Template codes are available on the course’s webpage at https://metaphor.ethz.ch/x/2019/hs/401-4671-00L/.

IMPORTANT INFORMATION

In order to take the exam for the course, you must submit a solution to this exercise sheet. After the deadline, a mandatory code review will take place, in which you will individually discuss your solution with one of the assistants. Details will follow.

To take the final exam, a partial solution to the exercise sheet is enough – a complete solution to at least one of the exercises, which doesn’t need to be perfect, suffices. This interview is not an exam; you will not be assigned a numerical grade. The purpose of the interview is to ensure that you’ve either solved the exercise yourself, or solved it in a small group and you are familiar with every aspect of the code.

A bonus of $+0.25$ to your exam grade will be awarded for handing in and defending either a complete, but not necessarily perfect, solution to both exercises; or a complete, correct solution to one.

Theory questions (2a-2d) will not be discussed, although they will be necessary to solve the exercise.
Exercise 1  Finite volume and discontinuous Galerkin methods for Euler equations in 1D

Euler equations in 1D are written as:

\[ u_t + f(u)_x = 0, \quad (1a) \]
\[ u = \begin{pmatrix} \rho \\ \rho v \\ E \end{pmatrix}, \quad f(u) = \begin{pmatrix} \rho v \\ \rho v^2 + p \\ (E + p)v \end{pmatrix}, \quad (1b) \]

where the density \( \rho \), velocity \( v \) and energy \( E \) are unknown, and the pressure \( p \) is determined by the equation of state:

\[ E = \frac{p}{\gamma - 1} + \frac{1}{2} \rho v^2, \quad \text{for } \gamma = 7/5 = 1.4. \quad (2) \]

Additionally, the speed of sound \( c \) and enthalpy \( H \) are given by:

\[ c = \sqrt{\frac{\gamma p}{\rho}}, \quad H = \frac{E + p}{\rho}. \quad (3) \]

In this exercise, we implement finite volume and discontinuous Galerkin methods for the 1D Euler equations (1) on a domain \( \Omega \) discretized by a uniform grid \( T_h \), with the cell-centers denoted by \( x_i \), the interfaces denoted by \( x_{i+1/2} \) and the cell size \( h \).

**Hint:** Please read the README.md which comes with the code. It will briefly walk you through the individual parts of the exercise.

**Hint:** The subproblems are ordered, and it is recommended that 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 use the right path for your system. This is likely the case if you’re using VS2019. The path should be relative to the executable.

**Hint:** You can also create new config, e.g. filename.json, and run with this configuration as

```
./fvm_euler path_relative_to_exec/filename.json
```

**Hint:** If you run into compilation issues due to library filesystem missing in snapshot_writer.cpp, your compiler is very likely not fully-C++17 complying; updating it should fix the issue.

**Finite volume method**

The semi-discrete formulation of FVM is

\[ \frac{d}{dt} U_i + \frac{1}{\Delta x} (F_{i+1/2} - F_{i-1/2}) = 0 \quad (4) \]
where $\mathbf{U}_i$ is the approximate cell-average of $\mathbf{u}$ in cell $i$ and $\mathbf{F}_{i+1/2}$ is the numerical flux through the interface $i + 1/2$. We consider two-point numerical fluxes given by

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

(5)

where the traces $\mathbf{U}^-_{i+1/2}, \mathbf{U}^+_{i+1/2}$ are approximate values of $\mathbf{u}(x_{i+1/2}, t)$ to the left and to the right of the interface, respectively.

1a)

Templates: include/ancse/model.hpp, src/ancse/model.cpp.

Implement the Euler equations model.

**Hint:** The class Burgers is given as an example for the Burgers’ equation. You can add additional routines to the interfacing class Model, if needed.

**Hint:** Write tests to check the accuracy of the Euler equations model in tests/test_model.cpp

**Solution:** To start with, we implement some routines specific to the Euler model, which are not available through the interface.

Listing 1: Implementation of routines specific to Euler equations.

```cpp
inline std::tuple<double, double, double>
primitive(const Eigen::VectorXd &u_cons) const
{
    double rho = u_cons(0);
    double v = u_cons(1)/u_cons(0);
    double E = u_cons(2);
    double p = pressure(rho, v, E);
    return std::make_tuple (rho, v, p);
}

inline double pressure(double rho, double v, double E) const
{
    return (gamma-1)*(E - 0.5*rho*v*v);
}

inline double sound_speed(double rho, double p) const
{
    return sqrt(gamma*p/rho);
}

inline double enthalpy(double rho, double E, double p) const
{
    return (E+p)/rho;
}
```

3
Now, we implement the routines provided through the interface class.

Listing 2: Implementation of the interface routines of Euler equations.

```cpp
Eigen::VectorXd Euler::flux(const Eigen::VectorXd &u) const
{
    double rho, v, p;
    std::tie(rho, v, p) = primitive(u);
    double E = energy(rho, v, p);

    Eigen::VectorXd f(n_vars);
    f << u(1), rho*v*v + p, (E+p)*v;

    return f;
}

Eigen::VectorXd Euler::eigenvalues(const Eigen::VectorXd &u) const
{
    // ANCSE_CUT_START_TEMPLATE
    Eigen::VectorXd eigvals(n_vars);
    eigvals << v - c, v, v + c;
    return eigvals;

    Eigen::VectorXd eigvecs(n_vars, n_vars);
    double c = sqrt((gamma-1)*(H - 0.5*v*v));

    eigvecs << 1 , 1 , 1 ,
              v - c , v , v + c ,
              H - v*c , 0.5*v*v , H + v*c;

    return eigvecs;
}
```
double rho, v, p;
std::tie (rho, v, p) = primitive(u);
double c = sound_speed(rho, p);

Eigen::VectorXd eigvals(n_vars);
eigvals << v - c, v, v + c;

return eigvals;

Eigen::MatrixXd Euler::eigenvectors(const Eigen::VectorXd &u) const {

double rho, v, p;
std::tie (rho, v, p) = primitive(u);
double E = energy(rho, v, p);
double H = enthalpy(rho, E, p);

return eigenvectors(v, H);
}

double Euler::max_eigenvalue(const Eigen::VectorXd &u) const {
return (eigenvalues(u).cwiseAbs()).maxCoeff();
}

Eigen::VectorXd Euler::cons_to_prim(const Eigen::VectorXd &u_cons) const {

double rho, v, p;
std::tie (rho, v, p) = primitive(u_cons);

Eigen::VectorXd u_prim(n_vars);
return u_prim;
}
Eigen::VectorXd Euler::prim_to_cons(const Eigen::VectorXd &u_prim) const
{
    Eigen::VectorXd u_cons(n_vars);
    u_cons(0) = u_prim(0);
    u_cons(1) = u_prim(0)*u_prim(1);
    u_cons(2) = u_prim(2)/(gamma-1) + 0.5*u_prim(0)*u_prim(1)*u_prim(1);
    return u_cons;
}

Eigen::VectorXd Euler::roe_avg(const Eigen::VectorXd &uL, const Eigen::VectorXd &uR) const
{
    double rhoL, vL, pL;
    std::tie(rhoL, vL, pL) = this->primitive(uL);
    double EL = this->energy(rhoL, vL, pL);
    double HL = this->enthalpy(rhoL, EL, pL);
    
    double rhoR, vR, pR;
    std::tie(rhoR, vR, pR) = this->primitive(uR);
    double ER = this->energy(rhoR, vR, pR);
    double HR = this->enthalpy(rhoR, ER, pR);
    
    double rhoS = 0.5*(rhoL + rhoR);
    double tmp = (sqrt(rhoL) + sqrt(rhoR));
    double vS = (sqrt(rhoL)*vL + sqrt(rhoR)*vR)/tmp;
    double HS = (sqrt(rhoL)*HL + sqrt(rhoR)*HR)/tmp;
    
    Eigen::VectorXd uS(n_vars);
    uS(0) = rhoS;
    uS(1) = uS(0)*vS;
    uS(2) = rhoS*HS/gamma + 0.5*(gamma-1)/gamma*rhoS*vS*vS;
    return uS;
}
Implement the following numerical fluxes:

- Rusanov's
- Lax-Friedrichs
- Roe
- HLL
- HLLC

**Hint:** The class `CentralFlux` is given as an example for implementing the central flux.

**Hint:** Implement 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 3:** Implementation of Rusanov's flux.

```cpp
class Rusanov {
public:
  explicit Rusanov(const std::shared_ptr<Model> &model)
      : model(model) {}

  Eigen::VectorXd operator()(const Eigen::VectorXd &uL,
               const Eigen::VectorXd &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:
  std::shared_ptr<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 $\Delta t$. This can not be captured by the constructor at the beginning of the simulation. The current simulation time $t$, the time-step $\Delta t$ 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.

Lax Friedrichs flux can be implemented as follows.

**Listing 4:** Implementation of Lax-Friedrichs flux.

```cpp
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 std::shared_ptr<Model> &model,
        std::shared_ptr<SimulationTime> simulation_time)
        : simulation_time(std::move(simulation_time)),
            grid(grid),
            model(model) {}

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

            auto fL = model->flux(uL);
            auto fR = model->flux(uR);

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

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

Roe flux can be implemented as follows.

**Listing 5:** Implementation of Roe flux.
class Roe{
    public:
    explicit Roe(const std::shared_ptr<Model> &model)
    : model(model) {}

    Eigen::VectorXd operator()(const Eigen::VectorXd &uL,
    const Eigen::VectorXd &uR) const
    {
        auto fL = model->flux(uL);
        auto fR = model->flux(uR);

        auto uM = model->roe_avg(uL, uR);
        auto eigvalsM = model->eigenvalues(uM);
        auto eigvecsM = model->eigenvectors(uM);

        return 0.5*((fL + fR) - eigvecsM*(eigvalsM.cwiseAbs().asDiagonal())
            *eigvecsM.fullPivLu().solve(uR-uL));
    }

    private:
    std::shared_ptr<Model> model;
};

HLL flux can be implemented as follows.

Listing 6: Implementation of HLL flux.

class HLL {
    public:
    explicit HLL(const std::shared_ptr<Model> &model) : model(model) {}

    Eigen::VectorXd operator()(const Eigen::VectorXd &uL,
    const Eigen::VectorXd &uR) const
    {
        auto fL = model->flux(uL);
        double sL = model->eigenvalues(uL).minCoeff();

        auto fR = model->flux(uR);
        double sR = model->eigenvalues(uR).maxCoeff();

        // compute Roe average
        auto uM = model->roe_avg(uL, uR);
        auto eigvalsM = model->eigenvalues(uM);
        // Einfeldt-batten speeds
        sL = std::min(sL, eigvalsM.minCoeff());
        sR = std::max(sR, eigvalsM.maxCoeff());
    }
```cpp
// numerical flux
Eigen::VectorXd f = Eigen::VectorXd::Zero(uM.size());
if (sL > 0) {
    f = fL;
} else if (sL <=0 && sR >= 0) { // fs
    f = ((sR*fL - sL*fR)+ sL*sR*(uR - uL))/(sR - sL);
} else if (sR < 0) {
    f = fR;
}

return f;
}

private:
    std::shared_ptr<Model> model;
};

The HLLC flux is implemented specifically for the Euler model, which allows us to access the routines in the Euler model that are not available through the model interface. We take this into account when registering HLLC flux.

HLLC flux can be implemented as follows.

Listing 7: Implementation of HLLC flux.

class HLLCEuler {
public:
    explicit HLLCEuler(const std::shared_ptr<Euler> &model)
        : model(model) {
        n_vars = model->get_nvars();
    }

    Eigen::VectorXd operator()(const Eigen::VectorXd &uL,
                               const Eigen::VectorXd &uR) const
    {
        auto fl = model->flux(uL);
        auto fr = model->flux(uR);

        // left state
        auto [rhoL, vL, pL] = model->primitive(uL);
        double cL = model->sound_speed(rhoL, pL);
        Eigen::VectorXd eigvals_uL(n_vars);
        eigvals_uL << vL - cL, vL, vL + cL;
        double sL = eigvals_uL.minCoeff();

        // right state
        auto [rhoR, vR, pR] = model->primitive(uR);
```
double cR = model->sound_speed(rhoR, pR);
Eigen::VectorXd eigvals_uR(n_vars);
eigvals_uR << vR - cR, vR, vR + cR;
double sR = eigvals_uR.maxCoeff();

// Roe pressure
double sM = (pR - pL + (rhoL*vL)*(sL - vL) - (rhoR*vR)*(sR - vR))
/(rhoL*(sL - vL) - rhoR*(sR - vR));
double pM = 0.5*(pR + pL + rhoL*(sL - vL)*(sM - vL)
+ rhoR*(sM - vR)*(sR - vR));

// numerical flux
Eigen::VectorXd f = Eigen::VectorXd::Zero(fL.size());
if (sL >= 0) {
f = fL;
} else if(sL <= 0 && sM > 0) //fLs
{
    Eigen::VectorXd u_prim(n_vars);
    double rhoLs = rhoL*(vL - sL)/(sM - sL);
    u_prim << rhoLs, sM, pM;
    auto uLs = model->prim_to_cons(u_prim);
    f = fL + sL*(uLs - uL);
}
else if(sM <= 0 && sR >= 0) //fRs
{
    Eigen::VectorXd u_prim(n_vars);
    double rhoRs = rhoR*(vR - sR)/(sM - sR);
    u_prim << rhoRs, sM, pM;
    auto uRs = model->prim_to_cons(u_prim);
    f = fR + sR*(uRs - uR);
} else if (sR <= 0) {
    f = fR;
}

return f;

private:
    std::shared_ptr<Euler> model;
    int n_vars;
};
1c)

Templates: include/ancse/limiters.hpp, include/ancse/reconstruction.hpp

Implement piecewise linear reconstruction of the trace values \( U_{i+1/2}^{\pm} \) based on

- conservative variables \( u_{\text{cons}} = (\rho, \rho v, E) \)
- primitive variables \( u_{\text{prim}} = (\rho, v, p) \)

with the following slope-limiters:

- minmod
- superbee
- monotonized central

**Hint:** The class *PWConstantReconstruction* is given as an example for implementing piecewise constant reconstruction.

**Hint:** Consider implementing this using a template class which accepts the slope-limiter as a template parameter.

**Hint:** Implement tests in *tests/test_reconstruction.cpp*.

**Solution:** Some commonly used functions have been provided already.

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 performs very few operations and is expected to call in a very tight loop we chose static polymorphism.

**Listing 8:** Implementation of different slope limiters.

```cpp
struct MinMod {
    inline double operator()(double a, double b) const
    { return minmod(a, b); }
};

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);
    }
};
```
Reconstruction in the conserved and primitive variables has been implemented as template specializations of a base template, which allows us to choose between conserved or primitive reconstruction when they are registered.

Listing 9: Implementation of conserved and primitive reconstruction.

```cpp
enum {Conserved, Primitive};

template <class SlopeLimiter, int ReconstructionVars>
class PWLinearReconstruction;

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

        void set(const Eigen::MatrixXd &u) const
            { up.resize(u.rows(),u.cols());
                up = u;
            }

        std::pair<Eigen::VectorXd, Eigen::VectorXd> operator()(int i) const
            { return (*this)(up.col(i - 1), up.col(i), up.col(i + 1), up.col(i + 2));
            }

        std::pair<Eigen::VectorXd, Eigen::VectorXd> operator()(const Eigen::VectorXd &ua,
                    const Eigen::VectorXd &ub,
                    const Eigen::VectorXd &uc,
                    const Eigen::VectorXd &ud) const
            {
                auto sL = ub - ua;
                auto sM = uc - ub;
                auto sR = ud - uc;

                Eigen::VectorXd uL(ua.size());
                Eigen::VectorXd uR(ua.size());

                return {uL, uR};
            }
```
```cpp
for (int i = 0; i < uL.size(); i++)
{
    uL(i) = ub(i) + 0.5 * slope_limiter(sL(i), sM(i));
    uR(i) = uc(i) - 0.5 * slope_limiter(sM(i), sR(i));
}

return {std::move(uL), std::move(uR)};
}

private:
    SlopeLimiter slope_limiter;
    mutable Eigen::MatrixXd up;
};

template <class SlopeLimiter>
class PWLinearReconstruction<SlopeLimiter, Primitive> {
public:
    explicit PWLinearReconstruction(const std::shared_ptr<Model> &model,
                                       const SlopeLimiter &slope_limiter)
        : model(model), slope_limiter(slope_limiter) {}

    void set(const Eigen::MatrixXd &u) const
    {
        up.resize(u.rows(), u.cols());
        for (int i = 0; i < u.cols(); ++i) {
            up.col(i) = model->cons_to_prim(u.col(i));
        }
    }

    std::pair<Eigen::VectorXd, Eigen::VectorXd> operator()(int i) const {
        auto [uLp, uRp] = (*this)(up.col(i - 1), up.col(i), up.col(i + 1), up.col(i + 2));
        return {std::move(model->prim_to_cons(uLp)), std::move(model->prim_to_cons(uRp))};
    }

    std::pair<Eigen::VectorXd, Eigen::VectorXd> operator()(const Eigen::VectorXd &ua,
                                                      const Eigen::VectorXd &ub,
                                                      const Eigen::VectorXd &uc,
                                                      const Eigen::VectorXd &ud) const
    {
```
auto sL = ub - ua;
auto sM = uc - ub;
auto sR = ud - uc;

Eigen::VectorXd uL(ua.size());
Eigen::VectorXd uR(ua.size());

for (int i=0; i<uL.size(); i++)
    { 
        uL(i) = ub(i) + 0.5 * slope_limiter(sL(i), sM(i));
        uR(i) = uc(i) - 0.5 * slope_limiter(sM(i), sR(i));
    }

return {std::move(uL), std::move(uR)};
}

private:
    std::shared_ptr<Model> model;
    SlopeLimiter slope_limiter;

    mutable Eigen::MatrixXd up;
};

1d)

Template: include/ancse/fvm_rate_of_change.hpp

Complete the loop that applies your fluxes and numerical reconstructions.

**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 switch to dynamic polymorphism.

**Listing 10:** Implementation of the “flux loop” for FVM.

```c++
virtual void operator() (Eigen::MatrixXd &dudt,
                       const Eigen::MatrixXd &u0) const override {
    // ANCSE_CUT_START_TEMPLATE
    // ANCSE_COMMENT implement the flux loop here.
    auto n_cells = grid.n_cells;
    auto n_ghost = grid.n_ghost;

    double dx = grid.dx;
    Eigen::VectorXd fL = Eigen::VectorXd::Zero(u0.rows());
    Eigen::VectorXd fR = Eigen::VectorXd::Zero(u0.rows());
```
Eigen::VectorXd uL(u0.rows()), uR(u0.rows());

reconstruction.set(u0);

for (int i = n_ghost-1; i < n_cells - n_ghost; ++i) {
    std::tie(uL, uR) = reconstruction(i);
    fL = fR;
    fR = numerical_flux(uL, uR);

    dudt.col(i) = (fL - fR) / dx;
}

//// ANCSE_END_TEMPLATE

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

**Hint:** The class ForwardEuler is given as example for implementing forward Euler.

**Solution:** The formula for SSP2 is

\[
    u^* = u_0 + \Delta t L(u^n)
\]

\[
    u^{**} = u^* + \Delta t L(u^*)
\]

\[
    u^{n+1} = \frac{1}{2}(u^n + u^{**}).
\]

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.

We implement different time-integration schemes with dynamic polymorphism. Therefore, we define a abstract base class TimeIntegrator which only defines the interface and nothing else.

The complete implementation is

**Listing 11:** Implementation of SSP2 Runge-Kutta.

```cpp
class SSP2 : public RungeKutta {
private:
    using super = RungeKutta;

public:
    SSP2(std::shared_ptr<RateOfChange> rate_of_change_,
         std::shared_ptr<BoundaryCondition> boundary_condition_,
```
std::shared_ptr<Limiting> limiting_,
int n_rows,
int n_cols)
: super(std::move(rate_of_change_),
      std::move(boundary_condition_),
      std::move(limiting_),
u_star(n_rows, n_cols),
dudt(n_rows, n_cols) {}

virtual void operator()(Eigen::MatrixXd &u1,
                        const Eigen::MatrixXd &u0,
double dt) const override
{
        (*rate_of_change)(dudt, u0);
        u_star = u0 + dt * dudt;
        post_euler_step(u_star);

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

        u1 = 0.5 * (u0 + u_star);
        post_euler_step(u_star);
}

private:
    mutable Eigen::MatrixXd u_star;
    mutable Eigen::MatrixXd dudt;
}

1f)

Templates: include/ancse/cfl_condition.hpp, src/ancse/cfl_condition.cpp
Implement the CFL condition. This should be a new class that derives from CFLCondition, which implements the computation of a CFL-satisfying \( dt \) for a generic problem.

**Hint:** Implement tests in tests/test_cfl_condition.cpp.

**Solution:** The CFL condition is

\[
\Delta t \leq CFL \frac{\Delta x}{a_{\text{max}}}, \quad a_{\text{max}} = \max_i f'(U_i), \tag{9}
\]

with \( CFL < 1 \).
We implement the CFL condition for FVM as a template specialization as follows, where the variable “FVM” is declared in `includes.hpp`.

**Listing 12: Implementation of CFL condition for FVM.**

```cpp
/// Compute the CFL condition for generic models.
/**
 * Compute the maximum eigenvalue given all the data, and compute the
 * CFL restricted time-step accordingly.
 *
 * Note: this requires looping over the entire grid, and computing a
 * reduction, i.e. max.
 */
template <int DiscretizationType>
class StandardCFLCondition;

template <>
class StandardCFLCondition <FVM> : public CFLCondition {
public:
    StandardCFLCondition(const Grid &grid,
                          const std::shared_ptr<Model> &model,
                          double cfl_number);

    virtual double operator()(const Eigen::MatrixXd &u) const override;

private:
    Grid grid;
    std::shared_ptr<Model> model;
    double cfl_number;
};
```

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

Start by registering your implementation of Euler equations model, see `src/ancse/model.cpp`.

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`.

Finally, register your implementation of SSP2, see `src/ancse/runge_kutta.cpp`. Note, follow the example of how it is done for `ForwardEuler`.

**Solution:** Please look at the provided reference implementation.
1h)

You can try the following with your FVM code:

- Consider Sod’s shock tube problem on a domain $\Omega := (0, 1)$ with outflow boundary conditions and initial discontinuity at $x = 0.5$. The initial states to the left and to the right of the discontinuous interface are

$$
\begin{pmatrix}
\rho_L \\
v_L \\
p_L
\end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \quad \begin{pmatrix}
\rho_R \\
v_R \\
p_R
\end{pmatrix} = \begin{pmatrix} 0.125 \\ 0 \\ 0.1 \end{pmatrix}.
$$

For details about this experiment, you can read:

- Section 14.13 of Finite Volume Methods for Hyperbolic problems, R. LeVeque, 2002. You can find this book through the course web page for AdvNumCSE.

Study the performance of the different numerical schemes for this experiment.

Is the piecewise linear reconstruction based on conservative variables better than the one based on primitive variables?

**Solution:**

A comparison for different fluxes is given in Figures 1 and 2.

![Figure 1: Comparison of fluxes for first-order FVM; density and velocity plots; $T = 0.2$.](image-url)
Figure 2: Comparison of fluxes for first-order FVM; pressure and energy plots; $T = 0.2$.

The comparison for different limiters is given in Figure 3.

Figure 3: Comparison of limiters for second-order FVM with limiting in primitive variables; density, velocity, pressure and energy plots; $T = 0.2$. 
The superbee limiter is compared for limiting in conserved and primitive variables in Figure 4.

Figure 4: Comparison of limiting in conserved versus primitive variables for second-order FVM using superbee limiter; density, velocity, pressure and energy plots; $T = 0.2$.

- Consider a "vacuum" problem on a domain $\Omega := (0, 1)$ with outflow boundary conditions and initial discontinuity at $x = 0.5$. The initial states to the left and to the right of the discontinuous interface are

$$\begin{pmatrix} \rho_L \\ v_L \\ p_L \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix}, \quad \begin{pmatrix} \rho_R \\ v_R \\ p_R \end{pmatrix} = \begin{pmatrix} 1 \\ +2 \\ 1 \end{pmatrix}. \quad (11)$$

Does the numerical scheme with Roe’s numerical flux work for this experiment?

Roe’s numerical flux fails for this experiment, see Figure 4. From the given initial data, we suspect that it happens because we have two rarefaction waves moving in the opposite direction, originating from the interface at $x = 0.5$. 

21
**Figure 5:** Results for the "vacuum" test with Roe’s flux; density, velocity, pressure and energy plots; $T = 0.005$.

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

**Hint:** Use piecewise linear reconstruction with minmod limiter, Rusanov’s flux and SSP2 time-stepping on 2048 cells as the reference solution.

**Discontinuous Galerkin method**

Let $\mathbb{P}_p$ be the space of the polynomials with maximum degree $p$. Define a scalar *piecewise* polynomial space

$$ W_h^p := \{ w \in L^2(\Omega) : w|_K \in \mathbb{P}_p(K), \ K \in \mathcal{T}_h \}. $$

(12)

This means that a function $w \in W_h^p$ is a polynomial in the cells and it is discontinuous across cell interfaces.

We can build an orthonormal basis for $W_h^p$ using Legendre polynomials, which are given below on
the domain $[-1, 1]$ for $p = 2$:

\[ \begin{align*}
P_0(x) &= 1, & P_1(x) &= x, & P_2(x) &= \frac{1}{2}(3x^2 - 1).
\end{align*} \]

The Legendre polynomials are orthogonal:

\[ \int_{-1}^{+1} P_k(x) P_l(x) \, dx = \frac{2}{2k + 1} \delta_{kl}, \]

where $\delta_{kl} = 1$, if $k = l$, and $\delta_{kl} = 0$, if $k \neq l$. Transform $P_k$ to the domain $[0, 1]$ and define the normalized functions

\[ \phi_k(x) := \sqrt{2k + 1} P_k(2x - 1) \chi_{[0, 1]}(x), \]

such that

\[ \int_{0}^{+1} \phi_k(x) \phi_l(x) \, dx = \delta_{kl}. \]

Here $\chi_{[0, 1]}(x) = 1$, if $x \in [0, 1]$ and $\chi_{[0, 1]}(x) = 0$, otherwise.

Given $N$ grid cells, uniformly spaced with $x_{i+1/2} - x_{i-1/2} = h$, let $\{\varphi_{i,j}\}$ denote the basis of $W_p^h$, i.e. a function $w \in W_p^h$ can be written as

\[ w(x) = \sum_{i=0}^{N} \sum_{j=0}^{p} w_{i,j} \varphi_{i,j}(x), \]

where we define the function

\[ \varphi_{i,j}(x) := \frac{1}{\sqrt{h}} \phi_j \left( \frac{x - x_{i-1/2}}{h} \right). \tag{13} \]

Note, $\varphi_{i,j}(x) = 0$, if $x \notin (x_{i-1/2}, x_{i+1/2})$, i.e. the function $\varphi_{i,j}$ has local support on $(x_{i-1/2}, x_{i+1/2})$.

For a given $m \in \mathbb{N}$, we can easily define a vector piecewise polynomial space

\[ W^h := \left[ W^p_h \right]^m, \tag{14} \]

i.e. for a vector function $w \in W^h$, its components $u_l \in W^p_h$ for $l = 1, \ldots, m$. For 1D Euler equations $m = 3$.

We seek a solution of the form

\[ u_h(x, t) = \sum_{i=0}^{N} \sum_{j=0}^{p} U_{i,j}(t) \varphi_{i,j}(x), \]

with $U_{i,j} : \mathbb{R} \to \mathbb{R}^m$ the coefficient of $\varphi_{i,j}$ in the expansion of $u_h(\cdot, t)$. Note that, for fixed $t \geq 0$, $u_h(\cdot, t)$ can be uniquely identified with a vector $(U_{i,j})_{l=1, j=0, i=1}^{l=m, j=p, i=N}$ of coefficients in $\mathbb{R}^{m(p+1)N}$.

To derive the DG formulation for the Euler equations (1), we take its inner product with a test function $w_h \in W^h$, integrate over an arbitrary cell $K_i := (x_{i-1/2}, x_{i+1/2}) \in \mathcal{T}_h$ and use integration
by parts:
\[
\int_{K_i} (\mathbf{u}_h)_t \cdot \mathbf{w}_h \, dx + \int_{K_i} \mathbf{f}(\mathbf{u}_h)_x \cdot \mathbf{w}_h \, dx = 0
\]
\[
d\left( \int_{K_i} \mathbf{u}_h \cdot \mathbf{w}_h \, dx \right) + \left( \mathbf{f}(\mathbf{u}) \cdot \mathbf{w}_h \right) |_{x+1/2} - \left( \mathbf{f}(\mathbf{u}) \cdot \mathbf{w}_h \right) |_{x-1/2} - \int_{K_i} \mathbf{f}(\mathbf{u}_h) \cdot (\mathbf{w}_h)_x \, dx = 0.
\]
Here \( \mathbf{u}_h \in \mathbf{W}_h^p \) is the approximate solution and the flux through the interface \( i + 1/2, \mathbf{f}(\mathbf{u}(x_{i+1/2})) \), can be approximated with a numerical flux \( \mathbf{F} \) by
\[
\mathbf{f}(\mathbf{u}(x_{i+1/2})) \approx \hat{\mathbf{f}}_{i+1/2} := \mathbf{F} \left( \mathbf{u}_h(x_{i+1/2}), \mathbf{u}_h(x_{i+1/2}) \right).
\]
(15)
The semi-discrete formulation of the DG method is
\[
\frac{d}{dt} \left( \int_{K_i} \mathbf{u}_h \cdot \mathbf{w}_h \, dx \right) = \hat{\mathbf{f}}_{i-1/2} \cdot \mathbf{w}_h |_{x_{i-1/2}} - \hat{\mathbf{f}}_{i+1/2} \cdot \mathbf{w}_h |_{x_{i+1/2}} + \int_{K_i} \mathbf{f}(\mathbf{u}_h) \cdot (\mathbf{w}_h)_x \, dx.
\]
(16)

**Remark:** If \( \mathbf{W}_0 \) is used, i.e. the space of piecewise constant functions, the volume integral on the right-hand side in (16) vanishes as \( (\mathbf{w}_h)_x = 0 \), and dividing by the cell size \( h \) we re-obtain the FVM semi-discrete formulation (4).

**Remark:** By construction of the DG method, it is easy to evaluate the values of \( \mathbf{u} \) to the left and to the right of cell interfaces for computing numerical fluxes. No reconstruction is needed!

**Remark:** For \( p \geq 1 \), we need to limit the DG solution coefficients corresponding to polynomials of degree greater than equal to 1. However, one can separate the evolution step and the limiting procedure. For example - if we use forward Euler time integrator to solve (16), then first we will evolve the solution coefficients from \( t_n \) to \( t_{n+1} \) and then perform the limiting.

**Remark:** Since \( \mathbf{u}_h(\cdot, t) \in \mathbf{W}_h^p \), we have
\[
\int_{K_i} \mathbf{u}_h(\cdot, t) \cdot \mathbf{w}_h \, dx = \int_{K_i} \sum_{k=0}^{p} U_{i,k}(t) \varphi_{i,k} \cdot \mathbf{w}_h \, dx.
\]
Due to orthonormality of the basis polynomials, taking \( \mathbf{w}_h := \varphi_{i,j} \mathbf{e}_l \), with \( \mathbf{e}_l \in \mathbb{R}^m \) the unit vector with \( (\mathbf{e}_l)_l^j = \delta_{lj} \), we obtain
\[
\int_{K_i} \mathbf{u}_h \cdot \mathbf{w}_h \, dx = \sum_{k=0}^{p} U_{i,k,l}(t) \delta_{lk} = U_{i,j,l}(t)
\]

**Remark:** For implementation, we re-arrange the vector of coefficients \( (U_{i,j,l})_{l=0,j=0,i=1}^{m+j=p,i=N} \in \mathbb{R}^{m(p+1)N} \) into a matrix \( \mathbf{U} \in \mathbb{R}^{m(p+1)\times N} \) of \( m(p+1) \) rows and \( N \) columns, with entries
\[
U_{i,j+(p+1)l} := U_{i+l,j,l+1}, \quad \text{for } j = 0, \ldots, p, \quad l = 0, \ldots, m-1, \quad i = 0, \ldots, N-1.
\]
This particular arrangement is used in the initialization step, which has already been provided, of the DG solver.
Template: src/ancse/polynomial_basis.cpp.

Implement the basis functions $\phi_k$, with maximum degree 2, as described above.

PolynomialBasis::operator() takes as input one point $\xi \in [0,1]$, and return a vector of $p + 1$ components containing $\{\phi_k(\xi)\}_{k=0}^p$, scaled by an appropriate scaling factor, which you can assume is already contained in PolynomialBasis::scaling_factor.

Do the same in function PolynomialBasis::deriv for the spatial derivatives $\partial_\xi \phi_k(\xi)$.

Solution:

Listing 13: Implementation of the Legendre polynomial basis.

```cpp
/// Computes the Legendre polynomial basis
/// at a given reference point $x_i \in [0,1]
Eigen::VectorXd PolynomialBasis :: operator() (double xi) const
{
    // ANCE_CUT_START_TEMPLATE
    Eigen::VectorXd basis(p+1);
    basis(0) = 1;
    if (p > 0) {
        double z = 2*xi - 1;
        basis(1) = sqrt(3)*z;
        if (p > 1) {
            basis(2) = 0.5*sqrt(5)*(3*z*z-1);
        }
    }
    return (basis*scaling_factor);
    // ANCE_END_TEMPLATE
    // ANCE_RETURN_VALUE Eigen::VectorXd::Zero(p+1);
}

/// Computes the derivative of Legendre polynomial basis
/// at a given reference point $x_i \in [0,1]
Eigen::VectorXd PolynomialBasis :: deriv (double xi) const
{
    // ANCE_CUT_START_TEMPLATE
    Eigen::VectorXd basis_deriv(p+1);
    basis_deriv(0) = 0;
    if (p > 0) {
        double z = 2*xi - 1;
        basis_deriv(1) = 2*sqrt(3);
    }
    return (basis_deriv*scaling_factor);
    // ANCE_END_TEMPLATE
    // ANCE_RETURN_VALUE Eigen::VectorXd::Zero(p+1);
}
```
if (p > 1) {
    basis_deriv(2) = 6*sqrt(5)*z;
}
}

return (basis_deriv*scaling_factor);

/// ANCSE_END_TEMPLATE
/// ANCSE_RETURN_VALUE Eigen::VectorXd::Zero(p+1);

1j)

Template: src/ancse/dg_handler.cpp.

Implement the function build_solution, which builds the solution at a given point in a specified cell.

That is, DGHandler::build_solution receives the vector of coefficients $U_i \in \mathbb{R}^{m(p+1)}$ of the approximate solution $u_h$ in cell $K_i$, and a point $x$ in the cell $K_i$, and returns $(u_h)_l(x) = \sum_{k=0}^{p} U_i,k+l(p+1) \varphi_{i,k}(x)$, for $l = 0, \ldots, m-1$. Note, the physical point $x$ should be converted to the corresponding reference point $\xi \in [0,1]$, because $\varphi_{i,k}(x)$ is computed through $\phi_k$ using equation 13.

Implement the function build_cell_avg, which builds the cell averages in the given cells.

That is, DGHandler::build_cell_avg produces a matrix $\bar{u}_h \in \mathbb{R}^{m \times N}$, where

$$(\bar{u}_h)_{l,i} = \frac{1}{h} \int_{K_i} (u_h)_l(x) \, dx,$$

where $(u_h)_l$ refers to the $l$-th component of $u_h$.

**Hint:** What relationship is there between $\frac{1}{h} \int_{K_i} (u_h)_l(x) \, dx$ and the coefficient corresponding to $\varphi_{i,0}$ in $u_h|_{K_i}$?

**Solution:**

Legendre polynomials are orthogonal, then we can write

$$\int_{K_i} (u_h)_l(x) \, dx = U_{i,l(p+1)} \int_{K_i} \varphi_{i,0}(x) \, dx + \sum_{k=1}^{p} U_{i,k+l(p+1)} \int_{K_i} \varphi_{i,k}(x) \, dx$$

$$= \frac{1}{\sqrt{h}} \left( U_{i,l(p+1)} h \int_{0}^{1} \phi_0(\xi(x)) \, d\xi + \sum_{k=1}^{p} U_{i,k+l(p+1)} h \int_{0}^{1} \phi_k(\xi(x)) \phi_0(\xi(x)) \, d\xi \right).$$

$$\Rightarrow \frac{1}{h} \int_{K_i} (u_h)_l(x) \, dx = \frac{1}{\sqrt{h}} U_{i,l(p+1)}.$$
Listing 14: Building the solution and the cell averages.

```cpp
/// build solution from DG coefficients and the basis
/// pre-evaluated at a certain point
Eigen::VectorXd DGHandler::build_sol(const Eigen::VectorXd& u,
                                       const Eigen::VectorXd& basis) const
{
    Eigen::VectorXd uSol = Eigen::VectorXd::Zero(n_vars);

    // ANCSE_CUT_START_TEMPLATE
    for (int i = 0; i < n_vars; i++) {
        for (int q = 0; q < n_coeff; q++) {
            uSol(i) += u(q + i*n_coeff)*basis(q);
        }
    }
    // ANCSE_END_TEMPLATE

    return uSol;
}

/// build solution from DG coefficients at a given reference point
Eigen::VectorXd DGHandler::build_sol(const Eigen::VectorXd& u,
                                      double xi) const
{
    Eigen::VectorXd uSol(n_vars);

    // ANCSE_CUT_START_TEMPLATE
    auto basis = poly_basis(xi);
    uSol = build_sol(u, basis);
    // ANCSE_END_TEMPLATE

    return uSol;
}

/// build cell average
Eigen::MatrixXd DGHandler::build_cell_avg (const Eigen::MatrixXd& u) const
{
    auto n_cells = u.cols();
    Eigen::MatrixXd u0 (n_vars, n_cells);

    // ANCSE_CUT_START_TEMPLATE
    double basis0 = (poly_basis(1.0))(0);
    for (int j = 0; j < n_cells; j++) {
        for (int i = 0; i < n_vars; i++) {
            u0(i, j) += u(i, j) * basis0;
        }
    }
    // ANCSE_END_TEMPLATE

    return u0;
}
```
\[ u_0(i, j) = u(i*n_{\text{coeff}}, j)*\text{basis}0; \]

return \( u_0 \);

1k)

Templates: include/ancse/cfl_condition.hpp, src/ancse/cfl_condition.cpp

Implement the CFL condition using the cell averages \( \bar{u}_h \in \mathbb{R}^{m \times N} \). This should be a new class that derives from CFLCondition, which implements the computation of a CFL-satisfying \( dt \) for a generic problem.

**Hint:** Implement tests in tests/test_cfl_condition.cpp.

**Solution:** We implement the CFL condition for DG as a template specialization as follows, where the variable “DG” is declared in includes.hpp.

**Listing 15:** Implementation of CFL condition.

```cpp
template <>
class StandardCFLCondition <DG> : public CFLCondition {
    public:
        StandardCFLCondition(const Grid &grid,
                             const std::shared_ptr<Model> &model,
                             const DGHandler &dg_handler,
                             double cfl_number);

        virtual double operator()(const Eigen::MatrixXd &u) const override;

    private:
        Grid grid;
        std::shared_ptr<Model> model;
        DGHandler dg_handler;
        double cfl_number;
};
```

1l)

Template: src/ancse/dg_rate_of_change.cpp.
Implement the functions `eval_numerical_flux` and `eval_volume_integral`, which compute the numerical flux term and volume integral on the right-hand side in the formulation (16), respectively.

Register numerical fluxes in `make dg_rate_of_change`, as you have already done for FVM.

**Hint:** You can test your implementation without any limiting for $p = 0$ with forward Euler, it should give the same results as the first-order FVM.

**Solution:**

**Listing 16:** Implementation of the numerical flux and volume integral terms in DG.

```cpp
// DG numerical flux term
template <class NumericalFlux>
void DGRateOfChange<NumericalFlux>::eval_numerical_flux (Eigen::MatrixXd &dudt,
const Eigen::MatrixXd &u0) const {

  // ANCSE_CUT_START_TEMPLATE
  // ANCSE_COMMENT implement the loop for DG numerical flux term.
  auto n_cells = grid.n_cells;
  auto n_ghost = grid.n_ghost;
  auto n_vars = model->get_nvars();
  int n_coeff = 1 + poly_basis.get_degree();

  Eigen::VectorXd fL = Eigen::VectorXd::Zero(n_vars);
  Eigen::VectorXd fR = Eigen::VectorXd::Zero(n_vars);
  Eigen::VectorXd uL(n_vars), uR(n_vars);

  // evaluate basis to the left and right of a cell interface
  auto basisL = poly_basis(1.0);
  auto basisR = poly_basis(0.0);

  for (int j = n_ghost-1; j < n_cells - n_ghost; ++j) {
    uL = dg_handler.build_sol(u0.col(j), basisL);
    uR = dg_handler.build_sol(u0.col(j+1), basisR);
    fL = fR;
    fR = numerical_flux(uL, uR);

    for (int i = 0; i < n_vars; i++) {
      for (int q = 0; q < n_coeff; q++) {
        dudt(q + i*n_coeff, j) += fL(i)*basisR(q) - fR(i)*basisL(q);
      }
    }
  }
}
```

---

Implement the functions `eval_numerical_flux` and `eval_volume_integral`, which compute the numerical flux term and volume integral on the right-hand side in the formulation (16), respectively.

Register numerical fluxes in `make dg_rate_of_change`, as you have already done for FVM.

**Hint:** You can test your implementation without any limiting for $p = 0$ with forward Euler, it should give the same results as the first-order FVM.

**Solution:**

**Listing 16:** Implementation of the numerical flux and volume integral terms in DG.

```cpp
// DG numerical flux term
template <class NumericalFlux>
void DGRateOfChange<NumericalFlux>::eval_numerical_flux (Eigen::MatrixXd &dudt,
const Eigen::MatrixXd &u0) const {

  // ANCSE_CUT_START_TEMPLATE
  // ANCSE_COMMENT implement the loop for DG numerical flux term.
  auto n_cells = grid.n_cells;
  auto n_ghost = grid.n_ghost;
  auto n_vars = model->get_nvars();
  int n_coeff = 1 + poly_basis.get_degree();

  Eigen::VectorXd fL = Eigen::VectorXd::Zero(n_vars);
  Eigen::VectorXd fR = Eigen::VectorXd::Zero(n_vars);
  Eigen::VectorXd uL(n_vars), uR(n_vars);

  // evaluate basis to the left and right of a cell interface
  auto basisL = poly_basis(1.0);
  auto basisR = poly_basis(0.0);

  for (int j = n_ghost-1; j < n_cells - n_ghost; ++j) {
    uL = dg_handler.build_sol(u0.col(j), basisL);
    uR = dg_handler.build_sol(u0.col(j+1), basisR);
    fL = fR;
    fR = numerical_flux(uL, uR);

    for (int i = 0; i < n_vars; i++) {
      for (int q = 0; q < n_coeff; q++) {
        dudt(q + i*n_coeff, j) += fL(i)*basisR(q) - fR(i)*basisL(q);
      }
    }
  }
}
```

---
template <class NumericalFlux>
void DGRateOfChange<NumericalFlux>::eval_volume_integral(Eigen::MatrixXd &dudt, const Eigen::MatrixXd &u0) const {
    int n_coeff = 1 + poly_basis.get_degree();
    if (n_coeff == 1) {
        return;
    }
    auto n_cells = grid.n_cells;
    auto n_ghost = grid.n_ghost;
    auto n_vars = model->get_nvars();
    Eigen::VectorXd flux = Eigen::VectorXd::Zero(n_vars);
    Eigen::VectorXd uSol(n_vars);
    int n_quad = quad_points.size();
    Eigen::MatrixXd basis(n_coeff, n_quad);
    Eigen::MatrixXd basis_deriv(n_coeff, n_quad);

    // eval basis and its derivate for all quadrature points
    for (int k = 0; k < quad_points.size(); k++) {
        basis.col(k) = poly_basis(quad_points(k));
        basis_deriv.col(k) = poly_basis.deriv(quad_points(k));
    }

    for (int j = n_ghost-1; j < n_cells - n_ghost; ++j) {
        for (int k = 0; k < n_quad; k++) {
            uSol = dg_handler.build_sol(u0.col(j), basis.col(k));
            flux = model->flux(uSol);
            auto basis_deriv_ = basis_deriv.col(k);
            for (int i = 0; i < n_vars; i++) {
                for (int q = 0; q < n_coeff; q++) {
                    dudt(q + i*n_coeff, j) += quad_weights(k)*flux(i)*basis_deriv_(q);
                }
            }
        }
    }
}

DG Limiting

Without delving into too many details, we will provide a recipe to do limiting for the scalar case, which can then be applied component-wise for systems:

Define

\[
\bar{u}_i^-, \bar{u}_i^+ := \sum_{j=1}^p u_{i,j} \varphi_{i,j}(x^+_{i+1/2}), \quad \bar{u}_i^- := \sum_{j=1}^p u_{i,j} \varphi_{i,j}(x^-_{i-1/2}).
\]

This implies

\[
u_h(x^+_{i+1/2}) = \bar{u}_i^+ + \bar{u}_i^-,
\]

\[
u_h(x^-_{i-1/2}) = \bar{u}_i^- - \bar{u}_i^+.
\]

Additionally, define

\[
\Delta^+ \bar{u}_i := \bar{u}_{i+1,0} - \bar{u}_{i,0}, \quad \Delta^- \bar{u}_i := \bar{u}_{i,0} - \bar{u}_{i-1,0}.
\]

Using the so-called van Leer limiter:

\[
\hat{\bar{u}}_i^- := g(\bar{u}_i^-, \Delta^- u_{i,0}, \Delta^+ u_{i,0}), \quad \hat{\bar{u}}_i^+ := g(\bar{u}_i^+, \Delta^- u_{i,0}, \Delta^+ u_{i,0}),
\]

with the minmod function

\[
g(a_1, \ldots, a_n) := \begin{cases} 
\min_{1 \leq i \leq n} |a_i|, & s = \text{sgn}(a_1) = \ldots = \text{sgn}(a_n), \\
0, & \text{otherwise}. 
\end{cases}
\]

Finally, we recompute the coefficients post limiting \(\hat{u}_{i,j}\) from \(\hat{\bar{u}}_i^-\) and \(\hat{\bar{u}}_i^+\). You can easily check that \(\hat{u}_{i,j}\) can be uniquely determined for \(p = 1, 2\).

Template: \texttt{src/ancse/dg\_handler.cpp}.

Implement the function \texttt{build\_split\_sol}, which returns \(\bar{u}_{i,0}, \bar{u}_i^+\) and \(\bar{u}_i^-\), c.f. (17).

Implement the function \texttt{build\_limit\_coeffs}, which computes the coefficients \(\hat{U}_{i,j}\) after limiting.

Solution:
Listing 17: Implementation of the DG solution splitting and computation of the coefficients post limiting.

/// build split solution uSol_m = u0 + um, uSol_p = u0 - up
/// from DG coefficients
std::tuple<Eigen::MatrixXd, Eigen::MatrixXd, Eigen::MatrixXd> DGHandler::build_split_sol(const Eigen::MatrixXd& u) const
{
    auto n_cells = u.cols();
    if (n_coeff > 3) {
        throw std::runtime_error("Limiter not implemented for higher than 3rd order");
    }

    auto u0 = build_cell_avg(u);
    Eigen::MatrixXd um = Eigen::MatrixXd::Zero(n_vars, n_cells);
    Eigen::MatrixXd up = Eigen::MatrixXd::Zero(n_vars, n_cells);

    // ANCSE_CUT_START_TEMPLATE
    if (n_coeff > 1) {
        auto basis_m = poly_basis(1.0);
        auto basis_p = poly_basis(0.0);

        for (int j = 0; j < n_cells; j++) {
            for (int i = 0; i < n_vars; i++) {
                for (int q = 1; q < n_coeff; q++) {
                    um(i, j) += u(q + i*n_coeff, j)*basis_m(q);
                    up(i, j) -= u(q + i*n_coeff, j)*basis_p(q);
                }
            }
        }
    }
    // ANCSE_END_TEMPLATE

    return {std::move(u0), std::move(um), std::move(up)};
}

/// build DG coefficients from uSol_m = u0 + um, uSol_p = u0 - up
void DGHandler::compute_limit_coeffs(Eigen::MatrixXd &u,
                                      Eigen::MatrixXd &um,
                                      Eigen::MatrixXd &up) const
{
    if (n_coeff == 1) {
        return;
    }
    else if (n_coeff > 3) {
        throw std::runtime_error(32
"Limiter not implemented for higher than 3rd order";

auto n_cells = u.cols();
auto basis = poly_basis(1.0);

if (n_coeff == 2) {
    for (int j = 0; j < n_cells; j++) {
        for (int i = 0; i < n_vars; i++) {
            u(1 + i*n_coeff, j) = um(i,j)/basis(1);
        }
    }
}
else if (n_coeff == 3) {
    for (int j = 0; j < n_cells; j++) {
        for (int i = 0; i < n_vars; i++) {
            u(1 + i*n_coeff, j) = 0.5*(um(i,j) + up(i,j))/basis(1);
            u(2 + i*n_coeff, j) = 0.5*(um(i,j) - up(i,j))/basis(2);
        }
    }

Template: src/ancse/dg_limiting.cpp.

Implement the DG limiting procedure for the conserved variables.

Solution:

Listing 18: Implementation of the DG limiting loop.

void DGLimiting <Limiter> :: operator()(Eigen::MatrixXd &u) const
{
    if (!dg_handler.bool_limit_sol()) {
        return;
    }

    auto [u0, um, up] = dg_handler.build_split_sol(u);
}
Eigen::MatrixXd uml = Eigen::MatrixXd::Zero(um.rows(), um.cols());
Eigen::MatrixXd upl = Eigen::MatrixXd::Zero(um.rows(), um.cols());

for (int i = n_ghost - 1; i < n_cells - n_ghost; i++)
{
    auto [uml_, upl_] = (*this)(u0.col(i-1),
                                u0.col(i),
                                u0.col(i+1),
                                um.col(i),
                                up.col(i));
    uml.col(i) = uml_;
    upl.col(i) = upl_;}

dg_handler.compute_limit coeffs(u, uml, upl);

/// ANCSE_END TEMPLATE

/// DG limiting procedure
/** uc0 : cell average of cell 'i'
 * um0 : cell average of cell 'i-1'
 * up0 : cell average of cell 'i+1'
 * uSol_{right_face_of_cell_i} = uc0 + um
 * uSol_{left_face_of_cell_i} = uc0 - up
 */
template <class Limiter>
inline std::pair<Eigen::VectorXd, Eigen::VectorXd>
DGLimiting<Limiter> :: operator()(const Eigen::VectorXd &um0,
                                   const Eigen::VectorXd &uc0,
                                   const Eigen::VectorXd &up0,
                                   const Eigen::VectorXd &um,
                                   const Eigen::VectorXd &up) const
{
    Eigen::VectorXd uml(uc0.size());
    Eigen::VectorXd upl(uc0.size());

    // ANCSE CUT START TEMPLATE
    auto sm = uc0 - um0;
    auto sp = up0 - uc0;

    for (int i = 0; i < uc0.size(); i++)
    {
        uml(i) = limiter(um(i), sm(i), sp(i));
        upl(i) = limiter(up(i), sm(i), sp(i));
    }

    // ANCSE END TEMPLATE

34
return std::move(uml), std::move(upl));
}

**Hint:** You should test your implementation of DG limiting with Burgers’ equation. Run two Riemann problems, one that leads to a shock and another that leads to a rarefaction.

Repeat the Sod shock tube test described previously with your DG code.

**Solution:** The results of the Sod shock tube for DG schemes are given in Figure 6.
Figure 6: First- and second-order DG; density, velocity and pressure plots; $T = 0.2$. 
Exercise 2  Solving the compressible Euler equations on an unstructured mesh for computing flow past an airfoil

In this exercise, you will simulate the steady state airflow over a NACA airfoil. The simulation requires you to solve the Euler equations in two dimensions numerically on an unstructured grid.

**Hint:** Please read README.md carefully as it contains a lot of important information about the code.

We start with the following definitions:

- $t \in \mathbb{R}^+$ is the time,
- $x \in \mathbb{R}^2$ is the (Eulerian) position,
- $u(x, t) = (u_1(x, t), u_2(x, t)) \in \mathbb{R}^2$ is the velocity at $(x, t)$,
- $\rho(x, t) \in \mathbb{R}^+$ is the density at $(x, t)$,
- $p(x, t) \in \mathbb{R}$ is the pressure at $(x, t)$,
- $E(x, t) \in \mathbb{R}$ is the total energy at $(x, t)$.

The Euler equations are then given as:

\[
\begin{align*}
\rho_t + \nabla \cdot (\rho u) &= 0 \\
(\rho u)_t + \nabla \cdot (\rho u \otimes u + pI) &= 0 \\
E_t + \nabla \cdot ((E + p)u) &= 0.
\end{align*}
\]  

(21)

Here

\[
u \otimes u := \begin{pmatrix} u_1u_1 & u_1u_2 \\ u_2u_1 & u_2u_2 \end{pmatrix}
\]

The total energy of the system is given by

\[
E = \frac{1}{2} \rho u \cdot u + \rho e.
\]

For the rest of this exercise, we assume that the internal energy $e$ is given as

\[
e = \frac{p}{\gamma - 1}.
\]

37
where $\gamma$ is the adiabatic constant. We set

$$\gamma = 1.4.$$  

To make notation easier, we will consider the general form of (21) given as

$$U_t + \nabla_x \cdot F(U) = 0 \quad (22)$$

where $U : \mathbb{R}^d \times \mathbb{R}^+ \to \mathbb{R}^N$ is the vector of conserved variables, and $F : \mathbb{R}^N \to \mathbb{R}^{N \times d}$ is the flux tensor. We set

$$\nabla_x \cdot F(U(x,t)) := \left( \begin{array}{c} \frac{\partial}{\partial x_1} F_{1,1}(U(x,t)) + \ldots + \frac{\partial}{\partial x_d} F_{1,d}(U(x,t)) \\ \frac{\partial}{\partial x_1} F_{2,1}(U(x,t)) + \ldots + \frac{\partial}{\partial x_d} F_{2,d}(U(x,t)) \\ \vdots \\ \frac{\partial}{\partial x_1} F_{N,1}(U(x,t)) + \ldots + \frac{\partial}{\partial x_d} F_{N,d}(U(x,t)) \end{array} \right),$$

where

$$F(U) = \left( \begin{array}{c} F_1(U) \\ \vdots \\ F_N(U) \end{array} \right) = \left( \begin{array}{cccc} F_{1,1}(U) & F_{1,2}(U) & \ldots & F_{1,d}(U) \\ \vdots & \vdots & \ddots & \vdots \\ F_{N,1}(U) & F_{N,2}(U) & \ldots & F_{N,d}(U) \end{array} \right).$$

In the case of two dimensional Euler, $d = 2$ and $N = 4,$ and

$$U = \left( \begin{array}{c} \rho \\ \rho u_1 \\ \rho u_2 \\ E \end{array} \right) = : \left( \begin{array}{c} \rho \\ m_x \\ m_y \\ E \end{array} \right) \quad (23)$$

where we assume $\rho > 0.$

2a)

Write the compressible Euler equations (21) on the form (22). What is the flux function $F$ for the compressible Euler equations?

Solution: We set

$$F \left( \begin{array}{c} \rho \\ m_x \\ m_y \\ E \end{array} \right) = \left( \begin{array}{c} \frac{m_x}{\rho} + p \\ \frac{m_y}{\rho} + p \\ \frac{u_1}{\rho} + \frac{u_2}{\rho} \end{array} \right) \left( \begin{array}{cccc} m_x & m_y \\ \rho u_1 & \rho u_2 \\ \rho u_1 & \rho u_2 \end{array} \right) = \left( \begin{array}{cccc} m_x & m_y \\ \rho u_1 & \rho u_2 \\ \rho u_1 & \rho u_2 \end{array} \right) + \left( \begin{array}{cccc} m_x & m_y \\ \rho u_1 & \rho u_2 \\ \rho u_1 & \rho u_2 \end{array} \right). \quad (24)$$

We consider a polygonal domain $\Omega$ in $\mathbb{R}^2$ and assume we have a triangulation $T = \{K_i\}_{i=1}^M$. See Figure 7 for an illustration.
2b)

Let $1 \leq i \leq M$ and let $U : \Omega \times \mathbb{R}^{+} \rightarrow \mathbb{R}^{N}$, and define

$$U_i(t) := \frac{1}{|K|} \int_{K} U(x, t) \, dx.$$  

Furthermore, for $n \in \mathbb{R}^{2}$, we denote:

$$F(U) \cdot n := \begin{bmatrix} F_1(U) \cdot n \\ \vdots \\ F_N(U) \cdot n \end{bmatrix}.$$  

This represents the flux through a surface with normal vector $n$.

Given triangle $K_i \in \mathcal{T}$, and a neighbouring triangle $K_j$ (i.e. $K_i \cap K_j$ is a segment), we denote their common edge as $e_{i,j}$. For a given triangle $K_i$, we denote its three edges as $\{e_{i,j_k}\}_{k=1}^{3}$, with $K_{j_k}$ its three neighbours.

Remark: Technically this does not work if $K_i$ has one edge on $\partial \Omega$, since $K_{j_k}$ may not be a triangle in $\mathcal{T}$. However, in this case the notation is not useful because you do not want to treat all edges the same because boundary conditions apply to some of the edges. Therefore, either ignore the issue or set $K_{j_k} = \Omega^{c}$ (for the sole purpose of giving those edges a name).
Assume $U$ is a smooth solution of (22). Fix $t^n, \Delta t \geq 0$. Show that

$$\bar{U}_i(t^n + \Delta t) - \bar{U}_i(t^n) = -\frac{1}{|K_i|} \sum_{k=1}^{3} \int_{t^n}^{t^n+\Delta t} \int_{e_{i,jk}} F(U) \cdot n_{i,jk} dS dt,$$

(25)

where $\{e_{i,jk}\}_{k=1}^{3}$ are the edges of $K_i$, and $\{n_{i,jk}\}_{k=1}^{3}$ are the outward-pointing unit vectors normal to the triangle along edge $e_{i,jk}$.

Solution: Let us start from equation (22) in $K_i$:

$$U_t + \nabla_x \cdot F(U) = 0$$

(26)

We integrate in space (for domain $K_i$) and in time (for $t \in [t^n, t^n + \Delta t]$), and divide by the area of $K_i$, $|K_i|$. Since $U$ is smooth on $K_i$ compact, we have integrability and thus

$$0 = \frac{1}{|K_i|} \int_{K_i} \int_{t^n}^{t^n+\Delta t} \frac{\partial}{\partial t} U(x,t) dt \, dx + \frac{1}{|K_i|} \int_{K_i} \int_{t^n}^{t^n+\Delta t} \nabla_x \cdot F(U(x,t)) dt \, dx$$

(27)

$$= \frac{1}{|K_i|} \int_{K_i} \int_{t^n}^{t^n+\Delta t} \left( U(x,t^n + \Delta t) - U(x,t^n) \right) \, dx + \frac{1}{|K_i|} \int_{K_i} \int_{t^n}^{t^n+\Delta t} \nabla_x \cdot F(U(x,t)) \, dx \, dt$$

(28)

$$= \bar{U}_i(t^n + \Delta t) - \bar{U}_i(t^n) + \frac{1}{|K_i|} \int_{t^n}^{t^n+\Delta t} \int_{\partial K_i} F(U(x,t)) \cdot n \, dS dt$$

(29)

where in the last step, we have used the definition of $\bar{U}$, Gauss’ theorem, and $n$ is the outward-pointing unit vector normal to $\partial K_i$.

We conclude by noting that, since $K_i$ is a triangle, its boundary can be written as the union of its three edges $\partial K_i = \bigcup_{k=1}^{3} e_{i,jk}$, along each of which $n \equiv n_{i,jk}$. Since this union is disjoint (other than sets of measure 0), we can conclude

$$0 = \bar{U}_i(t^n + \Delta t) - \bar{U}_i(t^n) + \frac{1}{|K_i|} \sum_{k=1}^{3} \int_{t^n}^{t^n+\Delta t} \int_{e_{i,jk}} F(U) \cdot n_{i,jk} dS dt$$

\[\square\]

2c)

Fix an edge with unit outward normal $n$ and a transverse unit vector $\tau$ such that $n \cdot \tau = 0$. Verify that

$$F(U) \cdot n = \begin{pmatrix} \rho(u \cdot n) \\ \rho \rho(u \cdot n) u_1 + p \ n_1 \\ \rho \rho(u \cdot n) u_2 + p \ n_2 \\ (u \cdot n)(E + p) \end{pmatrix}.$$
This looks remarkably similar to any one of the two components of the flux tensor. But not close enough to allow us to use the approximate Riemann solvers we know from structured grids.

However, for a fixed edge we could choose to perform coordinate transform $x \mapsto (\xi, \zeta)$ such that $x = \xi u + \zeta \tau$. Define $u_\perp = u \cdot n$ and $u_\parallel = u \cdot \tau$, then the resulting system of PDEs is

\begin{align*}
\partial_t \rho + \partial_\xi \rho u_\perp + \partial_\zeta \rho u_\parallel &= 0 \\
\partial_t \rho u_\perp + \partial_\xi (\rho u_\perp^2 + p) + \partial_\zeta \rho u_\parallel u_\perp &= 0 \\
\partial_t \rho u_\parallel + \partial_\xi \rho u_\perp u_\parallel + \partial_\zeta (\rho u_\parallel^2 + p) &= 0 \\
\partial_t E + \partial_\xi u_\perp (E + p) + \partial_\zeta u_\parallel (E + p) &= 0
\end{align*}

Note that we choose to compute the rate of change of the density, normal component of the momentum, the transverse component of the momentum and the energy. Furthermore, note that flux in $\xi$-direction is

$$f(U) = \begin{pmatrix} \rho u_\perp \\ \rho u_\perp^2 + p \\ \rho u_\perp u_\parallel \\ u_\perp (E + p) \end{pmatrix}.$$  

Use these ideas to approximate the contribution of edge $e$

$$\int_{t}^{t+\Delta t} \int_{e_{i,j}} F(U) \cdot n_{i,j} \, dS \, dt.$$  

to the net flux by the HLLC approximate Riemann solver known from two-dimensional FVM on structured grids.

Remember, the approximate Riemann solver will compute the flux of momentum normal and transverse to the interface. You need to convert this into the flux of momentum in $x$- and $y$-directions.

**Hint:** You can take a shortcut and approximate with Rusanov’s numerical flux instead.

**Solution:** Let $T : \mathbb{R}^4 \to \mathbb{R}^4$ be the transform which map the momentum components into normal and tangential components; and $T^{-1}$ its inverse, i.e.

$$T(q) = (q_1, q_2 n_1 + q_3 n_2, q_2 \tau_1 + q_3 \tau_2, q_4)$$
$$T^{-1}(q) = (q_1, q_2 n_1 + q_3 \tau_1, q_2 n_2 + q_3 \tau_2, q_4)$$

In particular $T(u) = (\rho, \rho u_\perp, \rho u_\parallel, E)$.

Next observation is that in the rotate coordinate system the interface is perpendicular to the $\xi$-axis. Hence in the rotated system only $f$ needs to be considered.
Final observation is that the rotated system updates the normal and transverse momentum (which is different for each interface). However, we would like to work in the global \((x, y)\)-coordinate system.

If we put these ideas together we get:

\[
\int_{e_{i,j}} F(U) \cdot n_{i,j} dS = \int_{e_{i,j}} T^{-1} f(T(U)) dS \approx |e_{i,j}| T^{-1} F^{HLLC}(T(U))
\]  

(39)

Note, that the HLLC flux can be replaced by any other approximate flux.

The corresponding code to compute the HLLC flux is

**Listing 19:** The HLLC numerical flux.

```cpp
auto xvarL = std::pair{euler::pressure(uL), euler::speedOfSound(uL)};
auto xvarR = std::pair{euler::pressure(uR), euler::speedOfSound(uR)};

auto [sL, s_star, sR] = batten_speeds(uL, xvarL, uR, xvarR);

// HLLC flux
const EulerState &uK = (0.0 <= s_star ? uL : uR);
double pK = (0.0 <= s_star ? xvarL.first : xvarR.first);
auto nf = euler::flux(uK);

if (sL < 0.0 && 0.0 <= sR) {
    double sK = (0.0 <= s_star ? sL : sR);
    double vK = (0.0 <= s_star ? uL(1) / uL(0) : uR(1) / uR(0));
    double cK = (sK - vK) / (sK - s_star);

    nf(0) += sK * (cK * uK(0) - uK(0));
    nf(1) += sK * (cK * uK(0) * s_star - uK(1));
    nf(2) += sK * (cK * uK(2) - uK(2));
    nf(3) += sK * (cK
        * (uK(3)
        + (s_star - vK) * (uK(0) * s_star + pK / (sK - vK)))
        - uK(3));
}

return nf;
```

Which requires estimates of the wave speeds:

**Listing 20:** Estimates of the wave-speeds by Einfeldt-Batten.

```cpp
static std::tuple<double, double, double>
batten_speeds(const EulerState &uL,
```
const std::pair<double, double> &xvarL,
const EulerState &uR,
const std::pair<double, double> &xvarR) {

RoeAverage roe_average(uL(0), uR(0));

auto [pL, aL] = xvarL;
auto [pR, aR] = xvarR;

double vL = uL(1) / uL(0);
double vR = uR(1) / uR(0);
double v_tilda = roe_average(vL, vR);

double HL = (uL(3) + pL) / uL(0);
double HR = (uR(3) + pR) / uR(0);
double H_tilda = roe_average(HL, HR);

double vroe_square = square(roe_average(uL(1) / uL(0), uR(1) / uR(0)))
+ square(roe_average(uL(2) / uL(0), uR(2) / uR(0)));

double a_tilda = std::sqrt((GAMMA - 1.0) * (H_tilda - 0.5 * vroe_square));

double sL = std::min(vL - aL, v_tilda - a_tilda);
double sR = std::max(vR + aR, v_tilda + a_tilda);

// Given the velocities of the left- and right-most waves this
// is the choice of the middle wave speed.
//
// double s_star = (uR(1) * (sR - vR) - uL(1) * (sL - vL) + pL - pR)
// / (uR(0) * (sR - vR) - uL(0) * (sL - vL));

return {sL, s_star, sR};
}

Finally the Roe average can be computed as follows

Listing 21: A class to compute Roe averages.

/// The Roe average of two states.
class RoeAverage {
public:
    /// Construct a RoeAverage w.r.t. the densities ‘rho_L‘ and ‘rho_R‘.
    RoeAverage(double rho_L, double rho_R)
        : roe_ratio(std::sqrt(rho_R / rho_L)) {}

    /// Compute the Roe average.
    double operator()(double qL, double qR) {

43
\[
\text{return } (qL + qR \times \text{roe\_ratio}) / (1.0 + \text{roe\_ratio}) \;
\]

private:
  double roe\_ratio;
};

2d)

For \( f : \mathbb{R}^N \rightarrow \mathbb{R}^N \), let \( \{\lambda_i(f, U)\}_{i=1}^N \) be the eigenvalues of the Jacobian of \( f(U) \). Define
\[
\lambda_{\text{max}}(f, U) := \max_i |\lambda_i(f, U)|.
\]

Show that for flux \( F(U) \cdot n \), it holds that:
\[
\lambda_{\text{max}}(F \cdot n, U) = |u \cdot n| + \sqrt{\gamma p \rho}. \tag{40}
\]

Use this to derive the following CFL-condition:
\[
\Delta t < C_{\text{CFL}} \frac{\Delta x}{\max_i \lambda_{\text{max}}(F \cdot n, U_i)}, \tag{41}
\]

for all unit-vectors \( n \). Here \( \Delta x \) is the small inradius and \( C_{\text{CFL}} = 0.45 \).

Hint: Use the trick in subproblem 2c) to reduce the problem to computing the eigenvalues of the Jacobian of \( F_{1,1} \). Which you have either seen in the lecture or you can look them up online.

Solution: The eigenvalues of the Jacobian of \( F_{1,1} \) are \((u-c, u, u, u+c)\), with \( c \) the speed of sound.

Due to the rotation-invariance that we have seen in the previous task, the eigenvalues must coincide with the eigenvalues of the rotated flux \( f \), which will then be:
\[
(\lambda_1, \lambda_2, \lambda_3, \lambda_4) = (u_\perp - c, u_\perp, u_\perp, u_\perp + c) \tag{42}
\]

And thus the maximum eigenvalue in absolute value is
\[
\begin{cases} 
\lambda_1 = u \cdot n - c & \text{if } u \cdot n < 0 \\
\lambda_4 = u \cdot n + c & \text{if } u \cdot n \geq 0
\end{cases}
\]

and minding that \( c > 0 \), it is clear that in both cases,
\[
|\lambda_{\text{max}}| = |u \cdot n| + c
\]

If we take the maximum of \( |\lambda_{\text{max}}| \) over all directions \( n \) we get
\[
\sup_{n: |n| = 1} |u \cdot n| + c = |u| + c \tag{43}
\]
The geometric idea behind the CFL condition is that the numerical speed of propagation \( \Delta x / \Delta t \) must exceed the physical speed of propagation, i.e. \( |u| + c \). Some freedom exists in choosing the length scale \( \Delta x \), here we conservatively choose the minimum inradius.

Therefore,

\[
\Delta t \leq CFL \frac{\Delta x}{|u| + c}
\]

is one possible CFL condition.

The CFL condition is implemented as follows.


```
double operator()(const Eigen::MatrixXd &U) const {
    // ANCSE_COMMENT compute the cfl condition here.
    // ANCSE_COMMENT you can use 'assert_valid_timestep' to check if
    // ANCSE_COMMENT the computed value is valid.
    double max_ev = 0.0;

    int n_cells = U.rows();
    #pragma omp parallel for
    for (int i = 0; i < n_cells; ++i) {
        max_ev = std::max(max_ev, euler::maxEigenValue(U.row(i)));
    }

    double dt_cfl = cfl_number * dx / max_ev;
    assert_valid_timestep(dt_cfl);

    return dt_cfl;
    // ANCSE_RETURN_VALUE 0.001;
    // ANCSE_END_TEMPLATE
}
```

2e)

Implement two types of flux boundary conditions:

1. outflow flux boundary conditions
2. reflective (or solid wall) flux boundary conditions.
Flux boundary conditions specify a flux at the boundary of the domain. Therefore, they do not need any ghost-cells. Instead one needs to identify all interfaces at the boundary of the domain. For these faces one must not attempt to compute the numerical flux, instead one evaluates a flux boundary condition.

Consider edge $e$ of triangle $K$. The outflow boundary conditions are given by

$$\int_{t}^{t+\Delta t} \int_{e} F(U) \cdot n \, dS \, dt = \Delta t |e| F(U) \cdot n.$$  \hfill (45)

The reflective (or wall) flux boundary conditions are

$$\int_{t}^{t+\Delta t} \int_{e} F(U) \cdot n \, dS \, dt = \Delta t F(U, U^*)$$  \hfill (46)

where $U = (\rho, \rho u, E)$ are the cell-averages of the conserved variables in triangle $K$ and

$$U^* = (\rho, -\rho (u \cdot n) n + \rho (u \cdot \tau) \tau, E).$$  \hfill (47)

Here $F$ denotes the numerical flux derived in 2c).

**Hint:** The class `Mesh` has a member `getBoundaryType` which tells you which type of boundary condition must be applied to a given edge.

**Solution:** The two boundary conditions can be implemented as follows.

**Listing 23:** Implementation of the outflow boundary conditions.

```cpp
EulerState computeOutflowFlux(const Eigen::MatrixXd &U, int i, int k, const Mesh &mesh) const {
    auto normal = mesh.getUnitNormal(i, k);
    auto f = euler::flux(euler::localCoordinates(U.row(i), normal));
    return euler::globalCoordinates(f, normal);
}
```

**Listing 24:** Implementation of the reflective boundary conditions.
With the result you’ve derived so far, implement first order FVM for the Euler equations on an unstructured grid. Carefully debug your code at this stage, before going on to the second order FVM. Check convergence rates using the script `compute_convergence.py` and visualize the solution with `plot_on_mesh.py`.

**Hint:** See README.md for more details.
The final step in implementing the second order FVM is to compute the trace values of $U$ at the interfaces.

Analogous to the REA algorithm we will compute the reconstruction of a quantity $q$, such as $\rho$, $u_1$ or $p$, whose value $\{Q_i\}_i$ is given at the cell-centers $x_i$. We do this with:

$$q_i(x) = Q_i + (\nabla q(x_i)) \cdot (x - x_i).$$  \hspace{1em} (48)

There are two problems. The first one is that this corresponds to piecewise linear reconstruction without a slope-limiter. This was not stable for 1D problems and won’t be stable in this context. The second problem is how to compute the gradient $\nabla q$.

2g)  

Approximate the right hand side of

$$\nabla q(x_i) \approx \frac{1}{|K_i|} \int_{K_i} \nabla q \, dx$$  \hspace{1em} (49)

by Gauss’ theorem (applied to the vector field $q_c$, for certain constant $c \in \mathbb{R}^2$), and the mid-point rule for the resulting integrals over the boundary.

**Solution:** Let $c \in \mathbb{R}^2$ a constant vector, and let us apply the divergence theorem to the function $G := qc$:

$$\int_{K_i} \nabla \cdot G \, dx = \int_{\partial K_i} G \cdot n \, dS$$  \hspace{1em} (50)

$$\iff \int_{K_i} (\nabla q \cdot c) \, dx = \int_{\partial K_i} (c \cdot n)q \, dS$$  \hspace{1em} (51)

$$= \sum_{k=1}^{3} \int_{e_{i,j_k}} (c \cdot n_{i,j_k})q \, dS$$  \hspace{1em} (52)

$$\approx \sum_{k=1}^{3} |e_{i,j_k}|(c \cdot n_{i,j_k})q(x_{i,j_k})$$  \hspace{1em} (53)

where in the last step we have used the mid-point rule for each edge $e_{i,j_k}$; we denote the mid-point of each such edge by $x_{i,j_k}$. We denote by $|e_{i,j_k}|$ the length of the edge. Applying the argument above for all $c \in \{(1,0), (0,1)\}$, we obtain (component-by-component):

$$\int_{K_i} \nabla q \, dx = \sum_{k=1}^{3} |e_{i,j_k}|n_{i,j_k} q(x_{i,j_k})$$
We can implement the proposed approximate as follows. Note, that this is the same for both primitive and conserved variables. If \( \bar{q} \) are primitive variables, then the gradients are the gradients of the primitive variables; and if the input is conserved variables the output is in conserved variables.

**Listing 25**: Implementation of the outflow boundary conditions.

```cpp
/// Compute the gradients of \( \bar{q} \).
/** Note: this routine does not care if \( q \) are the conserved
 * or primitive variables.
 */
@param [out] dqdx approximation of \( dq/dx \). Has shape (n_cells, 4).
@param [out] dqdy approximation of \( dq/dy \). Has shape (n_cells, 4).
@param q_bar the cell-averages of ‘\( q \)’. Has shape (n_cells, 4).
@param mesh
*/
void compute_gradients(Eigen::MatrixXd &dqdx,
                      Eigen::MatrixXd &dqdy,
                      const Eigen::MatrixXd &q_bar,
                      const Mesh &mesh) {
    // ANCSE_CUT_START_TEMPLATE
    // ANCSE_COMMENT Compute the gradient of all 4 components of \( \bar{q} \)
    dqdx.setZero();
    dqdy.setZero();
    int n_cells = mesh.getNumberOfTriangles();
    #pragma omp parallel for
    for (int i = 0; i < n_cells; ++i) {
        double area = mesh.getTriangleArea(i);
        // w.l.o.g. there are enough neighbours.
    }
    int j = mesh.getNeighbour(i, k);
    Eigen::Vector2d xi = mesh.getCellCenter(i);
}
```
Fix $i \in \{1, \ldots, M\}$. For each edge $e_{i,j}$ of triangle $K_i$, shared with triangle $K_j$, let us denote its mid-point by $x_{i,j}$. One can compute a limited slope as follows

$$s_{i,j} = \xi(\nabla q(x_i) \cdot \Delta x_{i,j}, \nabla q(x_j) \cdot \Delta x_{i,j})$$

(54)

with $\Delta x_{i,j} = x_{i,j} - x_i$ and a slope limiter $\xi$ of your choice. The reconstructed values are then

$$q_{i,j} = Q_i + s_{i,j} \quad \text{and} \quad q_{j,i} = Q_j + s_{j,i}.$$  

(55)

Implement piecewise linear reconstruction of the primitive variables $(\rho, u_1, u_2, p)$ by using the previously derived expressions.

Solution: The following snippet shows how to compute the flux through an interface with piecewise linear reconstruction of the trace values.

Listing 26: Implementation of the flux through an interface between two cells in the interior.

```cpp
EulerState computeInteriorFlux(const Eigen::MatrixXd &U, int i, int k, const Mesh &mesh) const {
    // ANCSE_CUT_START_TEMPLATE
    int j = mesh.getNeighbour(i, k);
    // ANCSE_COMMENT Reconstruct the trace values of U and compute
    // ANCSE_COMMENT the numerical flux through the k-th interface of
    // ANCSE_COMMENT cell i.
    // ANCSE_END_TEMPLATE
    return ...
}
```
auto normal = mesh.getUnitNormal(i, k);

auto xi = mesh.getCellCenter(i);
auto xj = mesh.getCellCenter(j);
auto x_ij = mesh.getEdgeCenter(i, k);

auto grad_wi = getGradient(i);
auto grad_wj = getGradient(j);

EulerState uL, uR;
EulerState wL, wR;

for (int p = 0; p < 4; ++p) {
    auto sL = limited_slope(grad_wi.row(p), grad_wj.row(p), x_ij - xi);
    auto sR = limited_slope(grad_wi.row(p), grad_wj.row(p), x_ij - xj);

    wL(p) = pvars(i, p) + sL;
    wR(p) = pvars(j, p) + sR;
}

uL = euler::conservedVars(wL);
uR = euler::conservedVars(wR);

auto nf = hllc(euler::localCoordinates(uL, normal),
               euler::localCoordinates(uR, normal));

return euler::globalCoordinates(nf, normal);

//// ANCSE_RETURN_VALUE EulerState{}
//// ANCSE_END_TEMPLATE

2i)
First run a convergence test for your code. We do this by running a smooth vortex test case on a sequence of meshes.

The numerical experiment has already been implemented in vortex.cpp.

Hint: You can find more information in README.md.
2j)

Simulate and visualize the steady state airflow over an airfoil. Try different Mach numbers and angles of attack, by modifying the appropriate lines in naca_airfoil.cpp.

**Hint:** You will find more information in README.md.