Parallelization of "stencil-based" methods: step-69 (and beyond)

Martin Kronbichler[1], Matthias Maier, Ignacio Tomas[2]

Department of Mathematics
Texas A&M University

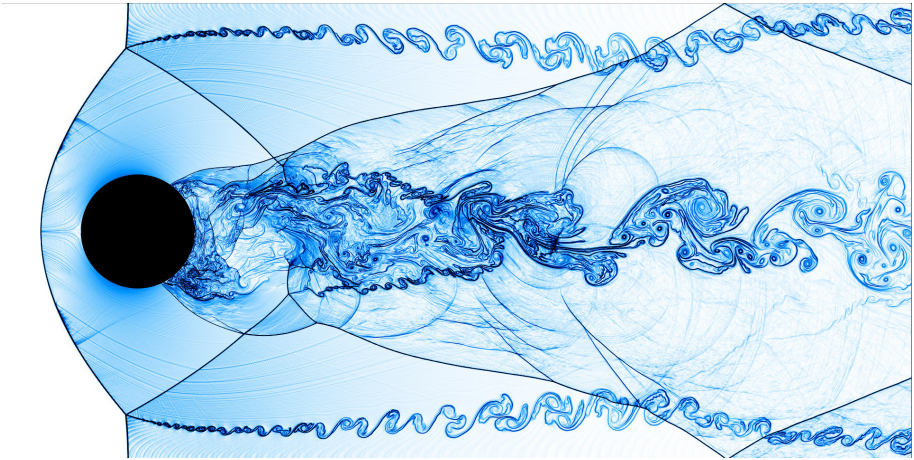
[1] Institute for Computational Mechanics, TUM, Munich

[2] CSRI, Sandia National Laboratories

Motivation

Motivation

Simulation of hyperbolic systems of conservation laws



Goals:
- robust (?!)
- accurate (?!)
- scalable (!)

Outline

The compressible Euler equations

Compressible Euler equations

Hyperbolic system of conservation laws

$$\mathbf{u}_t + \operatorname{div} f(\mathbf{u}) = \mathbf{0},$$

where $\mathbf{u}(\mathbf{x}, t) : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^{d+2}$, and $\mathbf{f}(\mathbf{u}) : \mathbb{R}^{d+2} \to \mathbb{R}^{(d+2)\times d}$, with space dim. $d \geq 1$.

$$\text{State:} \quad \mathbf{u} = \begin{bmatrix} \rho \\ \mathbf{m} \\ E \end{bmatrix} \qquad \text{Flux:} \quad f(\mathbf{u}) = \begin{bmatrix} \mathbf{m}^\top \\ \rho^{-1}\mathbf{m} \otimes \mathbf{m} + \mathbb{I}p \\ \frac{\mathbf{m}^\top}{\rho}(E + p) \end{bmatrix},$$

Here $\rho \in \mathbb{R}^+$ denotes the density, $\mathbf{m} \in \mathbb{R}^d$ is the momentum, and $E \in \mathbb{R}^+$ is the total energy of the system. The pressure $p \in \mathbb{R}^+$ is determined by an equation of state, e. g.,

$$\text{Polytropic gas equation:} \quad p(\mathbf{u}) := (\gamma - 1)\left(E - \tfrac{|\mathbf{m}|^2}{2\rho}\right), \quad \gamma \in (1, 5/3].$$

Compressible Euler equations

Variational principle viable ???

Testing with $\mathbf{u}$:

$$\frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\left\|\mathbf{u}\right\|_{L^2\Omega}^2 + \underbrace{\left(\operatorname{div} f(\mathbf{u}), \mathbf{u}\right)_{L^2(\Omega)}}_{???} = 0.$$

- no energy estimate available
- no (quasi) best approximation
- etc.

$\longrightarrow$ none of the classical finite element toolbox available...

Solution theory

We call $\mathbf{u}$ a viscosity solution if

$$\mathbf{u} = \lim_{\varepsilon \to 0} \mathbf{u}^\varepsilon,$$

with

$$\mathbf{u}_t^\varepsilon + \operatorname{div} f(\mathbf{u}^\varepsilon) = \varepsilon \Delta \mathbf{u}^\varepsilon.$$

$\longrightarrow$ global existence and uniqueness of viscosity solutions is an open problem.

So what can we do?

Compressible Euler equations

What can we meaningfully expect?

Invariant set

If $\mathbf{u}$ is a viscosity solution, then

$$\mathbf{u}(\mathbf{x}, t) \in \mathcal{B} \quad \forall\, \mathbf{x} \in \Omega,\ \forall\, t \geq 0,$$

where $\mathbf{u} \in \mathcal{B}$ implies

- positivity of density: $\rho > 0$
- positivity of internal energy:

$$E - \frac{|\mathbf{m}|^2}{2\rho} > 0$$

- local minimum principle on specific entropy:

$$s(\mathbf{u}) \geq \min_{x \in \Omega} s(\mathbf{u}_0(\mathbf{x}))$$

Conservation

Conservation of mass, momentum and total energy:

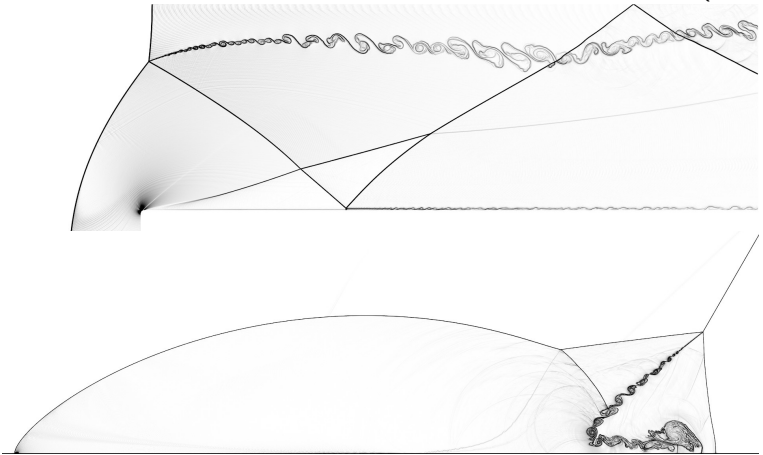
$$\frac{\mathrm{d}}{\mathrm{d}t} \int_\Omega \mathbf{u}\, \mathrm{d}t + \int_{\partial\Omega} \mathbf{n} \cdot f(\mathbf{u})\, \mathrm{d}o_{\mathbf{x}} = 0.$$

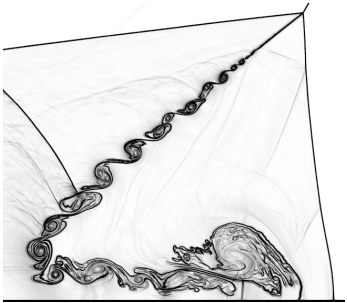
$\longrightarrow$ Robustness: We want a numerical scheme that is conservative and maintains the invariant set.

Compressible Euler equations

How does (a subset of) the community judge our code?

Picture norm comparison of flow characteristics to (known) experimental results.





Compressible Euler equations

To summarize:

- Robustness: conservative and invariant domain preserving

  Most importantly: This implies that our scheme will never crash
  (no matter what mesh was used, or (admissible) initial data was prescribed)

- Accuracy (not discussed): second order convergent in case of known, smooth viscosity solutions

- Scalability: Large-scale 3D computation

step-69

step-69

- Guermond & Popov, *Invariant domains and first-order continuous finite element approximation for hyperbolic systems*, SIAM J. Numer. Anal. 54(4):2466-2489.

(Grossly oversimplified) formal derivation

Let $\mathbf{u}_h(\mathbf{x},t) = \sum_j \varphi_j(\mathbf{x})\mathbf{U}_j(t)$ be a finite element approximation:

$$\frac{\mathrm{d}}{\mathrm{d}t}\big(\varphi_i, \mathbf{u}_h\big) + \big(\varphi_i, \operatorname{div}\mathbb{f}(\mathbf{u}_h)\big) = 0.$$

Forward Euler:

$$\sum_{j\in\mathcal{I}(i)} m_{ij}\frac{\mathbf{U}_j^{n+1}-\mathbf{U}_j^n}{\tau_n} + \Big(\varphi_i, \operatorname{div}\mathbb{f}\Big(\sum_{j\in\mathcal{I}(i)}\mathbf{U}_j^n\Big)\Big) = 0.$$

Lump mass matrix and approximate flux, $\mathbb{f}(\mathbf{u}_h^n) \approx \sum_j \mathbb{f}(\mathbf{U}_j^n)\,\varphi_j$:

$$m_i\frac{\mathbf{U}_i^{n+1}-\mathbf{U}_i^n}{\tau_n} + \sum_{j\in\mathcal{I}(i)}\mathbb{f}\big(\mathbf{U}_j^n\big)\cdot\mathbf{c}_{ij} = 0, \qquad m_i = \int_\Omega \varphi_i, \quad \mathbf{c}_{ij} = \int_\Omega \varphi_i\nabla\varphi_j.$$

step-69

- Guermond & Popov, *Invariant domains and first-order continuous finite element approximation for hyperbolic systems*, SIAM J. Numer. Anal. 54(4):2466-2489.

Scheme

$$m_i \frac{\mathbf{U}_i^{n+1} - \mathbf{U}_i^n}{\tau_n} + \sum_{j \in \mathcal{I}(i)} \mathbb{f}\left(\mathbf{U}_j^n\right) \cdot \mathbf{c}_{ij} - \sum_{j \in \mathcal{I}(i)} d_{ij}^n \mathbf{U}_j^n \;=\; 0,$$

where

$$m_i = \int_\Omega \varphi_i, \qquad \mathbf{c}_{ij} = \int_\Omega \varphi_i \nabla \varphi_j,$$

and where we have introduced a "graph viscosity" d_{ij}^n as stabilization term.

The correct construction of d_{ij}^n is key for robustness

step-69

Compute two characteristic propagation speeds associated with $\boldsymbol{U}_i^n$ or $\boldsymbol{U}_j^n$:

$$\lambda_-^1(\boldsymbol{U}_i^n, p^*) \coloneqq \tilde{u}_i^n - \tilde{c}_i^n \sqrt{1 + \frac{\gamma+1}{2\gamma}\left[\frac{p^* - \tilde{p}_i^n}{\tilde{p}_i^n}\right]_{\text{pos}}}, \quad \lambda_+^3(\boldsymbol{U}_j^n, p^*) \coloneqq \tilde{u}_j^n + \tilde{c}_j^n \sqrt{1 + \frac{\gamma+1}{2\gamma}\left[\frac{p^* - \tilde{p}_j^n}{\tilde{p}_j^n}\right]_{\text{pos}}},$$

and a two-rarefaction pressure:

$$\tilde{p}^*(\boldsymbol{U}_i^n, \boldsymbol{U}_j^n) = \tilde{p}_j \left(\frac{\tilde{c}_i + \tilde{c}_j - \frac{\gamma-1}{2}(\tilde{u}_j - \tilde{u}_i)}{\tilde{c}_i \left(\frac{\tilde{p}_i}{\tilde{p}_j}\right)^{-\frac{\gamma-1}{2\gamma}} + \tilde{c}_j} \right)^{\frac{2\gamma}{\gamma-1}},$$

and a monotone increasing and concave down function

$$\psi(p) \coloneqq f(\boldsymbol{U}_i^n, p) + f(\boldsymbol{U}_j^n, p) + \tilde{u}_j - \tilde{u}_i, \qquad f(\boldsymbol{U}, p) \coloneqq \begin{cases} \dfrac{\sqrt{2}\,(p - \tilde{p})}{\sqrt{\tilde{\rho}\left[(\gamma+1)\,p + (\gamma-1)\,\tilde{p}\right]}} & \text{if } p \geq \tilde{p}, \\ \left[(p/\tilde{p})^{\frac{\gamma-1}{2\gamma}} - 1\right] \dfrac{2\,\tilde{c}}{\gamma-1}, & \text{otherwise.} \end{cases}$$

Now:

$$\tilde{\lambda}_{\max} = \max\left(\left[\lambda_-^1(\boldsymbol{U}_i^n, p^*)\right]_{\text{neg}}, \left[\lambda_+^3(\boldsymbol{U}_j^n, p^*)\right]_{\text{pos}} \right), \quad p^* \coloneqq \begin{cases} \tilde{p}^*(\boldsymbol{U}_i^n, \boldsymbol{U}_j^n) & \text{if } \psi(p_{\max}) < 0, \\ \min(\tilde{p}_{\max}, \tilde{p}^*(\boldsymbol{U}_i^n, \boldsymbol{U}_j^n)) & \text{otherwise.} \end{cases}$$

step-69

Scheme

$$\mathbf{U}_i^{n+1} \;=\; \mathbf{U}_i^n \;+\; \frac{\tau_n}{m_i}\Big(\sum_{j\in\mathcal{I}(i)} \mathbb{f}\left(\mathbf{U}_j^n\right)\cdot \mathbf{c}_{ij} \;-\; \sum_{j\in\mathcal{I}(i)} d_{ij}^n \mathbf{U}_j^n \Big)$$

where
$$m_i = \int_\Omega \varphi_i, \quad \mathbf{c}_{ij} = \int_\Omega \varphi_i \nabla \varphi_j, \quad d_{ij}^n = \max\left(\tilde{\lambda}_{\max}(\boldsymbol{n}_{ij}, \boldsymbol{U}_i^n, \boldsymbol{U}_j^n)\,|\boldsymbol{c}_{ij}|\,,\; \tilde{\lambda}_{\max}(\boldsymbol{n}_{ji}, \boldsymbol{U}_j^n, \boldsymbol{U}_i^n)\,|\boldsymbol{c}_{ji}| \right)$$

$\mathcal{I}(i)$ — all column indices coupling to i.

Stencil based: compare to matrix-vector multiplication

$$y_i \;=\; \sum_{j\in\mathcal{I}(i)} a_{ij}\, x_j,$$

. . . but highly nonlinear!

step-69

euler_step

// Step 1: compute off-diagonal d_{ij}^n:

for $i = 1, \ldots, \mathcal{N}$ **do**

  for $j \in \mathcal{I}(i)$, $j \neq i$ **do**

    $d_{ij}^n \leftarrow \max\left(\tilde{\lambda}_{\max}(\boldsymbol{n}_{ij}, \boldsymbol{U}_i^n, \boldsymbol{U}_j^n)\,|\boldsymbol{c}_{ij}|\,,\ \tilde{\lambda}_{\max}(\boldsymbol{n}_{ji}, \boldsymbol{U}_j^n, \boldsymbol{U}_i^n)\,|\boldsymbol{c}_{ji}|\right)$

// Step 2: compute d_{ii}^n and τ_n:

$\tau_n \leftarrow +\infty$

for $i = 1, \ldots, \mathcal{N}$ **do**

  $d_{ii}^n \leftarrow -\sum_{j\in\mathcal{I}(i), j\neq i} d_{ij}^n, \qquad \tau_n \leftarrow \min\left(\tau_n\,,\ -c_{\mathrm{cfl}}\frac{m_i}{2d_{ii}^n}\right)$

// Step 3: perform update

for $i = 1, \ldots, \mathcal{N}$ **do**

  for $j \in \mathcal{I}(i)$ **do**

    $\boldsymbol{U}_i^{n+1} \leftarrow \mathbf{U}_i^n + \frac{\tau_n}{m_i}\left(\sum_{j\in\mathcal{I}(i)} \mathbb{f}\left(\mathbf{U}_j^n\right)\cdot \mathbf{c}_{ij} - \sum_{j\in\mathcal{I}(i)} d_{ij}^n \mathbf{U}_j^n\right)$

// Step 4: MPI synchronization

$\boldsymbol{U}$.update_ghost_values()

step-69

Parallelization approach (step-69)

- MPI parallelization via
  `parallel::distributed::Triangulation`
  `LinearAlgebra::distributed::Vector`
- Precompute m_i and c_{ij}.
  (This is the only place where finite elements enter)
- Distribute work onto threads with
  `parallel::apply_to_subranges()`
- Avoids global to (MPI) local index translations.
- (Asynchronous IO.)

```
const auto on_subranges =
  [&](
    std_cxx20::ranges::iota_view<unsigned int, unsigned int>::iterator i1,
    const std_cxx20::ranges::iota_view<unsigned int,
                                       unsigned int>::iterator i2) {

    for (const auto i :
         std_cxx20::ranges::iota_view<unsigned int, unsigned int>(*i1,
                                                                  *i2))
      {
        const auto U_i = gather(U, i);

        for (auto jt = sparsity.begin(i); jt != sparsity.end(i); ++jt)
          {
            const auto j = jt->column();

            if (j >= i)
              continue;

            const auto U_j = gather(U, j);

            const auto   n_ij = gather_get_entry(nij_matrix, jt);
            const double norm = get_entry(norm_matrix, jt);

            const auto lambda_max =
              ProblemDescription<dim>::compute_lambda_max(U_i, U_j, n_ij);

            double d = norm * lambda_max;

            if (boundary_normal_map.count(i) != 0 &&
                boundary_normal_map.count(j) != 0)
              {
                const auto n_ji = gather(nij_matrix, j, i);
                const auto lambda_max_2 =
                  ProblemDescription<dim>::compute_lambda_max(U_j,
                                                              U_i,
                                                              n_ji);

                const double norm_2 = norm_matrix(j, i);

                d = std::max(d, norm_2 * lambda_max_2);
              }

            set_entry(dij_matrix, jt, d);
            dij_matrix(j, i) = d;
          }
      }
  };

parallel::apply_to_subranges(indices_relevant.begin(),
                             indices_relevant.end(),
                             on_subranges,
                             4096);
```

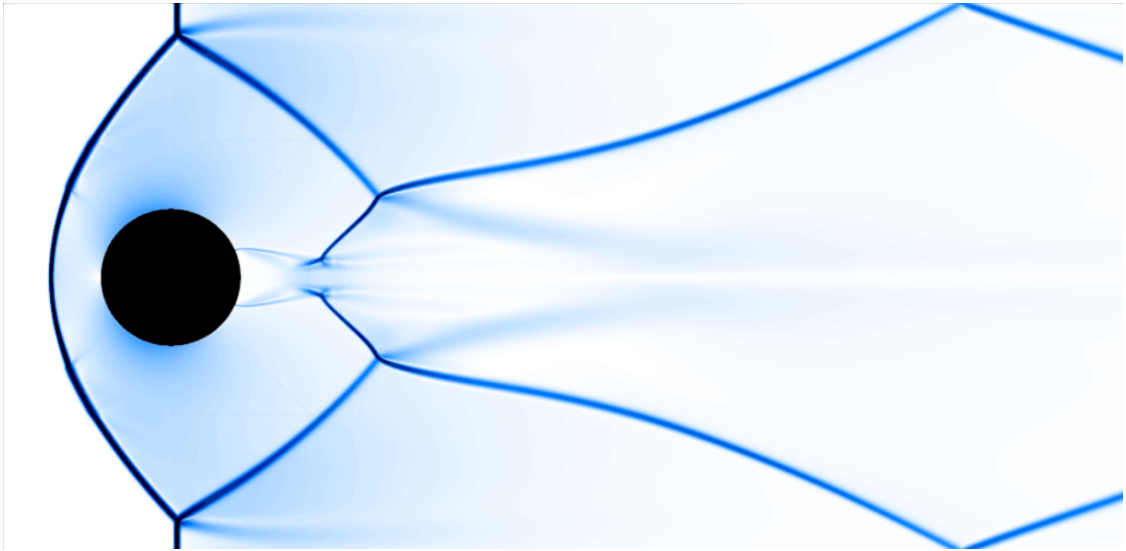
MPI local index translations:

```
{
  TimerOutput::Scope scope(
    computing_timer,
    "offline_data - create sparsity pattern and set up matrices");


  DynamicSparsityPattern dsp(n_locally_relevant, n_locally_relevant);

  const auto dofs_per_cell = discretization->finite_element.dofs_per_cell;
  std::vector<types::global_dof_index> dof_indices(dofs_per_cell);

  for (const auto &cell : dof_handler.active_cell_iterators())
    {
      if (cell->is_artificial())
        continue;

      /* We transform the set of global dof indices on the cell to the
       * corresponding "local" index range on the MPI process: */
      cell->get_dof_indices(dof_indices);
      std::transform(dof_indices.begin(),
                     dof_indices.end(),
                     dof_indices.begin(),
                     [&](types::global_dof_index index) {
                       return partitioner->global_to_local(index);
                     });

      /* And simply add, for each dof, a coupling to all other "local"
       * dofs on the cell: */
      for (const auto dof : dof_indices)
        dsp.add_entries(dof, dof_indices.begin(), dof_indices.end());
    }

  sparsity_pattern.copy_from(dsp);

  lumped_mass_matrix.reinit(sparsity_pattern);
  norm_matrix.reinit(sparsity_pattern);
  for (auto &matrix : cij_matrix)
    matrix.reinit(sparsity_pattern);
  for (auto &matrix : nij_matrix)
    matrix.reinit(sparsity_pattern);
}
```

```
template <std::size_t k>
DEAL_II_ALWAYS_INLINE inline Tensor<1, k>
gather(const std::array<LinearAlgebra::distributed::Vector<double>, k> &U,
       const unsigned int                                               i)
{
  Tensor<1, k> result;
  for (unsigned int j = 0; j < k; ++j)
    result[j] = U[j].local_element(i);
  return result;
}
```

Asynchronous IO:

```
auto data_out = std::make_shared<DataOut<dim>>();

data_out->attach_dof_handler(offline_data.dof_handler);

const auto &component_names = ProblemDescription<dim>::component_names;

for (unsigned int i = 0; i < problem_dimension; ++i)
  data_out->add_data_vector(output_vector[i], component_names[i]);

data_out->add_data_vector(schlieren_postprocessor.schlieren,
                          "schlieren_plot");

data_out->build_patches(discretization.mapping,
                        discretization.finite_element.degree - 1);

const auto output_worker = [this, name, t, cycle, checkpoint, data_out]() {
  if (checkpoint)
    {
      const unsigned int i =
        discretization.triangulation.locally_owned_subdomain();
      std::string filename =
        name + "-checkpoint-" + Utilities::int_to_string(i, 4) + ".archive";

      std::ofstream file(filename, std::ios::binary | std::ios::trunc);

      boost::archive::binary_oarchive oa(file);
      oa << t << cycle;
      for (const auto &it1 : output_vector)
        for (const auto &it2 : it1)
          oa << it2;
    }

  DataOutBase::VtkFlags flags(t,
                              cycle,
                              true,
                              DataOutBase::VtkFlags::best_speed);
  data_out->set_flags(flags);

  data_out->write_vtu_with_pvtu_record(
    "", name + "-solution", cycle, mpi_communicator, 6);
};

if (asynchronous_writeback)
  {
#def DEAL_II_WITH_THREADS
    background_thread_state = std::async(std::launch::async, output_worker);
#se
    AssertThrow(
      false,
      ExcMessage(
        "\"asynchronous_writeback\" was set to true but deal.II was built "
        "without thread support (\"DEAL_II_WITH_THREADS=false\")."));
#dif
```



. . . and beyond

. . . and beyond

$$m_i \left(\boldsymbol{U}_i^{L,n+1} - \boldsymbol{U}_i^n \right) = \tau_n \sum_{j \in \mathcal{I}(i)} \Big(- \mathbb{f}(\boldsymbol{U}_j^n) \cdot \boldsymbol{c}_{ij} + d_{ij}^{L,n} \big(\boldsymbol{U}_j^n - \boldsymbol{U}_i^n \big) \Big).$$

$$\sum_{j \in \mathcal{I}(i)} m_{ij} \big(\boldsymbol{U}_j^{H,n+1} - \boldsymbol{U}_j^n \big) = \tau_n \sum_{j \in \mathcal{I}(i)} \Big(- \mathbb{f}(\boldsymbol{U}_j^n) \cdot \boldsymbol{c}_{ij} + d_{ij}^{L,n} \frac{\alpha_i^n + \alpha_j^n}{2} \big(\boldsymbol{U}_j^n - \boldsymbol{U}_i^n \big) \Big),$$

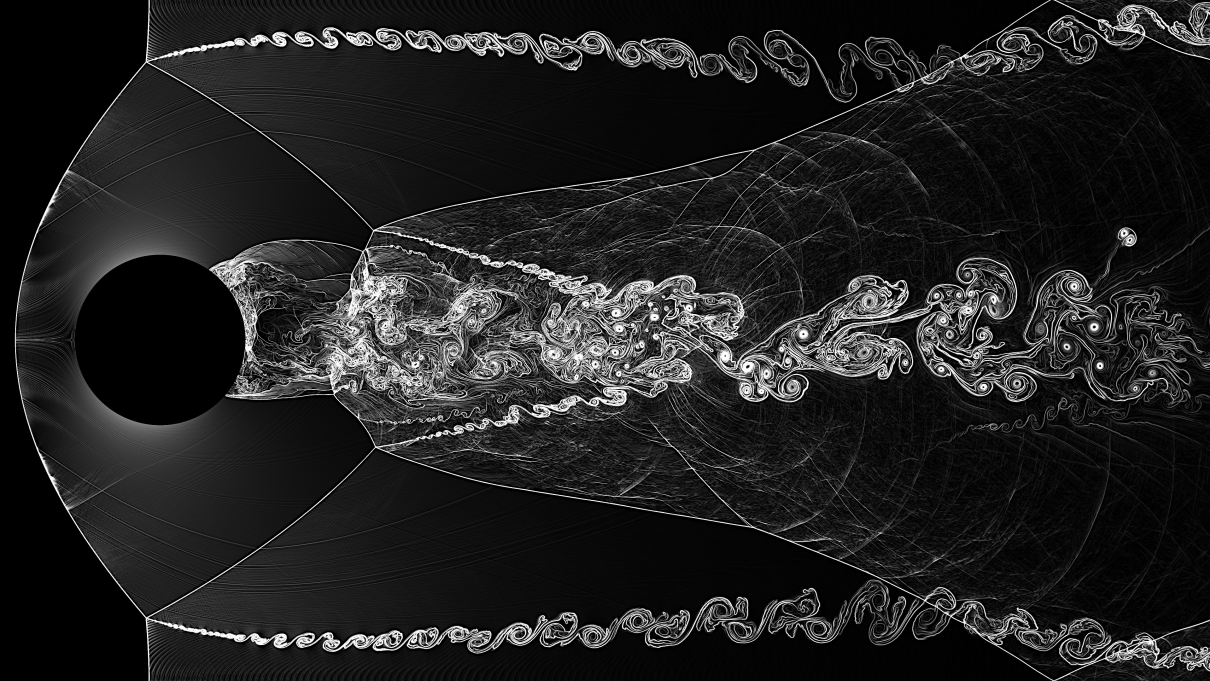
with a suitable indicator α_i.

- The low-order $\boldsymbol{U}_i^{L,n+1}$ is robust, the high-order $\boldsymbol{U}_i^{H,n+1}$ is not.
- Key idea: **Invariant-domain preserving convex limiting** [3]

$$\boldsymbol{U}_i^{H,n+1} - \boldsymbol{U}_i^{L,n+1} = \sum_{j \in \mathcal{I}(i)} \boldsymbol{P}_{ij}^n, \quad \text{where} \quad \boldsymbol{P}_{ij}^n = \dots$$

$$\boldsymbol{U}_i^{n+1} - \boldsymbol{U}_i^{L,n+1} = \sum_{j \in \mathcal{I}(i)} \ell_{ij}^n \, \boldsymbol{P}_{ij}^n, \quad 0 \le \ell_{ij}^n \le 1.$$

[3] Guermond, Nazarov, Popov, Tomas, *Second-order invariant domain preserving approximation of the Euler equations using convex limiting*, SIAM J. Sci. Comput. 40 (2018)



Coming soon. . .

https://github.com/conservation-laws/

Thank you for your attention!

2D 38M gridpoints: https://www.youtube.com/watch?v=xIwJZlsXpZ4
3D 1.8B gridpoints: https://www.youtube.com/watch?v=vBCRAF_c8m8

https://www.dealii.org/

https://github.com/conservation-laws/