
       condition as first column)
4  /// @param[out] time the time levels
5  /// @param[in] u0 the initial conditions, as column vector
6  /// @param[in] dt the time step size
7  /// @param[in] T the final time at which to compute the solution
8  /// @param[in] N the number of grid points, INCLUDING BOUNDARY
       POINTS
9  ///
10
11 void BurgersUpwind(Eigen::MatrixXd & u, Vector & time, const
       Vector u0, double dt, double T, int N)
12 {
13   const unsigned int nsteps = round(T/dt);
14   const double dx = 1./(N-1);
15   u.resize(N,nsteps+1);
16   time.resize(nsteps+1);
17
18   /* Initialize u */
19   u.col(0)<<u0;
20   time[0]=0;
```

```
21
22    for(unsigned k=0; k<nsteps; k++)
23      {
24        for(unsigned j=1; j<N; j++)
25          u(j,k+1) = u(j,k) -
                dt/dx*(u(j,k)*u(j,k)-u(j-1,k)*u(j-1,k))/2.;
26        /* Set boundary condition */
27        u(0,k+1) = u0(0);
28        time[k+1]=(k+1)*dt;
29      }
30  }
```

(5.3c) Run the function main contained in the template file burgers_upwind.cpp with the following parameters: final time $T = 2$, time step $\Delta t = 0.001$, $N = 101$ gridpoints (including those on the boundary) and initial data

$$u_0(x) = \begin{cases} 1 - 2x & \text{if } x \leq \frac{1}{2} \\ 0 & \text{else,} \end{cases} \tag{5.3.5}$$

already implemented in the template file as U0_s. Use the routine `sol_movie.m` provided in the handout to see a movie of the solution. Describe the behavior observed.

Solution: We can see that the solution slope of the solution becomes steeper and steeper, until eventually it forms a so-called *shock*. This is possible because in (5.3.1) the velocity of propagation of the wave is not constant. Thus, in this case, the characteristic lines are not parallel to each other (as it is, instead, in the linear transport equation with constant coefficients).

In particular, the characteristic lines are given by

$$x(t) = x_0 + u(x(t), t)t = x_0 + u_0(x_0)t, \quad x_0 \in [0, 1], t \in \mathbb{R}, \tag{5.3.6}$$

where we have used the fact that the solution u is constant along the characteristics.

The shock forms when the characteristic lines intersect. Once this happens, the concept of characteristics makes no sense anymore (at least once the characteristic has hit the shock).

After the shock has formed, it travels and leaves the domain at the right boundary.

The characteristic lines when the initial condition is (5.3.5) are shown in the left plot of Fig. 5.2.

(5.3d) Repeat the experiment of subproblem (5.3c), now with the initial data

$$u_0(x) = \begin{cases} 0 & \text{if } x \leq \frac{1}{2} \\ 1 & \text{else,} \end{cases} \tag{5.3.7}$$

already implemented in the template file as U0_r (all the other parameters as in the previous subproblem). Describe the behavior observed in this case.

Solution: In this case we see a so-called *rarefaction wave*. The characteristic lines are shown in right plot of Fig. 5.2. Note that, although the initial condition (5.3.7) is discontinuous, the solution $u = u(x, t)$ is globally continuous for every $t > 0$.

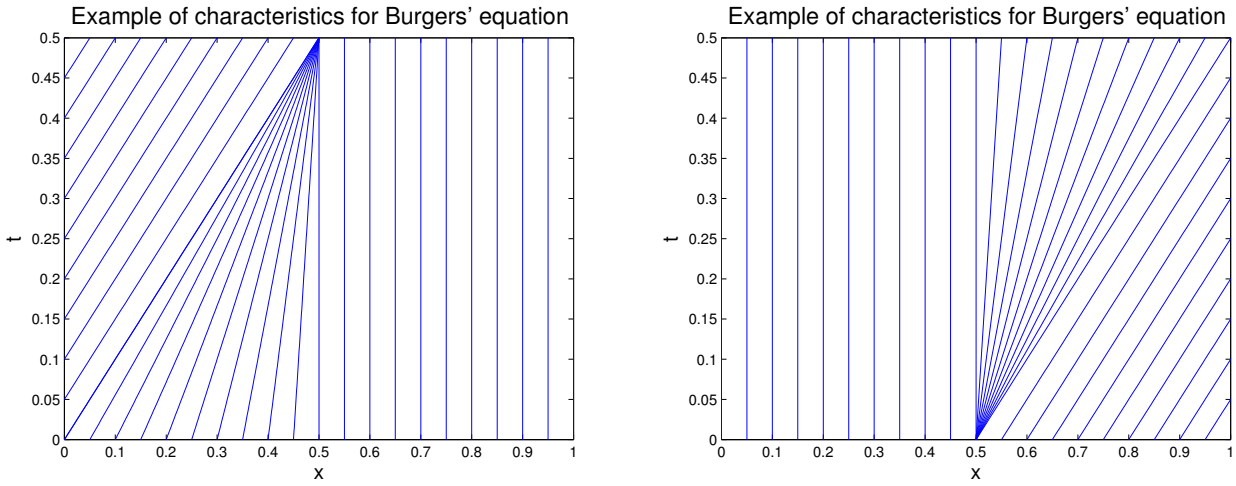

Figure 5.2: Plots for subproblems (5.3c) and (5.3d).

Published on December 9.

To be submitted on December 16.

Last modified on January 7, 2016