Series 2

ETH zürich

Advanced Numerical Methods for CSE
Last edited: December 7, 2021
Due date: Jenuray 8, 2022

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 Finite volume and discontinuous Galerkin methods for Euler equations in 1D

Euler equations in 1D are written as:

$$\mathbf{u}_t + \mathbf{f}(\mathbf{u})_x = \mathbf{0}, \tag{1a}$$

$$\mathbf{u} = \begin{pmatrix} \rho \\ \rho v \\ E \end{pmatrix}, \quad \mathbf{f}(\mathbf{u}) = \begin{pmatrix} \rho v \\ \rho v^2 + p \\ (E+p)v \end{pmatrix}, \tag{1b}$$

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

$$E = \frac{p}{\gamma - 1} + \frac{1}{2}\rho v^2, \text{ for } \gamma = 7/5 = 1.4. \tag{2}$$

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

$$c = \sqrt{\frac{\gamma p}{\rho}}, \quad H = \frac{E+p}{\rho}. \tag{3}$$

In this exercise, we implement finite volume and discontinuous Galerkin methods for the 1D Euler equations (1) on a domain Ω discretized by a uniform grid $\mathcal{T}_h$, 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 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.

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.

Part I. Finite volume method

The semi-discrete formulation of FVM is

$$\frac{d}{dt}\mathbf{U}_i + \frac{1}{\Delta x}\left(\mathbf{F}_{i+1/2} - \mathbf{F}_{i-1/2}\right) = 0 \tag{4}$$

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

$$\mathbf{F}_{i+1/2} = \mathbf{F}(\mathbf{U}^-_{i+1/2}, \mathbf{U}^+_{i+1/2}), \tag{5}$$

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.

```
inline std::tuple<double, double, double>
primitive(const Eigen::VectorXd &u_cons) const
{

    /// ANCSE_COMMENT Convert conservative to primitive;
    /// ANCSE_COMMENT double rho=0.;
    /// ANCSE_COMMENT double v=0.;
    /// ANCSE_COMMENT double p=0.;
    /// ANCSE_COMMENT return std::make_tuple (rho, v, p);


    //// ANCSE_CUT_START_TEMPLATE
    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);
    //// ANCSE_END_TEMPLATE
}

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;
}

inline double energy(double rho, double v, double p) const
{
    return (p/(gamma-1) + 0.5*rho*v*v);
}

inline Eigen::VectorXd eigenvalues(double v, double c) const
{

    Eigen::VectorXd eigvals(n_vars);
    /// ANCSE_COMMENT Compute eigenvalues
    //// ANCSE_CUT_START_TEMPLATE
    eigvals << v - c, v, v + c;
    //// ANCSE_END_TEMPLATE
    return eigvals;
}

inline Eigen::MatrixXd eigenvectors(double v, double H) const
{
    Eigen::MatrixXd eigvecs(n_vars, n_vars);
    /// ANCSE_COMMENT Compute eigenvectors
    //// ANCSE_CUT_START_TEMPLATE
    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;
    //// ANCSE_END_TEMPLATE

    return eigvecs;
}
```

Now, we iplement the routines provided through the interface class.

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

```
Eigen::VectorXd Euler::flux(const Eigen::VectorXd &u) const
{
    //// ANCSE_CUT_START_TEMPLATE
    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;
    //// ANCSE_END_TEMPLATE
    //// ANCSE_RETURN_VALUE Eigen::VectorXd::Zero(n_vars);
}

Eigen::VectorXd Euler::eigenvalues(const Eigen::VectorXd &u) const
{
    //// ANCSE_CUT_START_TEMPLATE
    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;
    //// ANCSE_END_TEMPLATE
    //// ANCSE_RETURN_VALUE Eigen::VectorXd::Zero(n_vars);
}

Eigen::MatrixXd Euler::eigenvectors(const Eigen::VectorXd &u) const
{
    //// ANCSE_CUT_START_TEMPLATE
    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);
    //// ANCSE_END_TEMPLATE
    //// ANCSE_RETURN_VALUE Eigen::MatrixXd::Zero(n_vars, n_vars);
}

double Euler::max_eigenvalue(const Eigen::VectorXd &u) const
{
    //// ANCSE_CUT_START_TEMPLATE
    return (eigenvalues(u).cwiseAbs()).maxCoeff();
    //// ANCSE_END_TEMPLATE
    //// ANCSE_RETURN_VALUE 0;
}


Eigen::VectorXd Euler::cons_to_prim(const Eigen::VectorXd &u_cons) const
{
    //// ANCSE_CUT_START_TEMPLATE
    double rho, v, p;
    std::tie (rho, v, p) = primitive(u_cons);

    Eigen::VectorXd u_prim(n_vars);
    u_prim << rho, v, p;

    return u_prim;
    //// ANCSE_END_TEMPLATE
    //// ANCSE_RETURN_VALUE Eigen::VectorXd::Zero(n_vars);
}

Eigen::VectorXd Euler::prim_to_cons(const Eigen::VectorXd &u_prim) const
{
    //// ANCSE_CUT_START_TEMPLATE
    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;
    //// ANCSE_END_TEMPLATE
    //// ANCSE_RETURN_VALUE Eigen::VectorXd::Zero(n_vars);
}

Eigen::VectorXd Euler::roe_avg(const Eigen::VectorXd &uL,
                               const Eigen::VectorXd &uR) const
{
    // left state
    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);

    // right state
    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);

    // Roe state
    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;
}
```

1b)

Template: include/ancse/numerical_flux.hpp

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.

```
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
    {
        //// ANCSE_CUT_START_TEMPLATE
        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));
        //// ANCSE_END_TEMPLATE
        //// ANCSE_RETURN_VALUE Eigen::VectorXd();
    }

  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 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.

Lax Friedrichs flux can be implemented as follows.

Listing 4: Implementation of Lax-Friedrichs flux.

```
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;
        //// 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_VALUE Eigen::VectorXd();
    }

  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
    {
        //// ANCSE_CUT_START_TEMPLATE
        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));

        //// ANCSE_END_TEMPLATE
        //// ANCSE_RETURN_VALUE Eigen::VectorXd();
    }
  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
    {
        //// ANCSE_CUT_START_TEMPLATE
        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());

        // 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;
        //// ANCSE_END_TEMPLATE
        //// ANCSE_RETURN_VALUE Eigen::VectorXd();
    }

  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
    {
        //// ANCSE_CUT_START_TEMPLATE
        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;
        //// ANCSE_END_TEMPLATE
        //// ANCSE_RETURN_VALUE Eigen::VectorXd();
    }

  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 $\mathbf{U}^{\pm}_{i+1/2}$ based on

- conservative variables $\mathbf{u}_{\text{cons}} = (\rho, \rho v, E)$
- primitive variables $\mathbf{u}_{\text{prim}} = (\rho, v, p)$

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

Listing 8: Implementation of different slope limiters.

```
struct MinMod {
    inline double operator()(double a, double b) const
    {
        //// ANCSE_CUT_START_TEMPLATE
        return minmod(a, b);
        //// ANCSE_END_TEMPLATE
        //// ANCSE_RETURN_VALUE 0.;
    }
};

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

        return maxmod(A, B);
        //// ANCSE_END_TEMPLATE
        //// ANCSE_RETURN_VALUE 0.;
    }
};

struct MonotonizedCentral {
    inline double operator()(double sL, double sR) const {
        //// ANCSE_CUT_START_TEMPLATE
        return minmod(2.0 * sL, 0.5 * (sL + sR), 2.0 * sR);
        //// ANCSE_END_TEMPLATE
        //// ANCSE_RETURN_VALUE 0.;
    }
};
```

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.

```
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
    {

        /// ANCSE_COMMENT Implement here the reconstruction using conservative
            ↪ variables.

        //// ANCSE_CUT_START_TEMPLATE
        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)};

        //// ANCSE_END_TEMPLATE
        //// ANCSE_RETURN_VALUE {std::move(Eigen::VectorXd()), std::move(Eigen::
            ↪ VectorXd())};


    }

  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) {
            /// ANCSE_COMMENT Implement here the transformation from conservative
                ↪ to primitive.
            //// ANCSE_CUT_START_TEMPLATE
            up.col(i) = model->cons_to_prim(u.col(i));
            //// ANCSE_END_TEMPLATE
        }
    }

    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
    {

        /// ANCSE_COMMENT Implement here the reconstruction using primitive
            ↪ variables.
        /// ANCSE_COMMENT The transformation can be done above.


        //// ANCSE_CUT_START_TEMPLATE

        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)};
        //// ANCSE_END_TEMPLATE
        //// ANCSE_RETURN_VALUE {std::move(Eigen::VectorXd()), std::move(Eigen::
            ↪ VectorXd())};

    }

  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.

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

        //// 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);
        //// ANCSE_COMMENT implement the flux loop here.
        //// ANCSE_CUT_START_TEMPLATE
        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
    }
```

1e)

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

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

$$u^* = u_0 + \Delta t L(u^n) \tag{6}$$
$$u^{**} = u^* + \Delta t L(u^*) \tag{7}$$
$$u^{n+1} = 1/2(u^n + u^{**}). \tag{8}$$

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.

```
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
    {
        //// ANCSE_COMMENT implement RK here.
        //// ANCSE_CUT_START_TEMPLATE
        (*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);
        //// ANCSE_END_TEMPLATE
    }

  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

$$\Delta t \leq C_{CFL} \frac{\Delta x}{a_{max}}, \quad a_{max} = \max_i \mathbf{f}'(\mathbf{U}_i), \tag{9}$$

with $C_{CFL} < 1$.

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.

```
StandardCFLCondition <FVM>
:: StandardCFLCondition(const Grid &grid,
                        const std::shared_ptr<Model> &model,
                        double cfl_number)
    : grid(grid), model(model), cfl_number(cfl_number) {}

double StandardCFLCondition <FVM>
:: operator()(const Eigen::MatrixXd &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.col(i)));
    }

    return cfl_number * grid.dx / a_max;
    //// ANCSE_END_TEMPLATE
    //// ANCSE_RETURN_VALUE 0.;

}
```

1g)

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

$$\begin{pmatrix} \rho_L \\ v_L \\ p_L \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \quad \begin{pmatrix} \rho_R \\ v_R \\ p_R \end{pmatrix} = \begin{pmatrix} 0.125 \\ 0 \\ 0.1 \end{pmatrix}. \tag{10}$$

  For details about this experiment, you can read:

  - G. Sod, A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws, J. Comput. Phys., 27 (1978), pp. 1–31.
    URL: `https://www.archives-ouvertes.fr/hal-01635155/document`

– 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.

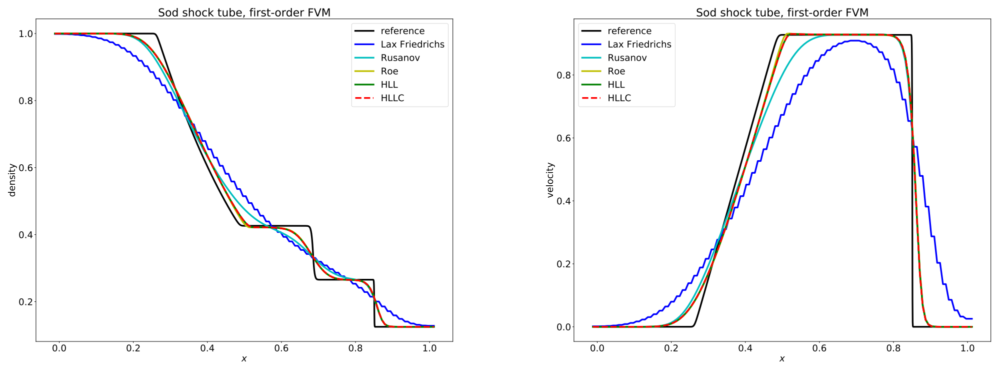

Figure 1: Comparison of fluxes for first-order FVM; density and velocity plots; $T = 0.2$.

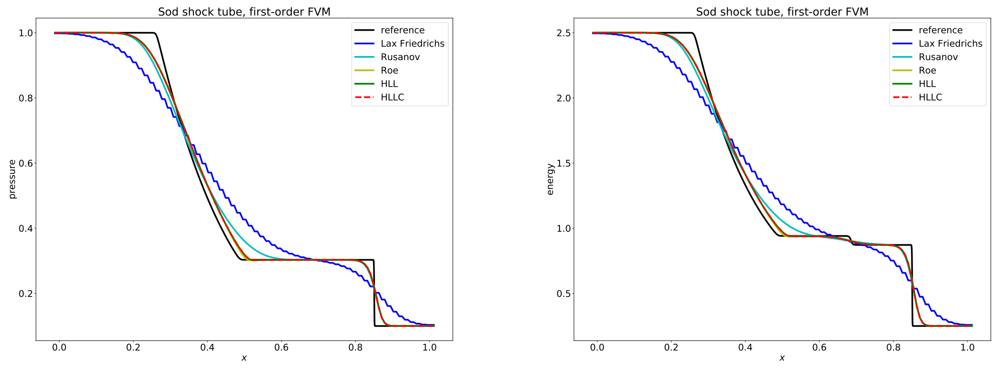

Figure 2: Comparison of fluxes for first-order FVM; pressure and energy plots; $T = 0.2$.

The comparison for different limiters is given in Figure 3.

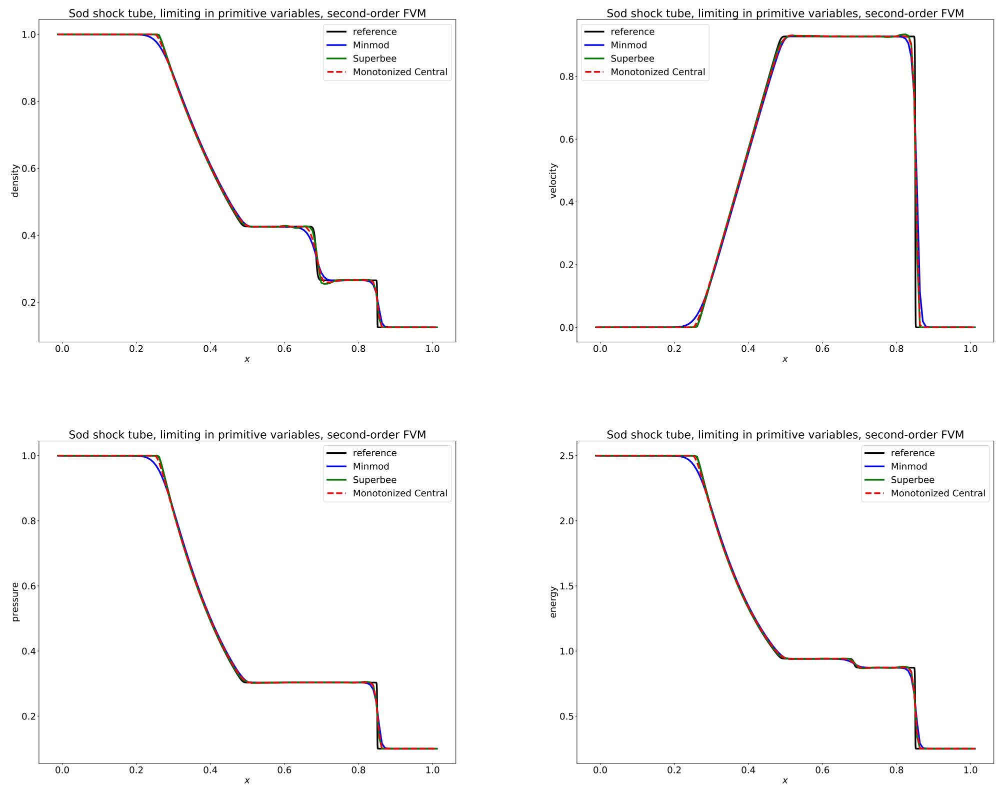

Figure 3: Comparison of limiters for second-order FVM with limiting in primitive variables; density, velocity, pressure and energy plots; $T = 0.2$.

The superbee limiter is compared for limiting in conserved and primitive variables in Figure 4.

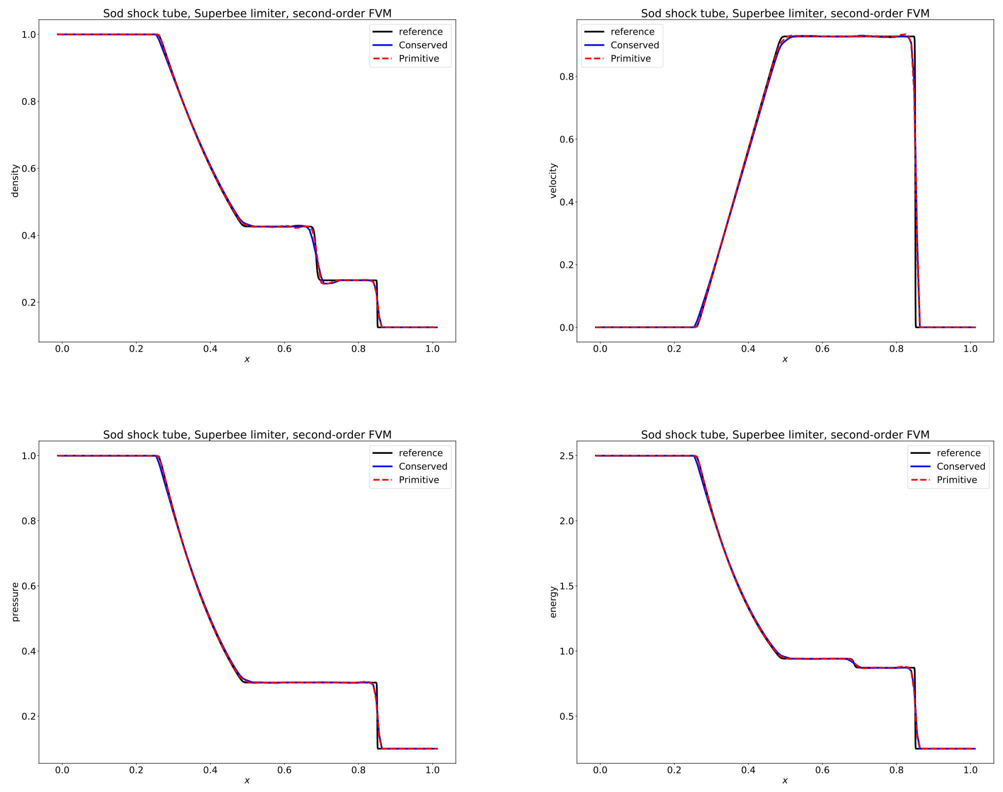

Figure 4: Comparison of limiting in conserved versus primitive variables for second-order FVM using superbee limiter; density, velocity, pressure and energy plots; $T = 0.2$.

- 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

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

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 $x = 0.5$.

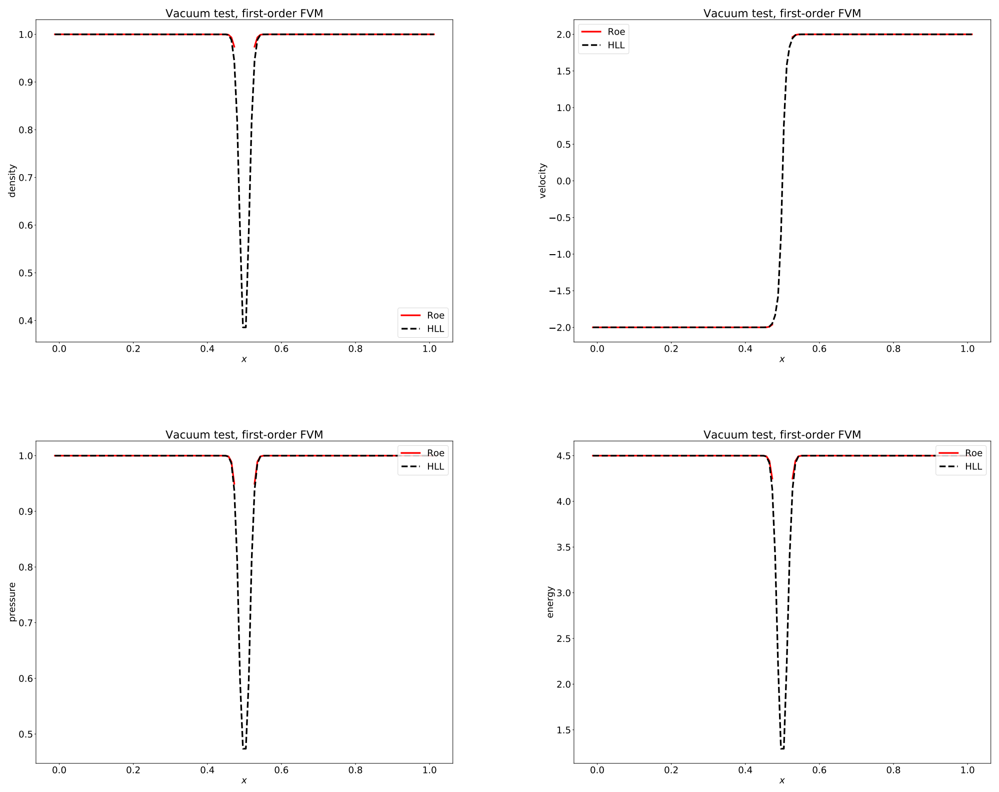

Figure 5: Results for the "vacuum" test with Roe's flux; density, velocity, pressure and energy plots; $T = 0.005$.

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.

Part II. Discontinuous Galerkin method (Optional)

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

$$W_h^p := \{w \in L^2(\Omega) : w|_K \in \mathbb{P}_p(K),\ K \in \mathcal{T}_h\}. \tag{12}$$

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

$$P_0(x)=1, \quad P_1(x)=x, \quad P_2(x)=\frac{1}{2}(3x^2-1).$$

The Legendre polynomials are orthogonal:

$$\int_{-1}^{+1} P_k \; P_l \; dx = \frac{2}{2k+1}\delta_{kl},$$

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

$$\phi_k(x) := \sqrt{2k+1}\; P_k(2x-1)\; \chi_{[0,1]}(x), \text{ such that } \int_0^{+1} \phi_k \; \phi_l \; dx = \delta_{kl}.$$

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_h^p, i.e. a function $w \in W_h^p$ 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

$$\mathbf{W}_h^p := [W_h^p]^m, \tag{14}$$

i.e. for a vector function $\mathbf{w} \in \mathbf{W}_h^p$, its components $w_l \in W_h^p$ for $l=1,\ldots,m$. For 1D Euler equations $m=3$.

We seek a solution of the form

$$\mathbf{u}_h(x,t) = \sum_{i=0}^{N}\sum_{j=0}^{p} U_{i,j}(t) \; \varphi_{i,j}(x),$$

with $U_{i,j}:\mathbb{R}\to\mathbb{R}^m$ the coefficient of $\varphi_{i,j}$ in the expansion of $\mathbf{u}_h(\cdot,t)$. Note that, for fixed $t \geq 0$, $\mathbf{u}_h(\cdot,t)$ can be uniquely identified with a vector $(U_{i,j,l})_{l=1,j=0,i=1}^{l=m,j=p,i=N}$ 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 $\mathbf{w}_h \in \mathbf{W}_h^p$, integrate over an arbitrary cell $K_i := (x_{i-1/2},\; x_{i+1/2}) \in \mathcal{T}_h$ and use integration by parts:

$$\begin{aligned}
\int_{K_i} (\mathbf{u}_h)_t \cdot \mathbf{w}_h \; dx + \int_{K_i} \mathbf{f}(\mathbf{u}_h)_x \cdot \mathbf{w}_h \; dx &= 0 \\
\frac{d}{dt}\left(\int_{K_i} \mathbf{u}_h \cdot \mathbf{w}_h \; dx\right) + (\mathbf{f}(\mathbf{u}) \cdot \mathbf{w}_h)\big|_{x^-_{i+1/2}} - (\mathbf{f}(\mathbf{u}) \cdot \mathbf{w}_h)\big|_{x^+_{i-1/2}} - \int_{K_i} \mathbf{f}(\mathbf{u}_h) \cdot (\mathbf{w}_h)_x \; dx &= 0.
\end{aligned}$$

Here $\mathbf{u}_h \in \mathbf{W}_h^p$ is the approximate solution and the flux through the interface $i + 1/2$, $\mathbf{f}(\mathbf{u}(x_{i+\frac{1}{2}}))$, can be approximated with a numerical flux $\mathbf{F}$ by

$$\mathbf{f}(\mathbf{u}(x_{i+\frac{1}{2}})) \approx \hat{\mathbf{f}}_{i+1/2} := \mathbf{F}\left(\mathbf{u}_h(x^-_{i+1/2}), \mathbf{u}_h(x^+_{i+1/2})\right). \tag{15}$$

The semi-discrete formulation of the DG method is

$$\frac{d}{dt}\left(\int_{K_i} \mathbf{u}_h \cdot \mathbf{w}_h \; dx\right) = \hat{\mathbf{f}}_{i-1/2} \cdot \mathbf{w}_h|_{x^+_{i-1/2}} - \hat{\mathbf{f}}_{i+1/2} \cdot \mathbf{w}_h|_{x^-_{i+1/2}} + \int_{K_i} \mathbf{f}(\mathbf{u}_h) \cdot (\mathbf{w}_h)_x \; dx. \tag{16}$$

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 $(\mathbf{w}_h)_x = \mathbf{0}$, 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 $(\mathbf{e}_l)_{\tilde{l}} = \delta_{l\tilde{l}}$, we obtain

$$\int_{K_i} \mathbf{u}_h \cdot \mathbf{w}_h \; dx = \sum_{k=0}^{p} U_{i,k,l}(t) \; \delta_{jk} = U_{i,j,l}(t)$$

Remark: For implementation, we re-arrange the vector of coefficients $(U_{i,j,l})_{l=1,j=0,i=1}^{l=m,j=p,i=N} \in \mathbb{R}^{m(p+1)N}$ into a matrix $\mathbf{U} \in \mathbb{R}^{m(p+1)\times N}$ of $m(p+1)$ rows and N columns, with entries

$$U_{i,\; j+(p+1)l} := U_{i+1,j,l+1}, \text{ for } j = 0, \ldots, p, \quad l = 0, \ldots, m-1, \quad i = 0, \ldots, N-1.$$

This particular arrangement is used in the initialization step, which has already been provided, of the DG solver.

1i)

Template: src/ancse/polynomial_basis.cpp.

Implement the basis functions ϕ_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.

```
/// Computes the Legendre polynomial basis
/// at a given reference point xi \in [0,1]
Eigen::VectorXd PolynomialBasis :: operator() (double xi) const
{
    Eigen::VectorXd basis(p+1);


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

/// Computes the derivative of Legendre polynomial basis
/// at a given reference point xi \in [0,1]
Eigen::VectorXd PolynomialBasis :: deriv (double xi) const
{
    Eigen::VectorXd basis_deriv(p+1);

    //// ANCSE_CUT_START_TEMPLATE
    basis_deriv(0) = 0;
    if (p > 0) {
        double z = 2*xi - 1;
        basis_deriv(1) = 2*sqrt(3);
        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_sol, which builds the solution at a given point in a specified cell.

That is, `DGHandler::build_sol` receives the vector of coefficients $U_{i,\cdot} \in \mathbb{R}^{m(p+1)}$ of the approximate solution $\mathbf{u}_h$ in cell K_i, and a point x in the cell K_i, and returns $(\mathbf{u}_h)_l(x) = \sum_{k=0}^{p} U_{i,k+l(p+1)}\varphi_{i,k}(x)$, for $l = 0, \dots, m-1$. 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 ϕ_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 $\mathbf{u}_h$.

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 $\mathbf{u}_h\big|_{K_i}$?

Solution:

Legendre polynomials are orthogonal, then we can write

$$\begin{aligned}\int_{K_i} (u_h)_l(x)\, dx &= U_{i,l(p+1)} \int_{K_i} \varphi_{i,0}(x)\, dx + \sum_{k=1}^{p} U_{i,k+l(p+1)} \int_{K_i} \varphi_{i,k}(x)\, dx \\ &= \frac{1}{\sqrt{h}}\left(U_{i,l(p+1)}\; h \int_0^1 \underbrace{\phi_0(\xi(x))}_{=1}\, d\xi + \sum_{k=1}^{p} U_{i,k+l(p+1)}\; h \underbrace{\int_0^1 \phi_k(\xi(x))\; \phi_0(\xi(x))\, d\xi}_{=0} \right).\end{aligned}$$

$$\implies \frac{1}{h}\int_{K_i} (u_h)_l(x)\, dx = \frac{1}{\sqrt{h}} U_{i,l(p+1)}.$$

Listing 14: Building the solution and the cell averages.

```
/// 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*n_coeff, j)*basis0;
        }
    }
    //// ANCSE_END_TEMPLATE

    return u0;
}
```

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.

```
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.

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

    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);

    //// ANCSE_CUT_START_TEMPLATE
    //// ANCSE_COMMENT implement the loop for DG numerical flux term.

    /// 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);
            }
        }
    }
    //// ANCSE_END_TEMPLATE
}

/// DG volume integral term
template <class NumericalFlux>
void DGRateOfChange<NumericalFlux>
:: eval_volume_integral(Eigen::MatrixXd &dudt,
                        const Eigen::MatrixXd &u0) const
{
    //// ANCSE_CUT_START_TEMPLATE
    //// ANCSE_COMMENT implement the loop for DG volume integral.
    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);
                }
            }
        }
    }
    //// ANCSE_END_TEMPLATE
}
```

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

$$\bar{u}_{i,0} := u_{i,0}\ \varphi_{i,0}, \tag{17a}$$

$$\bar{u}_i^- := +\sum_{j=1}^{p} u_{i,j}\ \varphi_{i,j}(x_{i+1/2}^-), \quad \bar{u}_i^+ := -\sum_{j=1}^{p} u_{i,j}\ \varphi_{i,j}(x_{i-1/2}^+). \tag{17b}$$

This implies

$$u_h(x_{i+1/2}^-) = \bar{u}_{i,0} + \bar{u}_i^-,$$
$$u_h(x_{i-1/2}^+) = \bar{u}_{i,0} - \bar{u}_i^+.$$

Additionally, define

$$\Delta_+\bar{u}_{i,0} := \bar{u}_{i+1,0} - \bar{u}_{i,0}, \quad \Delta_-\bar{u}_{i,0} := \bar{u}_{i,0} - \bar{u}_{i-1,0}. \tag{18}$$

Using the so-called van Leer limiter:

$$\tilde{\bar{u}}_i^- := g(\bar{u}_i^-, \Delta_-\bar{u}_{i,0}, \Delta_+\bar{u}_{i,0}), \quad \tilde{\bar{u}}_i^+ := g(\bar{u}_i^+, \Delta_-\bar{u}_{i,0}, \Delta_+\bar{u}_{i,0}), \tag{19}$$

with the minmod function

$$g(a_1, \dots, a_n) := \begin{cases} s \min_{1\le l\le n} |a_i|, & s = sgn(a_1) = \ldots = sgn(a_n), \\ 0, & \text{otherwise.} \end{cases} \tag{20}$$

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

1m)

Template: src/ancse/dg_handler.cpp.

Implement the function build_split_sol, which returns $\bar{\mathbf{u}}_{i,0}$, $\bar{\mathbf{u}}_i^+$ and $\bar{\mathbf{u}}_i^-$, c.f. (17).

Implement the function build_limit_coeffs, which computes the coefficients $\tilde{U}_{i,j}$ 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(
            "Limiter not implemented for higher than 3rd order");
    }

    //// ANCSE_CUT_START_TEMPLATE
    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);
            }
        }
    }
    //// ANCSE_END_TEMPLATE
}
```

1n)

Template: src/ancse/dg_limiting.cpp.

Implement the DG limiting procedure for the conserved variables.

Solution:

Listing 18: Implementation of the DG limiting loop.

```
/// applies the DG limiting procedure to the solution coefficients
/// for all cells.
template <class Limiter>
```

```
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());

    //// ANCSE_CUT_START_TEMPLATE

    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());
    auto sm = uc0 - um0;
    auto sp = up0 - uc0;

    //// ANCSE_CUT_START_TEMPLATE
    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

    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.

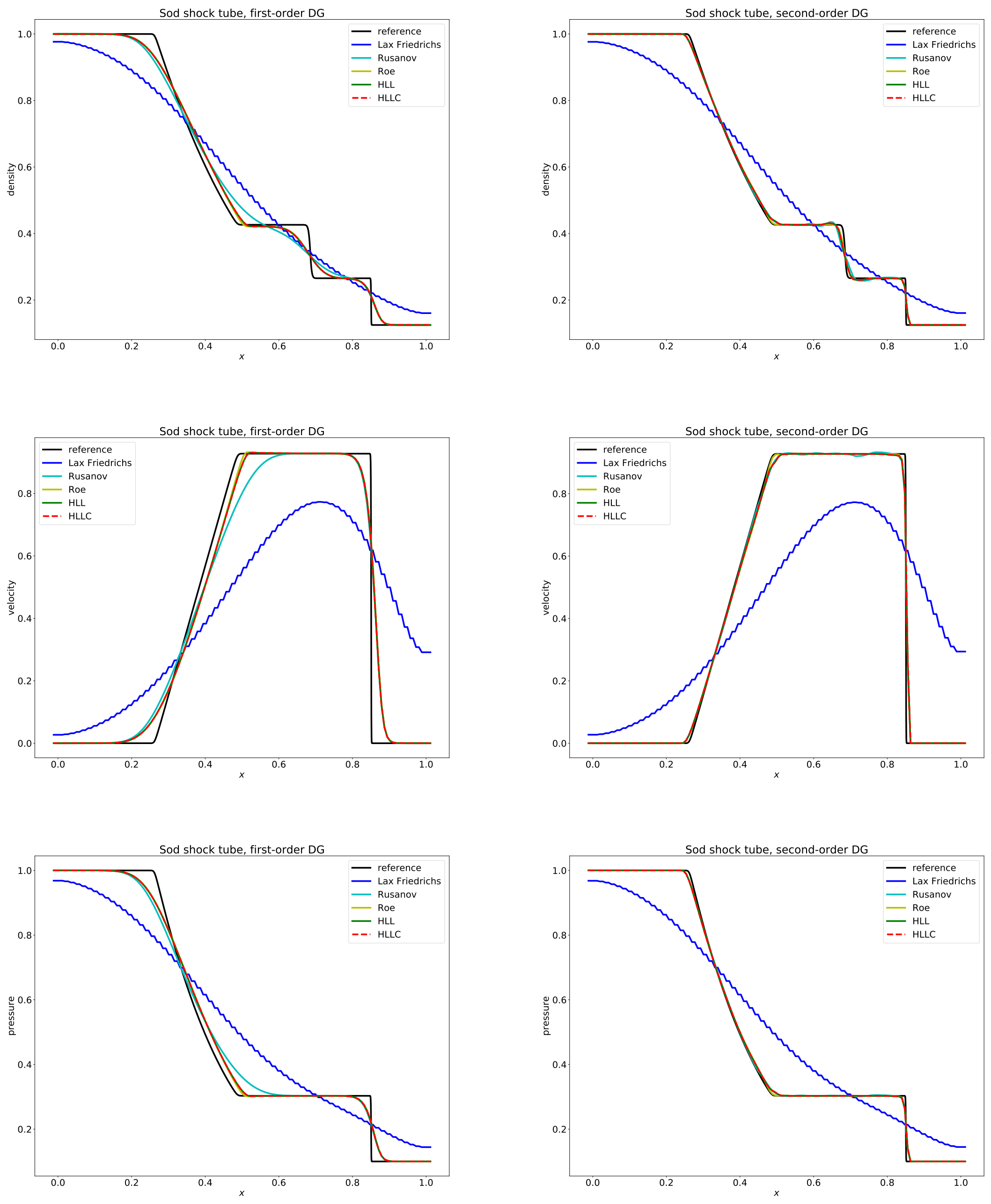

Figure 6: First- and second-order DG; density, velocity and pressure plots; $T = 0.2$.

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

The Euler equations are then given as:

$$\begin{aligned} \rho_t + \nabla \cdot (\rho \mathbf{u}) &= 0 \\ (\rho \mathbf{u})_t + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u} + p\mathbf{I}) &= 0 \\ E_t + \nabla \cdot ((E + p)\mathbf{u}) &= 0. \end{aligned} \tag{21}$$

Here

$$\mathbf{u} \otimes \mathbf{u} := \begin{pmatrix} u_1 u_1 & u_1 u_2 \\ u_2 u_1 & u_2 u_2 \end{pmatrix}$$

The total energy of the system is given by

$$E = \frac{1}{2} \rho \mathbf{u} \cdot \mathbf{u} + \rho e.$$

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

$$e = \frac{p}{\gamma - 1},$$

where γ is the adiabatic constant. We set

$$\gamma = 1.4.$$

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

$$\mathbf{U}_t + \nabla_x \cdot \mathbf{F}(\mathbf{U}) = 0 \tag{22}$$

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

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

where

$$\mathbf{F}(\mathbf{U}) = \begin{pmatrix} F_1(\mathbf{U}) \\ \vdots \\ F_N(\mathbf{U}) \end{pmatrix} = \begin{pmatrix} F_{1,1}(\mathbf{U}) & F_{1,2}(\mathbf{U}) & \cdots & F_{1,d}(\mathbf{U}) \\ \vdots & \vdots & \vdots & \vdots \\ F_{N,1}(\mathbf{U}) & F_{N,2}(\mathbf{U}) & \cdots & F_{N,d}(\mathbf{U}) \end{pmatrix}.$$

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

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ E \end{pmatrix} =: \begin{pmatrix} \rho \\ m_x \\ m_y \\ E \end{pmatrix} \tag{23}$$

where we assume $\rho > 0$.

2a)

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

Solution: We set

$$\mathbf{F}\begin{pmatrix} \rho \\ m_x \\ m_y \\ E \end{pmatrix} = \begin{pmatrix} m_x & m_y \\ \frac{m_x m_x}{\rho} + p & \frac{m_x m_y}{\rho} \\ \frac{m_y m_x}{\rho} & \frac{m_y m_y}{\rho} + p \\ \frac{m_x(E+p)}{\rho} & \frac{m_y(E+p)}{\rho} \end{pmatrix} = \begin{pmatrix} m_x & m_y \\ \rho u_1^2 + p & \rho u_1 u_2 \\ \rho u_2 u_1 & \rho u_2^2 + p \\ u_1(E+p) & u_2(E+p) \end{pmatrix}. \tag{24}$$

□

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

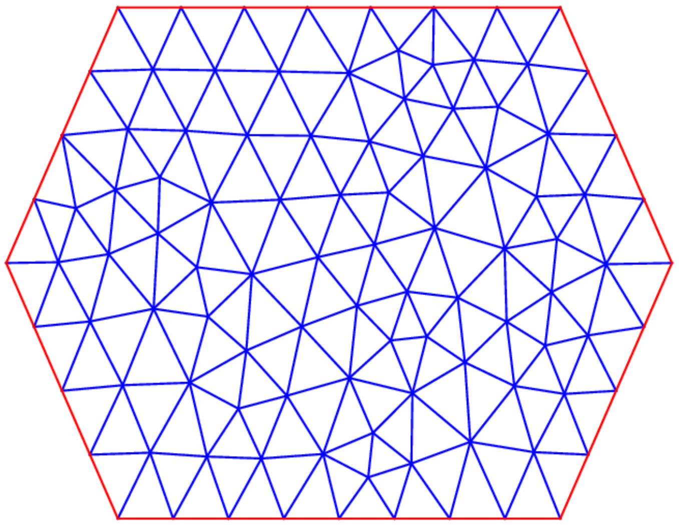

Figure 7: Example of our domain Ω.

2b)

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

$$\bar{\mathbf{U}}_i(t) := \frac{1}{|K|} \int_K \mathbf{U}(\mathbf{x}, t)\, d\mathbf{x}.$$

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

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

This represents the flux through a surface with normal vector $\mathbf{n}$.

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 $\mathbf{U}$ is a smooth solution of (22). Fix $t^n, \Delta t \geq 0$. Show that

$$\bar{\mathbf{U}}_i(t^n + \Delta t) - \bar{\mathbf{U}}_i(t^n) = -\frac{1}{|K_i|} \sum_{k=1}^{3} \int_{t^n}^{t^n+\Delta t} \int_{e_{i,j_k}} \mathbf{F}(\mathbf{U}) \cdot \mathbf{n}_{i,j_k} \, dS \, dt, \tag{25}$$

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

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

$$\mathbf{U}_t + \nabla_x \cdot \mathbf{F}(\mathbf{U}) = 0 \tag{26}$$

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 $\mathbf{U}$ is smooth on K_i compact, we have integrability and thus

$$\begin{aligned}
0 &= \frac{1}{|K_i|} \int_{K_i} \int_{t^n}^{t^n+\Delta t} \frac{\partial}{\partial t} \mathbf{U}(\mathbf{x}, t) \, dt \, d\mathbf{x} + \frac{1}{|K_i|} \int_{K_i} \int_{t^n}^{t^n+\Delta t} \nabla_x \cdot \mathbf{F}(\mathbf{U}(\mathbf{x}, t)) \, dt \, d\mathbf{x} && (27) \\
&= \frac{1}{|K_i|} \int_{K_i} \left(\mathbf{U}(\mathbf{x}, t^n + \Delta t) - \mathbf{U}(\mathbf{x}, t^n)\right) \, dt \, d\mathbf{x} + \frac{1}{|K_i|} \int_{t^n}^{t^n+\Delta t} \int_{K_i} \nabla_x \cdot \mathbf{F}(\mathbf{U}(\mathbf{x}, t)) \, d\mathbf{x} \, dt && (28) \\
&= \bar{\mathbf{U}}_i(t^n + \Delta t) - \bar{\mathbf{U}}_i(t) + \frac{1}{|K_i|} \int_{t^n}^{t^n+\Delta t} \int_{\partial K_i} \mathbf{F}(\mathbf{U}(\mathbf{x}, t)) \cdot \mathbf{n} \, dS \, dt && (29)
\end{aligned}$$

where in the last step, we have used the definition of $\bar{\mathbf{U}}$, Gauss' theorem, and $\mathbf{n}$ is the outward-pointing unit vector normal to ∂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 $\partial K_i = \bigcup_{k=1}^{3} e_{i,j_k}$, along each of which $\mathbf{n} \equiv \mathbf{n}_{i,j_k}$. Since this union is disjoint (other than sets of measure 0), we can conclude

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

$\square$

2c)

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

$$\mathbf{F}(\mathbf{U}) \cdot \mathbf{n} = \begin{pmatrix} \rho(\mathbf{u} \cdot \mathbf{n}) \\ \rho(\mathbf{u} \cdot \mathbf{n}) u_1 + p \, n_1 \\ \rho(\mathbf{u} \cdot \mathbf{n}) u_2 + p \, n_2 \\ (\mathbf{u} \cdot \mathbf{n})(E + p) \end{pmatrix}. \tag{30}$$

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 $\mathbf{x} \mapsto (\xi, \zeta)$ such that $\mathbf{x} = \xi \mathbf{n} + \zeta \tau$. Define $u_\perp = \mathbf{u} \cdot \mathbf{n}$ and $u_\parallel = \mathbf{u} \cdot \tau$, then the resulting system of PDEs is

$$\begin{aligned}
\partial_t \rho + \partial_\xi \rho u_\perp + \partial_\zeta \rho u_\parallel &= 0 && (31)\\
\partial_t \rho u_\perp + \partial_\xi (\rho u_\perp^2 + p) + \partial_\zeta \rho u_\parallel u_\perp &= 0 && (32)\\
\partial_t \rho u_\parallel + \partial_\xi \rho u_\perp u_\parallel + \partial_\zeta (\rho u_\parallel^2 + p) &= 0 && (33)\\
\partial_t E + \partial_\xi u_\perp (E + p) + \partial_\zeta u_\parallel (E + p) &= 0 && (34)
\end{aligned}$$

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 ξ-direction is

$$\mathbf{f}(\mathbf{U}) = \begin{pmatrix} \rho u_\perp \\ \rho u_\perp^2 + p \\ \rho u_\perp u_\parallel \\ u_\perp (E + p) \end{pmatrix}. \quad (35)$$

Use these ideas to approximate the contribution of edge e

$$\int_t^{t+\Delta t} \int_{e_{i,j}} \mathbf{F}(\mathbf{U}) \cdot \mathbf{n}_{i,j} \, dS \, dt, \quad (36)$$

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.

$$\begin{aligned}
T(q) &= (q_1, q_2 n_1 + q_3 n_2, q_2 \tau_1 + q_3 \tau_2, q_4) && (37)\\
T^{-1}(q) &= (q_1, q_2 n_1 + q_3 \tau_1, q_2 n_2 + q_3 \tau_2, q_4) && (38)
\end{aligned}$$

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 ξ-axis. Hence in the rotated system only $\mathbf{f}$ 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:

$$\int_{e_{i,j}} \mathbf{F}(\mathbf{U}) \cdot \mathbf{n_{i,j}} dS = \int_{e_{i,j}} T^{-1}\mathbf{f}(T(\mathbf{U}))dS \approx |e_{i,j}| T^{-1} F(T(\bar{\mathbf{U}}_i), T(\bar{\mathbf{U}}_j)) \tag{39}$$

where $F(\cdot,\cdot)$ is a chosen numerical flux.

2d)

For $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$, let $\{\lambda_i(\mathbf{f},\mathbf{U})\}_{i=1}^N$ be the eigenvalues of the Jacobian of $\mathbf{f}(\mathbf{U})$. Define

$$\lambda_{\max}(\mathbf{f},\mathbf{U}) := \max_i |\lambda_i(\mathbf{f},\mathbf{U})|.$$

Show that for flux $\mathbf{F}(\mathbf{U}) \cdot \mathbf{n}$, it holds that:

$$\lambda_{\max}(\mathbf{F} \cdot \mathbf{n}, \mathbf{U}) = |\mathbf{u} \cdot \mathbf{n}| + \sqrt{\frac{\gamma p}{\rho}}. \tag{40}$$

Use this to derive the following CFL-condition:

$$\Delta t < C_{CFL} \frac{\Delta x}{\max_i \lambda_{max}(\mathbf{F} \cdot \mathbf{n}, \mathbf{U}_i)}, \tag{41}$$

for all unit-vectors $\mathbf{n}$. Here Δx is the small inradius and $C_{CFL} = 0.45$.

Finally, implement the CLF condition in the file cfl_condition.hpp. Please consider that most of the functions you need for this task and the following are already implemented in euler.hpp.

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

Solution: The eigenvalues of the Jacobian of $F_{.,1}$ 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 $\mathbf{f}$, 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

$$\begin{cases} \lambda_1 = \mathbf{u} \cdot \mathbf{n} - c & \text{if } \mathbf{u} \cdot \mathbf{n} < 0 \\ \lambda_4 = \mathbf{u} \cdot \mathbf{n} + c & \text{if } \mathbf{u} \cdot \mathbf{n} \geq 0 \end{cases},$$

and minding that $c > 0$, it is clear that in both cases,

$$|\lambda_{\max}| = |\mathbf{u} \cdot \mathbf{n}| + c$$

If we take the maximum of $|\lambda_{max}|$ over all directions $\mathbf{n}$ we get

$$\sup_{\mathbf{n}:|\mathbf{n}|=1} |\mathbf{u} \cdot \mathbf{n}| + c = |u| + c \tag{43}$$

The geometric idea behind the CFL condition is that the numerical speed of propagation $\frac{\Delta x}{\Delta t}$ must exceed the physical speed of propagation, i.e. $|u| + c$. Some freedom exists in choosing the length scale Δx, here we conservatively choose the minimum inradius.

Therefore,

$$\Delta t \leq C_{CFL} \frac{\Delta x}{|u| + c} \tag{44}$$

is one possible CFL condition.

The CFL condition is implemented as follows.

Listing 19: Implementation of the CFL condition. label

```
    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();

        //// ANCSE_CUT_START_TEMPLATE
#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 in numerical_flux.hpp 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

$$\int_t^{t+\Delta t} \int_e \mathbf{F}(\mathbf{U}) \cdot \mathbf{n} \, dS dt = \Delta t |e| \mathbf{F}(\mathbf{U}) \cdot \mathbf{n}. \tag{45}$$

The reflective (or wall) flux boundary conditions are

$$\int_t^{t+\Delta t} \int_e \mathbf{F}(\mathbf{U}) \cdot \mathbf{n} \, dS dt = \Delta t \mathcal{F}(\mathbf{U}, \mathbf{U}^*, \mathbf{n}) \tag{46}$$

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

$$\mathbf{U}^* = (\rho,\ -\rho(\mathbf{u} \cdot \mathbf{n})\mathbf{n} + \rho(\mathbf{u} \cdot \tau)\tau,\ E). \tag{47}$$

Here $\mathcal{F}$ 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 20: Implementation of the outflow boundary conditions.

```
/// Compute the outflow flux through the k-th interface of cell i.
/** Note: you know that edge k is an outflow edge.
 */
EulerState computeOutflowFlux(const Eigen::MatrixXd &U,
                              int i,
                              int k,
                              const Mesh &mesh) const {
    //// ANCSE_COMMENT Implement the outflow flux boundary condition.

    auto normal = mesh.getUnitNormal(i, k);
    //// ANCSE_CUT_START_TEMPLATE

    auto f = euler::flux(euler::localCoordinates(U.row(i), normal));
    return euler::globalCoordinates(f, normal);

    //// ANCSE_RETURN_VALUE EulerState{};
    //// ANCSE_END_TEMPLATE
}
```

Listing 21: Implementation of the reflective boundary conditions.

```
/// Compute the reflective boundary flux through the k-th edge of cell i.
/** Note: you know that edge k is a reflective/wall boundary edge.
 */
EulerState computeReflectiveFlux(const Eigen::MatrixXd &U,
                                 int i,
                                 int k,
                                 const Mesh &mesh) const {


    //// ANCSE_COMMENT Implement the reflective flux boundary condition.

    auto normal = mesh.getUnitNormal(i, k);

    //// ANCSE_CUT_START_TEMPLATE
    auto uL = euler::localCoordinates(U.row(i), normal);

    EulerState uR = uL;

    // Flip the momentum of the normal component.
    uR[1] = -uR[1];

    auto nf = hllc(uL, uR);
    return euler::globalCoordinates(nf, normal);
    //// ANCSE_RETURN_VALUE EulerState{};
    //// ANCSE_END_TEMPLATE
}
```

The final step in implementing the second order FVM is to compute the trace values of $\mathbf{U}$ at the interfaces.

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

$$q_i(\mathbf{x}) = Q_i + (\nabla q(\mathbf{x}_i)) \cdot (\mathbf{x} - \mathbf{x}_i). \tag{48}$$

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 ∇q.

2f)

Approximate the right hand side of

$$\nabla q(\mathbf{x}_i) \approx \frac{1}{|K_i|} \int_{K_i} \nabla q \; d\mathbf{x} \tag{49}$$

by Gauss' theorem (applied to the vector field $q\mathbf{c}$, for certain constant $\mathbf{c} \in \mathbb{R}^2$), and the mid-point rule for the resulting integrals over the boundary. Then, implement the function `compute_gradients` in the file `gradient.hpp`.

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

$$\int_{K_i} \nabla \cdot \mathbf{G} \; d\mathbf{x} = \int_{\partial K_i} \mathbf{G} \cdot \mathbf{n} \; dS \tag{50}$$

$$\iff \int_{K_i} (\nabla q \cdot \mathbf{c}) \, d\mathbf{x} = \int_{\partial K_i} (\mathbf{c} \cdot \mathbf{n}) q \; dS \tag{51}$$

$$= \sum_{k=1}^{3} \int_{e_{i,j_k}} (\mathbf{c} \cdot \mathbf{n}_{i,j_k}) q \; dS \tag{52}$$

$$\approx \sum_{k=1}^{3} |e_{i,j_k}| (\mathbf{c} \cdot \mathbf{n}_{i,j_k}) q(\mathbf{x}_{i,j_k}) \tag{53}$$

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 $\mathbf{x}_{i,j_k}$. We denote by $|e_{i,j_k}|$ the length of the edge. Applying the argument above for all $\mathbf{c} \in \{(1,0),(0,1)\}$, we obtain (component-by-component):

$$\int_{K_i} \nabla q \; d\mathbf{x} = \sum_{k=1}^{3} |e_{i,j_k}| \, \mathbf{n}_{i,j_k} \, q(\mathbf{x}_{i,j_k})$$

We can implement the proposed approximate as follows. Note, that this is the same for both primitive and conserved variables. If q_bar 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 22: Implementation of the outflow boundary conditions.

```
/// Compute the gradients of q_bar.
/** 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) {



    dqdx.setZero();
    dqdy.setZero();

    int n_cells = mesh.getNumberOfTriangles();

    for (int i = 0; i < n_cells; ++i) {
        // Let's first deal with the case where we can't compute the gradient
        // via Gauss.
        if (!mesh.isValidNeighbour(i, 0) || !mesh.isValidNeighbour(i, 1)
            || !mesh.isValidNeighbour(i, 2)) {
            continue;
        }


        double area = mesh.getTriangleArea(i);


        // w.l.o.g. there are enough neighbours.
        for (int k = 0; k < 3; ++k) {
            auto n = mesh.getUnitNormal(i, k);
            double length = mesh.getEdgeLength(i, k);

            int j = mesh.getNeighbour(i, k);

            Eigen::Vector2d xi = mesh.getCellCenter(i);
            Eigen::Vector2d xj = mesh.getCellCenter(j);
            Eigen::Vector2d x_ij = mesh.getEdgeCenter(i, k);

            double al = (x_ij - xi).norm() / (xi - xj).norm();

            //// ANCSE_COMMENT Compute the gradient of all 4 components of q_bar
            //// ANCSE_CUT_START_TEMPLATE
            EulerState q_ij = (1.0 - al) * q_bar.row(i) + al * q_bar.row(j);

            dqdx.row(i) += q_ij * n[0] * length / area;
            dqdy.row(i) += q_ij * n[1] * length / area;

            //// ANCSE_END_TEMPLATE
        }
    }


}
```

2g)

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 $\mathbf{x}_{i,j}$. One can compute a limited slope as follows

$$s_{i,j} = \xi(\nabla q(\mathbf{x}_i) \cdot \Delta \mathbf{x}_{i,j}, \nabla q(\mathbf{x}_j) \cdot \Delta \mathbf{x}_{i,j}) \tag{54}$$

with $\Delta \mathbf{x}_{i,j} = \mathbf{x}_{i,j} - \mathbf{x}_i$ and a slope limiter ξ of your choice. The reconstructed values are then

$$q_{i,j} = Q_i + s_{i,j} \quad \text{and} \quad q_{j,i} = Q_j + s_{j,i}. \tag{55}$$

In the file numerical_flux.hpp implement the function to compute the fluxes among interior cells. Use piecewise linear reconstruction of the primitive variables (ρ, u_1, u_2, p) according to 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 23: Implementation of the flux through an interface between two cells in the interior.

```
/// Compute the flux through the k-th interface of cell i.
/** Note: This edge is an interior edge, therefore approximate the flux
 * through this edge with the appropriate FVM formulas.
 */
EulerState computeInteriorFlux(const Eigen::MatrixXd &U,
                               int i,
                               int k,
                                         const Mesh &mesh) const {

    //// 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.

    int j = mesh.getNeighbour(i, k);
    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);

    //// ANCSE_COMMENT w are the primitive variables
    //// ANCSE_COMMENT u are the conservative variables

    EulerState uL, uR;
    EulerState wL, wR;

    //// ANCSE_CUT_START_TEMPLATE

    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
}
```

2h)

Complete the loop to copute the rate of change in numerical_fluxes.hpp.

Solution: The loop to compute the rate of change is showed in the following code snippet:

Listing 24: Implementation of the loop to compute the rare of change.

```
class FluxRateOfChange {
  public:
    explicit FluxRateOfChange(int n_cells) {

        dwdx = Eigen::MatrixXd(n_cells, 4);
        dwdy = Eigen::MatrixXd(n_cells, 4);
        pvars = Eigen::MatrixXd(n_cells, 4);

    }

    void operator()(Eigen::MatrixXd &dudt,
                    const Eigen::MatrixXd &u,
                    const Mesh &mesh) const {


        //// ANCSE_COMMENT Compute the rate of change of u.
        //// ANCSE_COMMENT
        //// ANCSE_COMMENT Note: Please use the method `computeFlux` to abstract
        //// ANCSE_COMMENT away the details of computing the flux through a
        //// ANCSE_COMMENT given interface.
        //// ANCSE_COMMENT
        //// ANCSE_COMMENT Note: You can use `assert_valid_flux` to check
        //// ANCSE_COMMENT if what `computeFlux` returns makes any sense.
        //// ANCSE_COMMENT
        //// ANCSE_COMMENT Note: Do not assume `dudt` is filled with zeros.

        dudt.resize(u.rows(), u.cols());
        dudt.setZero();

        int n_cells = mesh.getNumberOfTriangles();

        for (int i = 0; i < n_cells; ++i) {
            pvars.row(i) = euler::primitiveVars(u.row(i));
        }

        //// ANCSE_COMMENT Note: This will only serve for the linear recontruction
            ↪ .
        //// ANCSE_COMMENT if the computation of the gradients is not done in
            ↪ gradient.hpp, then
        //// ANCSE_COMMENT dwdx and dwdy contains only zeros

        compute_gradients(dwdx, dwdy, pvars, mesh);

        //// ANCSE_CUT_START_TEMPLATE

#pragma omp parallel for
        for (int i = 0; i < n_cells; ++i) {
            double area = mesh.getTriangleArea(i);
            for (int k = 0; k < 3; ++k) {
                auto edge_length = mesh.getEdgeLength(i, k);

                auto nF = computeFlux(u, i, k, mesh);
                assert_valid_flux(mesh, i, k, nF);

                dudt.row(i) -= nF * edge_length / area;
            }
        }

        //// ANCSE_END_TEMPLATE
    }



    void assert_valid_flux(const Mesh &mesh,
                           int i,
                           int k,
                           const EulerState &nF) const {
        // This is mostly for debugging (but also important to check in
        // real simulations!): Make sure our flux contribution is not
        // nan (ie. it is not not a number, ie it is a number)
        if (!euler::isValidFlux(nF)) {
            // clang-format off
            throw std::runtime_error(
                "invalid value detected in numerical flux, " + euler::to_string(nF)
                + "\nat triangle: " + std::to_string(i)
                + "\nedge: "      + std::to_string(k)
                + "\nis_boundary: " + std::to_string(!mesh.isValidNeighbour(i, k)))
                    ↪ ;
            // clang-format on
        }
    }

    /// Compute the flux through the k-th interface of cell i.
    EulerState computeFlux(const Eigen::MatrixXd &U,
                           int i,
                        int k,
                        const Mesh &mesh) const {
        auto boundary_type = mesh.getBoundaryType(i, k);

        if (boundary_type == Mesh::BoundaryType::INTERIOR_EDGE) {
            return computeInteriorFlux(U, i, k, mesh);
        } else {
            if (boundary_type == Mesh::BoundaryType::OUTFLOW_EDGE) {
                return computeOutflowFlux(U, i, k, mesh);
            } else /* boundary_type == Mesh::BoundaryType::WING_EDGE */
            {
                return computeReflectiveFlux(U, i, k, mesh);
            }
        }
    }
```

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.