Series 1

ETH zürich

Advanced Numerical Methods for CSE
Last edited: December 21, 2021
Due date: December 17, 2021

Template codes are available on the course's webpage at `https://moodle-app2.let.ethz.ch/course/view.php?id=15920`.

IMPORTANT INFORMATION

You can achieve a 0.25 bonus by submitting a solution to Exercise 2 of Series 1 and Exercise 1 of Series 2. Exercises marked as "*Optional*" are beyond the scope of the course, and their content is not part of the oral exam.

Exercise 1 Initial value problem for Burgers' equation

1a)

Find the unique entropy solution $u : \mathbb{R} \times [0, \infty) \to \mathbb{R}$, $(x, t) \mapsto u(x, t)$ of the Burgers equation

$$\partial_t u + \partial_x \left(\frac{u^2}{2} \right) = 0, \tag{1}$$

with the following initial data

$$u_0(x) = \begin{cases} 0, & (x < -1), \\ 1, & (-1 < x < 0), \\ 1 - x, & (0 < x < 1), \\ 0, & (x > 1). \end{cases} \tag{2}$$

Solution: Note that the initial data is regular (Lipschitz), except at $x = -1$, where we have a Riemann problem with $u_L = 0$, $u_R = 1$. Since $f'(u) = u$, we find $f'(u_L) < f'(u_R)$ and thus we have a rarefaction wave emanating from $x_0 = -1$. The shape of the rarefaction wave is given by

$$U(\xi) = (f')^{-1}(\xi) = \xi, \qquad (\xi = (x - x_0)/t).$$

In the other parts of the domain, the characteristic velocities $v = f'(u_0(x))$ are

- $v = 0$, $x < -1$, (region I)
- $v = 1$, $-1 < x < 0$, (region II)
- $v = 1 - x$, $0 < x < 1$, (region III)
- $v = 0$, $x > 1$. (region IV)

Figure 1: Characteristics in each region.

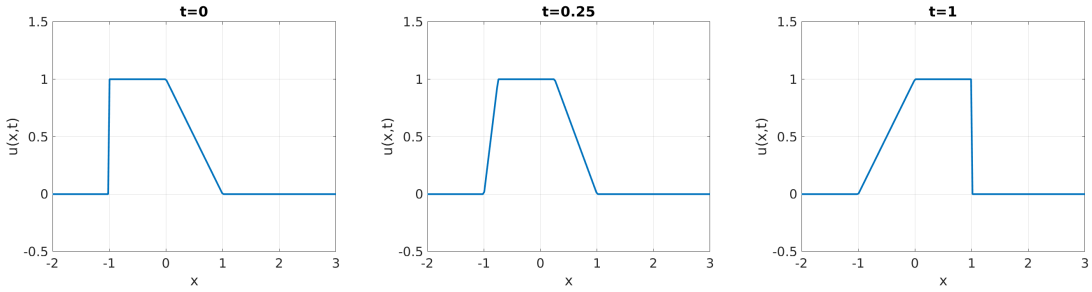
Combining the solution obtained from the method of characteristics and the rarefaction wave, we find the entropy solution

$$u(x,t) = \begin{cases} 0, & x < -1, \\ \frac{x+1}{t}, & -1 < x < t - 1, \\ 1, & t - 1 < x < t, \\ \frac{1-x}{1-t}, & t < x < 1, \\ 0, & x > 1. \end{cases} \tag{3}$$

Note that this solution is only valid as long as the characteristics from each region stay confined within that region. The first escape from region III happens at $t_1 = 1$, at which time

$$u(x,t_1) = \begin{cases} 0, & x < -1, \\ x + 1, & -1 < x < 0, \\ 1, & 0 < x < 1, \\ 0, & x > 1. \end{cases}$$

Note that a discontinuity has formed at $x = 1$ at this time.



(a) $t = 0$ **(b)** $t = 0.25$ **(c)** $t = 1$

Figure 2: Entropy solution $u(x,t)$ for $t \in [0,1]$.

To find the solution for later times, we again have to solve an initial value problem. In this case, we need to consider the Riemann problem at $x = 1$. We have $u_L = 1$, $u_R = 0$, so that $f'(u_L) = 1 > f'(u_R) = 0$, corresponding to a shock. The shock speed is determined with the Rankine-Hugionot condition

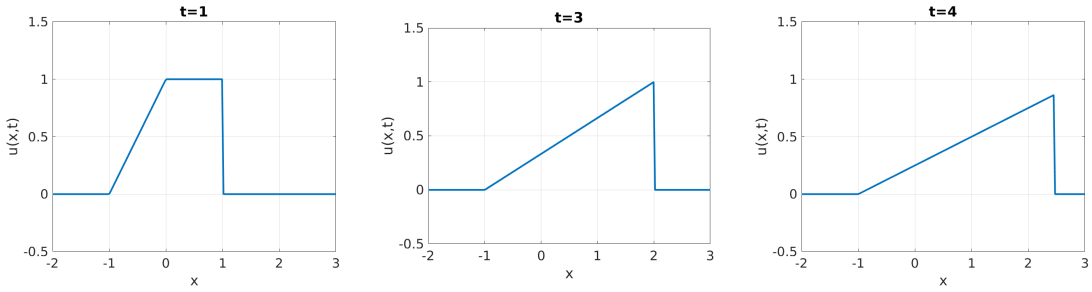
$$s = \frac{f(u_R) - f(u_L)}{u_R - u_L} = \frac{1}{2}.$$

The regular parts of the solution can again be determined from the method of characteristics, leading to

$$u(x,t) = \begin{cases} 0, & x < -1, \\ \frac{x+1}{t}, & -1 < x < t-1, \\ 1, & t-1 < x < (1+t)/2, \\ 0, & x > (1+t)/2, \end{cases} \tag{4}$$

for $t \geq t_1$. Again, this solution is not valid indefinitely in time, because the rarefaction wave (whose right-most part travels along $x_r(t) = t - 1$) will eventually catch up with the shock (which travels along $x_s(t) = (1+t)/2$). The first intersection $x_r(t_2) = x_s(t_2)$ occurs at $t_2 = 3$. At this time, the solution is

$$u(x,t_2) = \begin{cases} 0, & x < -1, \\ \frac{x+1}{3}, & -1 < x < 2, \\ 0, & x > 2. \end{cases}$$



(a) $t = 1$ **(b)** $t = 3$ **(c)** $t = 4$

Figure 3: Entropy solution $u(x,t)$ for $t \in [1,4]$.

To continue the solution further in time, we note that we now have a discontinuity at $x = 2$, **but the left-hand state is not-constant**. We can again check that $f'(u(x-,t_2)) = 1 > f'(u(x+,t_2)) = 0$, so that we must have a shock. However, in contrast to the case with constant states on both sides, the shock velocity is now **no longer constant**. Let $\sigma(t)$ denote the x-position of the shock at time t. We know that $\sigma(t_2) = 2$ at $t_2 = 3$. From the Rankine-Hugoniot condition, we find

$$\begin{aligned}\dot{\sigma}(t) &= \frac{f(u(\sigma(t)^+,t)) - f(u(\sigma(t)^-,t))}{u(\sigma(t)^+,t) - u(\sigma(t)^-,t)} \\ &= \frac{\left(u(\sigma(t)^+,t)\right)^2/2 - \left(u(\sigma(t)^-,t)\right)^2/2}{u(\sigma(t)^+,t) - u(\sigma(t)^-,t)} \\ &= \frac{1}{2}\left(u(\sigma(t)^+,t) + u(\sigma(t)^-,t)\right)\end{aligned}$$

Furthermore, the solution to the left and right of the shock can be determined by the method of characteristics. This clearly yields $u(\sigma(t)+,t) = 0$ for all $t > t_2$. On the other hand, the solution (rarefaction wave) to the left is given by

$$u(\sigma(t)^-,t) = \frac{\sigma(t)+1}{t}.$$

We thus find the following evolution equation for $\sigma(t)$:

$$\dot{\sigma} = \frac{\sigma+1}{2t} \implies \int \frac{d\sigma}{\sigma+1} = \int \frac{dt}{2t} \implies \log\left(\frac{\sigma+1}{\sigma(t_2)+1}\right) = \frac{1}{2}\log\left(\frac{t}{t_2}\right).$$

Thus, substitution of $t_2 = 3$ and $\sigma(t_2) = 2$ yields

$$\sigma(t) = -1 + \sqrt{3t}.$$

Thus, the solution for $t \geq t_2$ is finally found to be given by

$$u(x,t) = \begin{cases} 0, & x < -1, \\ \frac{x+1}{t}, & -1 < x < \sqrt{3t} - 1, \\ 0, & x > \sqrt{3t} - 1, \end{cases} \tag{5}$$

for $t \geq t_2 = 3$.

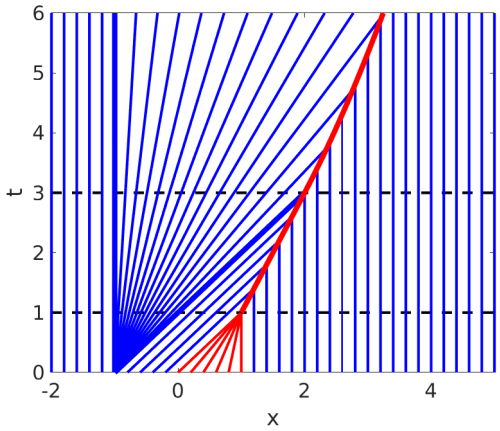


Figure 4: Characteristics for the entropy solution $u(x,t)$.

To conclude: The entropy solution $u(x,t)$ is given by equation (3) for $t \in [0,1]$, by equation (4) for $t \in [1,3]$ and by equation (5) for $t \geq 3$.

Exercise 2 Finite Volume Method for scalar equations in 1D

In this exercise we implement finite volume methods for the one-dimensional, scalar hyperbolic conservation law

$$u_t + f(u)_x = 0, \tag{6}$$

with flux $f(u) = \frac{1}{2}u^2$ on a uniform grid. The cell-centers are denoted by x_i and the interfaces by $x_{i+1/2}$. The semi-discrete formulation of FVM is

$$\frac{d}{dt}U_i + \frac{1}{\Delta x}\left(F_{i+1/2} - F_{i-1/2}\right) = 0 \tag{7}$$

where $F_{i+1/2}$ is the numerical flux through the interface $i+1/2$ and U_i the approximate cell-average of u. The numerical fluxes we consider are two point fluxes, given by

$$F_{i+1/2} = F(U^-_{i+1/2}, U^+_{i+1/2}) \tag{8}$$

with traces $U^{\pm}_{i+1/2}$, i.e. approximations of the value of $u(x_{i+1/2}, t)$ from the left and right of the interface.

Hint: The templates are a simplified version of our research codes. Therefore please read the README.md which comes with the code. It will walk you through the individual parts of the exercise.

Hint: The subproblems are ordered, and it's recommended you solve them in the stated order.

Hint: The file config.json is used to configure the simulation. The path the this file has been hard-coded. Depending on your setup you might need to change edit src/ancse/config.cpp to used the right path for your system. This is likely the case if you're using VS2019. The path should be relative to the executable.

2a)

Implement the following numerical fluxes:

- Rusanov's
- Lax-Friedrichs
- Roe
- Godunov
- Enquist-Osher

Hint: The file include/ancse/numerical_flux.hpp contains an example of how to implement the central flux, called CentralFlux.

Hint: Write tests to check for the correctness of your implementation. You can find the tests in tests/test_numerical_flux.cpp.

Solution: The example of the flux again suggest static polymorphism. Which since the flux is again only very few operations and is called in every cell, this is reasonable.

Rusanov's flux can be implemented as follows.

Listing 1: Implementation of Rusanov's flux.

```
/// Rusanov's flux (or local Lax-Friedrichs).
/** This flux works for any model. */
class Rusanov {
  public:
    explicit Rusanov(const Model &model) : model(model) {}

    double operator()(double uL, double uR) const {
        double a = std::max(model.max_eigenvalue(uL), model.max_eigenvalue(uR));

        auto fL = model.flux(uL);
        auto fR = model.flux(uR);

        return 0.5 * ((fL + fR) - a * (uR - uL));
    }

  private:
    Model model;
};
```

The Lax-Friedrichs flux at first glance does not fit into the suggested pattern, since it needs access to the *current* size of the time-step Δt. This can not be captured by the constructor at the beginning of the simulation. The current simulation time t, the time-step Δt and to a lesser degree the current iteration number, i.e. number of time-steps performed, are often used in unexpected contexts. We solve this problem by storing this information in a small struct and sharing it, i.e. every object that needs access to these pieces of information needs to hold a shared pointer to the "simulation time". This is what we call SimulationTime.

The Lax-Friedrichs flux can then be implemented as follows.

Listing 2: Implementation of Lax-Friedrichs flux.

```
/// Lax-Friedrichs numerical flux.
/** This flux works for any model. */
class LaxFriedrichs {
  public:
    // Note: This version is a bit tricky. A numerical flux should be
```

```
    // a      function of the two trace values at the interface, i.e. what we
    // call      ‘uL‘, ‘uR‘. However, it requires ’dt’ and ’dx’. Therefore,
    // these      need to be made available to the flux. This is one of the
    // reasons      why ‘SimulationTime‘.
    LaxFriedrichs(const Grid &grid,
                  const Model &model,
                  std::shared_ptr<SimulationTime> simulation_time)
        : simulation_time(std::move(simulation_time)),
          grid(grid),
          model(model) {}

    double operator()(double uL, double uR) const {
        double dx = grid.dx;
        double dt = simulation_time->dt;

        //// ANCSE_CUT_START_TEMPLATE

        auto fL = model.flux(uL);
        auto fR = model.flux(uR);

        return 0.5 * ((fL + fR) - dx / dt * (uR - uL));
        //// ANCSE_END_TEMPLATE



        //// ANCSE_RETURN_TEMPLATE

    }

  private:
    std::shared_ptr<SimulationTime> simulation_time;
    Grid grid;
    Model model;
};
```

Both Engquist-Osher's flux and Godunov's flux require some pen and paper calculations in order to be implemented efficiently for Burgers' equation. Their implementation is given below.

Listing 3: Implementation of Engquist-Osher's flux.

```
/// The numerical flux by Engquist and Osher.
/** Engquist-Osher requires the some pen&paper computations specific to the
 * convervation law at hand. However for convex fluxes there’s a convenient and
 * efficient formula.
 *
 * This  version is for the convex fluxes.
 */
```

```
class EngquistOsher {
  public:
    explicit EngquistOsher(const Model &model) : model(model) {}

    double operator()(double uL, double uR) const {
        double omega = model.arg_min_flux();
        double f_plus = model.flux(std::max(uL, omega));
        double f_minus = model.flux(std::min(uR, omega));

        return f_plus + f_minus;
    }

  private:
    Model model;
};
```

Listing 4: Implementation of Godunov's flux.

```
/// The numerical flux due to Godunov
/** Godunov's flux requires the some pen&paper computations specific to the
 * convervation law at hand. However for convex fluxes there's a convenient and
 * efficient formula.
 *
 * This version is for the convex fluxes.
 */
class Godunov {
  public:
    explicit Godunov(const Model &model) : model(model) {}

    double operator()(double uL, double uR) const {
        double omega = model.arg_min_flux();

        double f_plus = model.flux(std::max(uL, omega));
        double f_minus = model.flux(std::min(uR, omega));

        return std::max(f_plus, f_minus);
    }

  private:
    Model model;
};
```

2b)

Implement piecewise linear reconstruction of the trace values $U^{\pm}_{i+1/2}$, with the following slope-limiters:

- minmod
- minabs
- superbee
- monotonized central
- van Leer's limiter

Hint: The file include/ancse/reconstruction.hpp contains an example of how to implement piecewise constant reconstruction, called PWConstantReconstruction.

Hint: Implement tests in tests/test_reconstruction.cpp.

Remark:

$$\text{minabs}(\sigma_L, \sigma_R) := \begin{cases} \sigma_L & \text{if } |\sigma_L| < |\sigma_R| \\ \frac{\sigma_L + \sigma_R}{2} & \text{if } |\sigma_R| = |\sigma_L| \\ \sigma_R & \text{if } |\sigma_R| < |\sigma_L| \end{cases}$$

Solution: We start by implementing some of the commonly used functions, e.g. min_mod.

Listing 5: Implementation of general math functions.

```
inline double minmod(double a, double b) {
    return 0.5 * (sign(a) + sign(b)) * std::min(std::abs(a), std::abs(b));
}

//// ANCSE_CUT_START_TEMPLATE
inline double maxmod(double a, double b) {
    return 0.5 * (sign(a) + sign(b)) * std::max(std::abs(a), std::abs(b));
}

inline double minmod(double a, double b, double c) {
    return minmod(a, minmod(b, c));
}

inline double minabs(double a, double b) {
    double tolerance = 1e-10;
    if (std::abs( std::abs(a) - std::abs(b) ) < tolerance)
        return 0.5*(a+b);
    return std::abs(a) < std::abs(b) ? a : b;
```

```
}
//// ANCSE_END_TEMPLATE
```

Next we chose to implement each slope-limiter as a separate function object. Since we do not know if some slope-limiter at some point might need to have some state, we choose against implementing the slope limiters as structs with static methods. Since a typical slope-limiter perform very few operations and is expected to call in a very tight loop we chose static polymorphism.

This the example for min_mod.

Listing 6: Implementation of MinMod.

```
struct MinMod {
    inline double operator()(double sL, double sR) const {
        return minmod(sL, sR);
    }
};
```

The other slope-limiters follow similarly. Take note of van Leer's slope-limiter. It requires a guard against zero devision. This essentially requires the limiter to know as of what size something is considered to be equal to zero *up to round off errors*. This naturally depends on the problem and should be something that can be chosen at runtime. Nevertheless, we assume that everything is of order 1, i.e. large (in absolute value) numbers are approx. 10^3 and very small numbers are 10^{-8}, but mostly everything is between 10^{-4} and 1. It is a first example of why the slope-limiters might need state.

Listing 7: Implementation of the remaining slope-limiters.

```
struct MinAbs {
    inline double operator()(double sL, double sR) const {
        return minabs(sL, sR);
    }
};

struct SuperBee {
    inline double operator()(double sL, double sR) const {
        double A = minmod(2.0 * sL, sR);
        double B = minmod(sL, 2.0 * sR);

        return maxmod(A, B);
    }
};

struct VanLeer {
    inline double operator()(double sL, double sR) const {
        double r = sL / (sR + eps);
        return (r + std::abs(r)) / (1 + std::abs(r)) * sR;
    }
```

```
  private:
    double eps = 1e-10;
};

struct MonotonizedCentral {
    inline double operator()(double sL, double sR) const {
        return minmod(2.0 * sL, 0.5 * (sL + sR), 2.0 * sR);
    }
};

struct Unlimited {
    inline double operator()(double a, double b) const { return 0.5 * (a + b); }
};
```

Finally we are ready to implement the actual piecewise linear reconstruction of the two trace values. The mathematical formula is (if assuming a equidistant grid):

$$U^{-}_{i+1/2} = U_i + 0.5\ \xi(U_i - U_{i-1}, U_{i+1} - U_i) \tag{9}$$

$$U^{+}_{i+1/2} = U_{i+1} - 0.5\ \xi(U_{i+1} - U_i, U_{i+2} - U_{i+1}) \tag{10}$$

where ξ is a slope-limiter.

We code this up as follows.

Listing 8: Implementation of the remaining slope-limiters.

```
template <class SlopeLimiter>
class PWLinearReconstruction {
  public:
    explicit PWLinearReconstruction(const SlopeLimiter &slope_limiter)
        : slope_limiter(slope_limiter) {}

    std::pair<double, double> operator()(const Eigen::VectorXd &u,
                                         int i) const {
        return (*this)(u[i - 1], u[i], u[i + 1], u[i + 2]);
    }

    std::pair<double, double>
    operator()(double ua, double ub, double uc, double ud) const {


        //double uL = 0. ; //Remove when implemented
        //double uR = 0. ; //Remove when implemented

        //// ANCSE_CUT_START_TEMPLATE
        double sL = ub - ua;
```

```
        double sM = uc - ub;
        double sR = ud - uc;

        double uL = ub + 0.5 * slope_limiter(sL, sM);
        double uR = uc - 0.5 * slope_limiter(sM, sR);

        //// ANCSE_END_TEMPLATE

        return {uL, uR};
    }

  private:
    SlopeLimiter slope_limiter;
};
```

2c)

Complete the loop that applies your fluxes and numerical reconstructions in fvm_rate_of_change.hpp.

Hint: Boundary conditions haven't been yet implemented, but assume they are in place when writing this class. That is, you can trust the ghost cells to contain appropriate values, and you only need to update the central, non-ghost cells here.

Solution: We have fluxes and reconstruction procedures coded up. The loop which computes the rate of change of each cell, which is due to the fluxes, can now be implemented. Note that since computing the rate of change for *all* cells in the grid, is never only a few operations, we can (finally) switch do dynamic polymorphism.

Listing 9: Implementation of the "flux loop".

```
/// Compute the rate of change due to FVM.
/** The semidiscrete approximation of a PDE using FVM is
 * du_i/dt     = - (F_{i+0.5} - F_{i-0.5}) / dx.
 * This  computes the right hand side of the ODE.
 *
 * @tparam NumericalFlux see e.g. `CentralFlux`.
 * @tparam Reconstruction see e.g. `PWConstantReconstruction`.
 */
template <class NumericalFlux, class Reconstruction>
class FVMRateOfChange : public RateOfChange {
  public:
    FVMRateOfChange(const Grid &grid,
                    const NumericalFlux &numerical_flux,
                    const Reconstruction &reconstruction)
        : grid(grid),
          numerical_flux(numerical_flux),
```

```
        reconstruction(reconstruction) {}

    virtual void operator()(Eigen::VectorXd &dudt,
                            const Eigen::VectorXd &u0) const override {

        auto n_cells = grid.n_cells;
        auto n_ghost = grid.n_ghost;

        double dx = grid.dx;
        double fL, fR = 0.0;
        double uL, uR;
        //// ANCSE_CUT_START_TEMPLATE
        //// ANCSE_COMMENT implement the flux loop here.
        for (int i = n_ghost - 1; i < n_cells - n_ghost; ++i) {
            std::tie(uL, uR) = reconstruction(u0, i);

            fL = fR;
            fR = numerical_flux(uL, uR);

            dudt[i] = (fL - fR) / dx;
        }
        //// ANCSE_END_TEMPLATE
    }

  private:
    Grid grid;
    NumericalFlux numerical_flux;
    Reconstruction reconstruction;
};
```

2d)

Implement the second-order strong stability preserving Runge-Kutta (SSP2) scheme.

Hint: The files include/ancse/runge_kutta.hpp and src/ancse/runge_kutta.cpp contains an example of how to implement forward Euler, c.f. ForwardEuler.

Solution: The formula for SSP2 is

$$u^* = u_0 + \Delta t L(u^n) \tag{11}$$
$$u^{**} = u^* + \Delta t L(u^*) \tag{12}$$
$$u^{n+1} = 1/2(u^n + u^{**}). \tag{13}$$

In the implementation of the time-integrator, the class RateOfChange defines the interface for objects

that implement L. Different choices of FVM result in different operators L. However, in our case these have all been implemented in the template class FVMRateOfChange.

The implementation is

Listing 10: Definition of SSP2.

```
SSP2::SSP2(std::shared_ptr<RateOfChange> rate_of_change,
           std::shared_ptr<BoundaryCondition> boundary_condition,
           int n_cells)
    : super(std::move(rate_of_change), std::move(boundary_condition)),
      u_star(n_cells),
      dudt(n_cells) {}

void SSP2::
operator()(Eigen::VectorXd &u1, const Eigen::VectorXd &u0, double dt) const {

    // You can reduce memory consumption by using ‘u1‘ as the temporary
    // buffer ‘u_star‘.

    //// ANCSE_CUT_START_TEMPLATE

    (*rate_of_change)(dudt, u0);
    u_star = u0 + dt * dudt;
    (*boundary_condition)(u_star);

    (*rate_of_change)(dudt, u_star);
    u_star = u_star + dt * dudt;

    u1 = 0.5 * (u0 + u_star);
    (*boundary_condition)(u1);

    //// ANCSE_END_TEMPLATE
}
```

2e)

Implement periodic boundary conditions.

Hint: The file include/ancse/boundary_condition.hpp implements already Outflow boundary conditions, which can be used as an example.

Solution: Periodic boundary conditions simply set the ghost cells to the values of the cells on the other side of the domain, thus creating the illusion of a periodic domain.

Listing 11: Definition of PeriodicBC.

```
PeriodicBC::PeriodicBC(int n_ghost) : n_ghost(n_ghost) {}

void PeriodicBC::operator()(Eigen::VectorXd &u) const {
    using index_t = Eigen::Index;
    index_t n_cells = u.size();

    //// ANCSE_CUT_START_TEMPLATE

    for (index_t i = 0; i < n_ghost; ++i) {
        u[i] = u[n_cells - 2 * n_ghost + i];
        u[n_cells - n_ghost + i] = u[n_ghost + i];
    }

    //// ANCSE_END_TEMPLATE
}
```

2f)

Implement the CFL condition in include/ancse/cfl_condition.hpp.

Hint: Please remember to test your implementation.

Solution: The CFL condition is

$$\Delta t \leq C_{CFL} \frac{\Delta x}{a_{max}}, \quad a_{max} = \max_i f'(u_i), \tag{14}$$

with $C_{CFL} < 1$.

Which we implemented as follows. Note that even for this class where superficially there is only one correct answer, we define an abstract base class which we then implement. The reason is that we want to maintain the flexibility this approach provides. On second thought, we realize that there are a number of good reasons why other CFL conditions might play a role, e.g. for linear advection, $\max_i f'(u_i)$ can be computed once per simulation, therefore we may want to implement an optimized version specifically for linear advection. Another reason is when we parallelize the code. With this approach we could hide the dependencies on OpenMP or MPI in separate subclasses.

Listing 12: Definition of StandardCFLCondition.

```
StandardCFLCondition::StandardCFLCondition(const Grid &grid,
                                           const Model &model,
                                           double cfl_number)
    : grid(grid), model(model), cfl_number(cfl_number) {}

double StandardCFLCondition::operator()(const Eigen::VectorXd &u) const {

    auto n_cells = grid.n_cells;
```

```
    auto n_ghost = grid.n_ghost;

    double a_max = 0.0;

    //// ANCSE_CUT_START_TEMPLATE

    for (int i = grid.n_ghost; i < n_cells - n_ghost; ++i) {
        a_max = std::max(a_max, model.max_eigenvalue(u[i]));
    }

    return cfl_number * grid.dx / a_max;

    //// ANCSE_END_TEMPLATE

    //// ANCSE_RETURN_TEMPLATE

}
```

2g)

To enable selecting the different schemes at run time we need to implement factories for each component.

Start by registering your implementation of SSP2, see src/ancse/runge_kutta.cpp. Note, there is already an example of what to do for ForwardEuler .

Next, register the numerical flux and reconstruction. You'll find an example of how to do this in src/ancse/fvm_rate_of_change.cpp.

Solution: Please consult the reference implementation.

2h)

You now have a code with which you can discover the behaviour of a large number of numerical schemes.

Here are some ideas of what you can do:

- Observe how the numerical schemes behave on a smooth test case, e.g.

$$u(x, t = 0) = u_0(x) = \sin(2\pi x) \tag{15}$$

  on a domain $D = [0, 1]$ with periodic boundary conditions.

- The Roe flux is known to produce an entropy violating shock instead of a rarefaction. You can see this behaviour by looking at the following two step functions

$$u_0(x) = \begin{cases} -1 & x < 0.5 \\ 1 & \text{otherwise} \end{cases} \tag{16}$$

and

$$u_0(x) = \begin{cases} 1 & x < 0.5 \\ -1 & \text{otherwise.} \end{cases} \tag{17}$$

Use a domain $D = [0, 1]$ and outflow boundary conditions. Choose the final time such that the waves don't reach the boundary.

- You can investigate which solves are stable in the presence of discontinuities. Compare different slope-limiters. As initial condition you could again pick a step function and use outflow boundary conditions.
- How sharp is the CFL condition?

Hint: Use piecewise linear reconstruction with minmod, Rusanov's flux and SSP2 timestepping on 2048 cells as a reference solution.

Hint: When assessing the performance of a scheme use a moderate number of cells, in the range of 10s to 100s.

2i)

Optional. Since you now have a 1D version of our research codes, you can do a number of other interesting things.

- Implement the third-order strong stability preserving method. You can read up on this in the review paper by Gottlieb, Shu and Tadmor:

  `https://doi.org/10.1137/S003614450036757X`

  Of particular interest is the introduction and Section 4 "Nonlinear SSP Runge–Kutta Methods".

  This task is not very hard and the paper contains some surprising (and depressing) results about strong stability preserving Runge-Kutta schemes.

  One natural question is: why can't we just use standard RK methods such as *the* fourth-order Runge-Kutta method?

- Implement fifth order WENO. You could read this well-written review paper by Shu:

https://doi.org/10.1137/070679065

Keep in mind that you are implementing a finite volume method, not a finite difference method. In particular, pay attention to the difference of WENO *interpolation* and WENO *reconstruction*.

Only the introduction, Section 2.1 "WENO Interpolation" and Section 2.2 "WENO reconstruction" are needed for this task.

This task is interesting because you will learn about the difference of cell-averages and point-values, which only becomes important at third-order. Furthermore, WENO is one of the most powerful reconstruction schemes available; and once you have the formulae it should be straight forward to implement it in this code.

- You could parallelize the code by using either OpenMP or MPI. Both are currently standard ways of parallelizing numerical codes.
- Writing the output to JSON is not scalable. The two real options are HDF5 and NetCDF. You could implement a subclass of SnapshotWriter which would use either one of these libraries to write the output.

  This task requires some substantial coding and debugging, but these libraries are currently the two most popular ways of writing simulation output to a file. Therefore, it's worth learning them.

Exercise 3 Linear transport equation in 2D (Optional)

Until now we have considered the scalar linear advection equation only in 1D, i.e. $u_t + au_x = 0$, $u(x,0) = u_0(x)$. In this exercise, we are going to study the natural generalization to more than one dimension. In other words: given $\Omega \subset \mathbb{R}^2$ and $T > 0$, we want to find $u : \Omega \times [0,T] \to \mathbb{R}$ such that

$$\partial_t u(\mathbf{x},t) + \mathbf{a}(\mathbf{x}) \cdot \nabla_{\mathbf{x}} u(\mathbf{x},t) = 0, \qquad \forall (\mathbf{x},t) \in \Omega \times [0,T] \tag{18}$$

$$u(\mathbf{x},0) = u_0(\mathbf{x}), \qquad \forall \mathbf{x} \in \Omega \tag{19}$$

which can be written as $\partial_t u + a_1 \partial_x u + a_2 \partial_y u = 0$, with $\mathbf{a} = (a_1, a_2) \in C^1(\Omega; \mathbb{R}^2)$ the advecting velocity. We call $\nabla_{\mathbf{x}} u := (\partial_x u, \partial_y u)$, $\mathbf{x} = (x,y)$, and $\cdot$ denotes the Euclidean scalar product.

For this problem, we take $\Omega = [a,b] \times [c,d] \ni \mathbf{0}$, and we study the specific choice of

$$\mathbf{a}(x,y) := (y, -x). \tag{20}$$

This choice corresponds to the **rotation of a solid** around the origin.

3a)

Characteristic curves in this case are still defined as the curves $\boldsymbol{\gamma} = (\gamma_1, \gamma_2) : [0,T] \to \mathbb{R}^2$ such that

$$\frac{d}{dt} \boldsymbol{\gamma}_{\mathbf{x_0}}(t) = \mathbf{a}(\boldsymbol{\gamma}_{\mathbf{x_0}}(t)) \tag{21}$$

$$\boldsymbol{\gamma}_{\mathbf{x_0}}(0) = \mathbf{x_0}. \tag{22}$$

Find the explicit expression of characteristic curves for problem (18) with velocity $\mathbf{a}$ chosen as in (20), and draw a sketch.

Solution: From eqs. (21)-(22), we obtain a coupled system of ODES:

$$\begin{bmatrix} \gamma_1'(t) \\ \gamma_2'(t) \end{bmatrix} = \begin{bmatrix} \gamma_2(t) \\ -\gamma_1(t) \end{bmatrix}$$

One can observe that the solutions will be of the type $\gamma_1(t) = A\cos t + B \sin t$ and $\gamma_2(t) = B\cos(t) - A\sin(t)$, with $A, B \in \mathbb{R}$ such that eq. (22) holds. In other words, characteristic curves are of the form

$$\gamma_{\mathbf{x_0}}(t) = \begin{bmatrix} x_0 \cos t + y_0 \sin t \\ -x_0 \sin t + y_0 \cos t \end{bmatrix} = \underbrace{\begin{bmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{bmatrix}}_{=:R} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

where is a rotation matrix around the origin. Characteristics are clockwise circles centered in $(0,0)$. Fig 5 shows three characteristics as curves in 3D space, where the z coordinate corresponds to time.

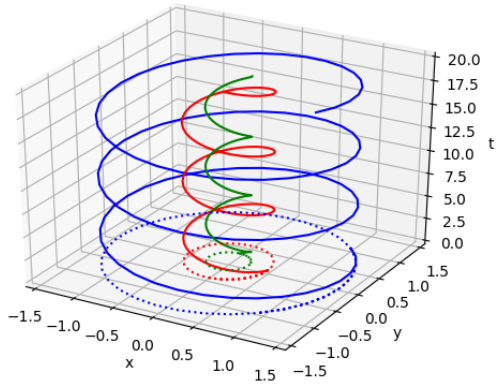


Figure 5: Three characteristic curves for velocity (20). Dashed: projection on (x, y) plane.

3b)

Let $C_{ij} = [x_i, x_{i+1}) \times [y_j, y_{j+1}) \subset \Omega$, and let $\mathbf{a}$ be an arbitrary **divergence free** advection velocity, i.e. $\nabla_{\mathbf{x}} \cdot \mathbf{a} = 0$. Derive from eq. (18) the following equality:

$$\partial_t \frac{1}{|C_{ij}|} \int_{C_{ij}} u(\mathbf{x}, t) d\mathbf{x} + \frac{1}{|C_{ij}|} \int_{x_i}^{x_{i+1}} \left(a_2(x, y_{j+1})u(x, y_{j+1}) - a_2(x, y_j)u(x, y_j)\right) dx \tag{23}$$

$$+ \frac{1}{|C_{ij}|} \int_{y_j}^{y_{j+1}} \left(a_1(x_{i+1}, y)u(x_{i+1}, y) - a_1(x_i, y)u(x_i, y)\right) dy = 0 \tag{24}$$

where $|C_{ij}| = (x_{i+1} - x_i)(y_{j+1} - y_j)$ is the volume of C_{ij}.

Hint: Remember the **divergence theorem** (or Gauss' theorem): if $V \subset \mathbb{R}^d$ is a compact domain with piecewise smooth boundary, and $F \in C^1(U)$ for U an open set containing V, then

$$\int_V (\nabla_{\mathbf{x}} \cdot F) dV = \int_{\partial V} (F \cdot \nu) dS \tag{25}$$

where $\nu(\mathbf{s})$ is the unit **outward-pointing** vector normal to ∂V at point $\mathbf{s}$.

Hint: What do you know about $\nabla_{\mathbf{x}} \cdot (u\mathbf{a})$?

Solution: Let $\mathbf{a} \in C^1(\Omega; \mathbb{R}^2)$ be such that $\nabla_{\mathbf{x}} \cdot \mathbf{a} = 0$. Then we can rewrite eq. (18) as:

$$0 = \partial_t u + \mathbf{a} \cdot \nabla_x u = \partial_t u + \mathbf{a} \cdot \nabla_x u + u \underbrace{\nabla_{\mathbf{x}} \cdot \mathbf{a}}_{=0} = \partial_t u + \nabla_x \cdot (u\mathbf{a})$$

Let us divide the equation above by $|C_{ij}|$ and integrate over C_{ij}. Applying derivation under the integral sign, we obtain:

$$0 = \partial_t \frac{1}{|C_{ij}|} \int_{C_{ij}} u(\mathbf{x},t) d\mathbf{x} + \frac{1}{|C_{ij}|} \int_{C_{ij}} \nabla_x \cdot (u\mathbf{a})(\mathbf{x},t) d\mathbf{x} \tag{26}$$

$$\overset{\text{Gauss thm}}{=} \partial_t \bar{u}_{ij} + \frac{1}{|C_{ij}|} \int_{\partial C_{ij}} (u\mathbf{a})(\mathbf{x},t) \cdot \nu(\mathbf{x}) d\mathbf{x} \tag{27}$$

where $\nu(\mathbf{x})$ is the unit outward-pointing vector orthogonal to ∂C, and $\bar{u}_{ij}$ is the average of u in cell C_{ij}.

Since our cell is a rectangle, it is trivial to divide its boundary into four regions with constant outward-pointing normal vector: the top border $[x_i, x_{i+1}) \times \{y_{j+1}\}$, where the perpendicular vector pointing outwards from C is $\nu \equiv (0,1)$; the bottom border $[x_i, x_{i+1}) \times \{y_j\}$ with $\nu \equiv (0,-1)$, the right border $\{x_{i+1}\} \times [y_j, y_{j+1})$ with $\nu \equiv (1,0)$, and the left border $\{x_i\} \times [y_j, y_{j+1})$ with $\nu(\mathbf{x}) \equiv (-1,0)$.

We can benefit of the simple expression of the outward pointing vectors: e.g. $(u\mathbf{a}) \cdot (-1,0) \equiv -ua_1$; with this, the integral above becomes:

$$0 = \partial_t \bar{u}_{ij} + \frac{1}{|C_{ij}|} \int_{x_i}^{x_{i+1}} \left(a_2(x, y_{j+1}) u(x, y_{j+1}) - a_2(x, y_j) u(x, y_j)\right) dx \tag{28}$$

$$+ \frac{1}{|C_{ij}|} \int_{y_j}^{y_{j+1}} \left(a_1(x_{i+1}, y) u(x_{i+1}, y) - a_1(x_i, y) u(x_i, y)\right) dy \tag{29}$$

□

3c)

We will use (23) to implement a **first-order** finite volume numerical scheme for solid rotation in two dimensions. For that, set a domain $\Omega = [x_{min}, x_{max}) \times [y_{min}, y_{max})$, and fix parameters N_x, N_y. Let $\Delta x = (x_{max} - x_{min})/N_x$, $\Delta y = (y_{max} - y_{min})/N_y$. For $i \in \{0, \ldots, N_x\}$, $j \in \{0, \ldots, N_y\}$, define the **Cartesian grid**

$$x_i := x_{min} + i\Delta x, \qquad y_j := y_{min} + j\Delta y.$$

The grid is depicted in Figure 6.

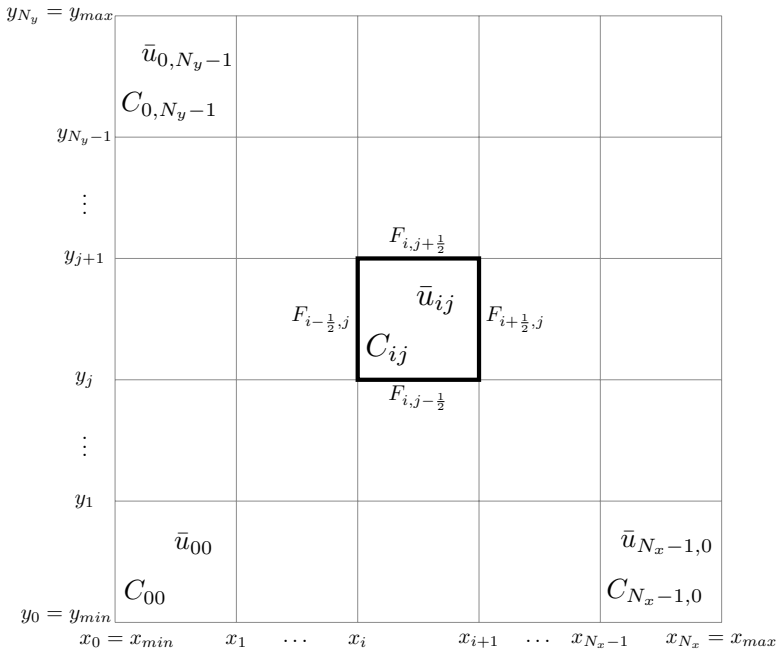


Figure 6: The Cartesian grid.

In file `linear_transport.cpp`, complete function `applyBoundaryConditions`, that fills the first and last row and column of an `(Nx+2) x (Ny+2)` matrix with the appropriate values so that **periodic boundary conditions** are used.

Solution: See listing 1.

Listing 13: Boundary conditions

```
Eigen::Index r = u.rows();
Eigen::Index c = u.cols();

for(int i = 1 ; i < r-1 ; i++) {
    u(i, c-1) = u(i, 1);
    u(i, 0) = u(i, c-2);
}
for( int j = 1 ; j < c-1; j++) {
    u(r-1, j) = u(1, j);
    u(0, j) = u(r-2, j);
}
// this is first order, so we ignore corners
```

3d)

We want to solve problem (18)-(19), with $\Omega = [-1,1]^2$, $T = 2$, and

$$u_0(\mathbf{x}) := \chi_{[-0.3,0.3]^2} = \begin{cases} 1 & \text{if } \mathbf{x} \in [-0.3,0.3]^2 \\ 0 & \text{otherwise} \end{cases}$$

Write an initialization loop for variable `u` in `main.cpp` using provided function `ic`.

Hint: You can use point evaluations $U^0_{i,j} := u_0(x_i + \frac{1}{2}\Delta x,\ y_j + \frac{1}{2}\Delta y)$ rather than compute cell averages. Why is this OK? Would it be an acceptable strategy for a higher-order scheme?

Hint: You may find useful the attached script `plot_init.py`, which plots the initial condition.

Solution: See Listing 14.

Listing 14: Initial condition

```
for (int i = 1 ; i <= Nx ; i++) {
    for(int j = 1 ; j <= Ny ; j++) {
        u(i,j) = ic(xmin + (i-0.5)*dx, ymin + (j-0.5)*dy) ;
    }
}
```

The choice of using point values, without integrating, is due to the accuracy of the midpoint quadrature rule. Point values and cell averages are a second-order approximation of each other; that is,

$$\left| f(x_{i+\frac{1}{2}}) - \frac{1}{\Delta x} \int_{x_i}^{x_{i+1}} f(x)dx \right| \leq C(x_{i+1} - x_i)^2$$

(and analogously in 2D). There is no point in doing somewhat cumbersome calculations to obtain a second-order accurate approximation for the initial condition, since our scheme will be first-order accurate anyway.

3e)

To conclude the implementation, we need to code a discretization of (23)-(24), which depends on several non-trivial integrals at the boundary. For that, use the following (at least) **first-order** accurate estimates:

- $\int_a^b f(x)dx \approx (b-a) f\left(\frac{a+b}{2}\right)$ (midpoint quadrature rule)
- For all $(x,y) \in C_{ij}$, $u(x,y) \approx \bar{u}_{ij}$ (follows from 2D midpoint quadrature rule)
- Remember to apply an **upwind criterion** to choose an approximation of u at cell interfaces!

Using the above approximations, find numerical fluxes $F_{i,j+\frac{1}{2}}$, $F_{i,j-\frac{1}{2}}$, $F_{i+\frac{1}{2},j}$, $F_{i-\frac{1}{2},j}$ where e.g.

$$F_{i,j+\frac{1}{2}} \approx \int_{x_i}^{x_{i+1}} a_2(x, y_{j+1}) u(x, y_{j+1}) dx,$$

and use them to implement function `updateUpwind`, which takes the value of the solution at time-step t^n (as `u_old`) and updates `u` with the values at time t^{n+1}.

Finish your implementation by calling your function in the main loop of the program.

Remark: as a CFL condition, we can use the following (sub-optimal) generalization [1] of 1D CFL:

$$\left| \frac{\Delta t}{\Delta x} \max_{\mathbf{x} \in \Omega} a_1(\mathbf{x}) \right| + \left| \frac{\Delta t}{\Delta y} \max_{\mathbf{x} \in \Omega} a_2(\mathbf{x}) \right| \leq 1$$

Remark: You can modify the parameters of the simulation in file `config.json`. What happens with the result of the simulation as you increase the number of points? How does the runtime of the program scale?

Remark: Note that the path to `config.json` has been hard-coded. You might need to adjust this for you setup, e.g. if you're using VS2019.

Remark: You can generate an animation of your solution with the provided Python script `sol_movie.py`.

Solution:

We start with

$$F^N := \int_{x_i}^{x_{i+1}} a_2(x, y_{j+1}) u(x, y_{j+1}) dx,$$

where N denotes north (i.e. the top border of the cell). A straightforward application of the midpoint quadrature rule in 1 and 2 dimensions gives the following sequence of (at least) first-order approximations:

$$F^N \overset{\text{midpoint rule}}{\approx} a_2(x_{i+\frac{1}{2}}, y_{j+1}) u(x_{i+\frac{1}{2}}, y_{j+1}) \Delta x \tag{30}$$

$$\overset{u(x,y)|_{C_{ij}} \approx \bar{u}_{ij}}{\approx} a_2(x_{i+\frac{1}{2}}, y_{j+1}) \bar{u}_{i,k} \Delta x, \tag{31}$$

where k can be chosen to be j or $j+1$, since point (x, y_{j+1}) lies on the boundary between $C_{i,j}$ and $C_{i,j+1}$. As we learned from the 1D case, we should use some stable numerical flux for the evaluation; as suggested, we use the upwind flux, so we obtain:

$$F^N \approx \Delta x a_2(x_{i+\frac{1}{2}}, y_{j+1})^+ \bar{u}_{i,j} + \Delta x a_2(x_{i+\frac{1}{2}}, y_{j+1})^- \bar{u}_{i,j+1} =: F_{i,j+\frac{1}{2}}.$$

[1] A derivation of this expression, together with a less restrictive alternative, can be found e.g. in R. Leveque's *Finite Volume Methods for Hyperbolic Problems*, chapter 20.

With this, and using forward Euler in time for simplicity, we can finish our implementation; see listing 3.

Listing 15: One time-step of the algorithm

```
for (int i = 1 ; i < u_old.cols()-1 ; i++) {
    for (int j = 1 ; j < u_old.rows()-1 ; j++) {
        auto x_ctr = xmin + (i-0.5)*dx;
        auto y_ctr = ymin + (j-0.5)*dy;
        auto a_N = a(x_ctr, y_ctr + dy/2);
        auto a_S = a(x_ctr, y_ctr - dy/2);
        auto a_E = a(x_ctr + dx/2, y_ctr);
        auto a_W = a(x_ctr - dx/2, y_ctr);
        u(i,j) = u_old(i,j) - (dt/dy)*(F(u_old(i,j), u_old(i, j+1), a_N(1)) - F
            ↪ (u_old(i,j-1), u_old(i, j), a_S(1)))
                        - (dt/dx)*(F(u_old(i,j), u_old(i+1, j), a_E(0)) - F(
                            ↪ u_old(i-1,j), u_old(i, j), a_W(0))) ;
    }
}
```

The scheme works, but it is **very** diffusive; the square profile quickly turns into a circle as the simulation progresses. We again refer to chapter 20 in Leveque's book for better algorithms for 2D scalar conservation laws.

Regarding **runtime**: observe that the error of this method is $O(\Delta x + \Delta y)$. Increasing N_x without increasing N_y (or viceversa) won't improve the results. Furthermore, note that the maximum Δt that we can choose scales linearly with Δx and Δy.

Putting all those observations together, this means that in order to reduce the error by half, we would need to double both N_x and N_y (i.e. each time-step would require four times as many computations), and as a consequence Δt will halve (i.e. we need twice as many time-steps), for a total of approximately **eight times** more computations required for halving the error.

In other words: if we assume $O(N_x) = O(N_y)$, then runtime scales like $O(N_x^3)$: reducing the error one order of magnitude would require three orders of magnitude more computations! This can be seen clearly in Figure 7.

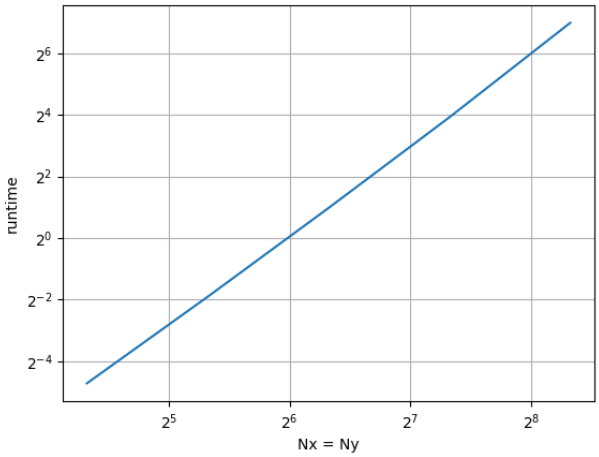


Figure 7: Runtime (in seconds) at increasing resolutions, CFL=0.9. Note cubic scaling.