A Matrix-Free Steady-State Solver for Eilmer4

Rowan Gollan

06 April 2017

Talk Outline:

- Introduction to solving steady flows
- Specific solution method for Eilmer4
- Demonstration cases

Update schemes in Eilmer4

explicit updates:

- Euler, predictor-corrector, Runge-Kutta 3 family
- time efficient for computing unsteady flows
- application: simulation of gas dynamics in hypersonic impulse facilities

implicit updates:

- can be constructed to give accelerated convergence to steady-state
- time efficient for computing steady flows
- applications:
  - + steady flow analysis of test articles in impulse facilities
  - + aerodynamic optimisation for hypersonic vehicles and inlets

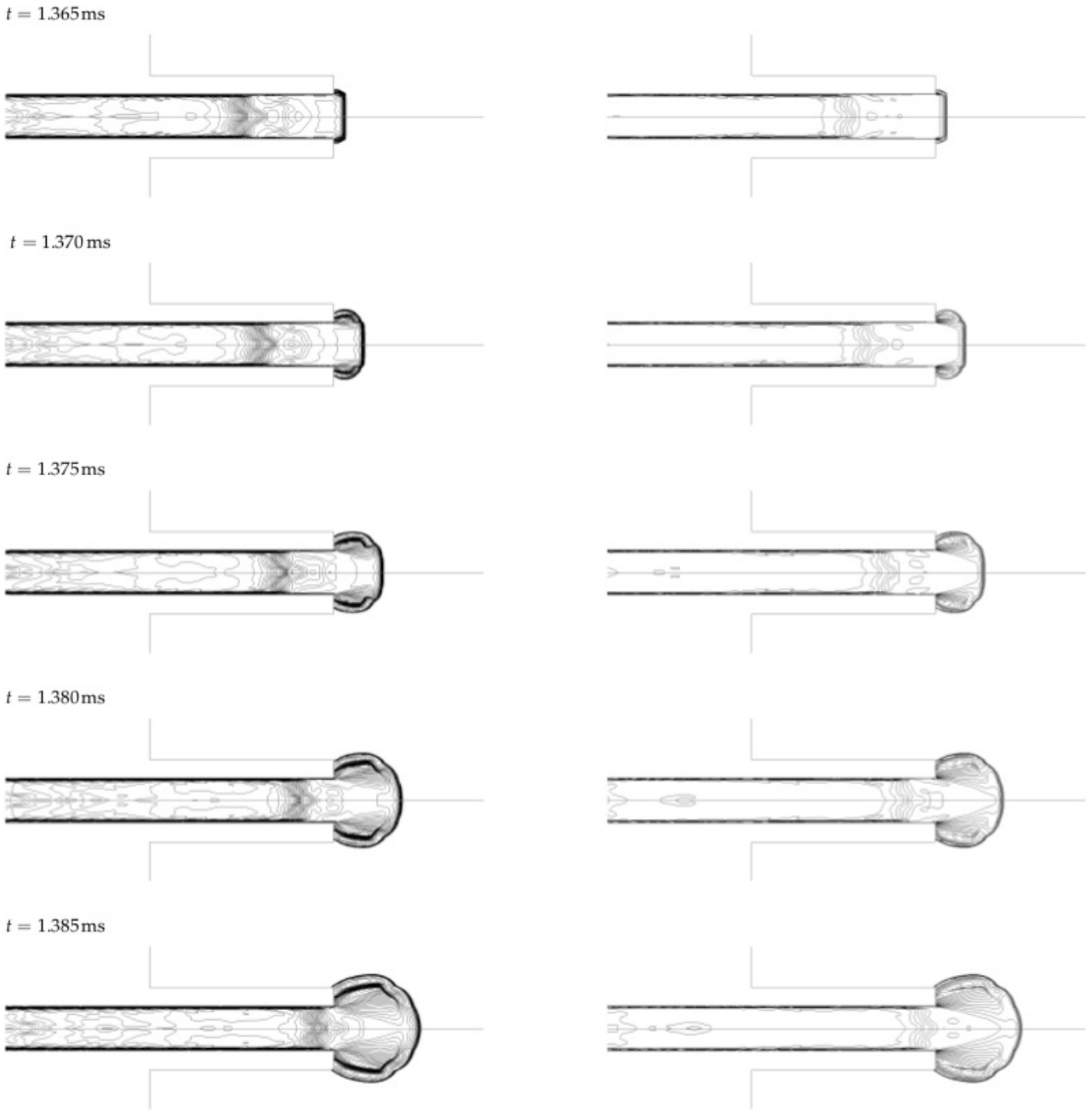
Update schemes in Eilmer4

explicit u

- Euler, predi
- time efficie
- application

implicit u

- can be cons
- time efficie
- applications
  - + steady f
  - + aerodyr

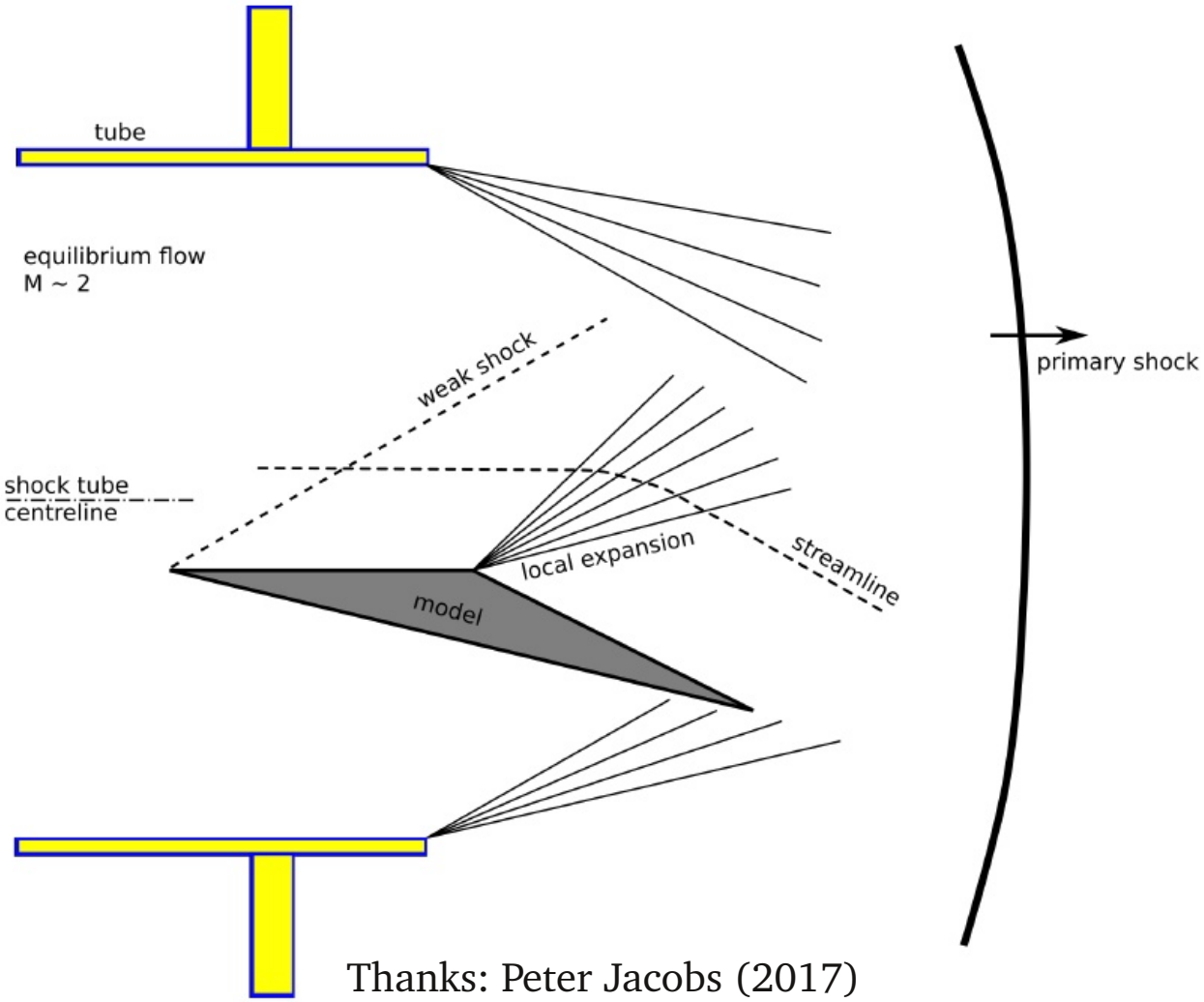


ılse facilities

ly-state

nlets

Update schemes in Eilmer4



Thanks: Peter Jacobs (2017)

System of compressible flow equations

Conservation of mass:

$$\frac{\partial}{\partial t}\rho = -\nabla \cdot \rho\mathbf{u}$$

Conservation of momentum:

$$\frac{\partial}{\partial t}\rho\mathbf{u} = -\nabla \cdot \rho\mathbf{u}\mathbf{u} - \nabla p - \nabla \cdot \left\{-\mu(\nabla\mathbf{u} + (\nabla\mathbf{u})^{\dagger}) + \tfrac{2}{3}\mu(\nabla \cdot \mathbf{u})\boldsymbol{\delta}\right\}$$

Conservation of total energy:

$$\frac{\partial}{\partial t}\rho E = -\nabla \cdot (e + \frac{p}{\rho})\mathbf{u} + \nabla \cdot [k\nabla T] + -\left(\nabla \cdot \left[\left\{-\mu(\nabla\mathbf{u} + (\nabla\mathbf{u})^{\dagger}) + \tfrac{2}{3}\mu(\nabla \cdot \mathbf{u})\boldsymbol{\delta}\right\} \cdot \mathbf{u}\right]\right)$$

collapse into compact notation, and let $\boldsymbol{F}$ *represent discretised system*

$$\frac{d\mathbf{u}}{dt} = \mathbf{F}(\mathbf{u})$$

Solution strategies to find a steady state

time-marching with explicit steps:

- follow transient solution time-accurately until steady state is reached
- can require many many steps
- timestep size is limited by stability constraints

time-marching with implicit steps:

Solve $\frac{d\mathbf{u}}{dt} = \mathbf{F}(\mathbf{u})$ with a sequence of cheap, approximate time steps.

- large timesteps permitted (CFL > 1); stability is greatly increased with implicit steps
- simplest example: backward Euler update scheme
- can require many iterations, but each iteration is relatively cheap

Newton iterations to solve steady-state system directly:

Set $\frac{d\mathbf{u}}{dt} = 0$, so solve $\mathbf{F}(\mathbf{u}) = 0$ with Newton's method.

- quadratic convergence <u>when</u> the guess $\mathbf{u}$ is in region of convergence
- typically far fewer iterations are required compared to time-marching, but each Newton iteration is relatively expensive

Selecting a method for Eilmer4... the Wish List

- able to treat **F(u)** as a black box
- good for high speed flows (ie. grids with high aspect ratio cells)
- works for both structured and unstructured grids
- avoid the need to derive and code implicit boundary conditions
- easily parallelized and scales well in parallel
- efficient on memory, particularly in 3D

Newton-Krylov method

Eilmer4's steady-state solver: Newton method

- Using Newton's method to solve the zero equation, F(u) = 0.
- These steps are called the outer iterations.

$$\text{Taylor expansion: } \mathbf{F}(\mathbf{u}^{k+1}) = \mathbf{F}(\mathbf{u}) + \mathbf{F}'(\mathbf{u}^k)\left[\mathbf{u}^{k+1} - \mathbf{u}^k\right] + \text{h.o.t}$$

- Set $\mathbf{F}(\mathbf{u}^{k+1}) = 0$
- Ignore higher order terms
- Notation substitution...

$$\mathbf{J}(\mathbf{u}^k)\Delta\mathbf{u}^k = -\mathbf{F}(\mathbf{u}^k), \quad \mathbf{u}^{k+1} = \mathbf{u}^k + \Delta\mathbf{u}^k$$

0. Given $\mathbf{u}^0$
1. Repeat for $k = 0$,..., good enough:

    1a. Solve $\mathbf{J}(\mathbf{u}^k)\Delta\mathbf{u}^k = -\mathbf{F}(\mathbf{u}^k)$
    1b. $\mathbf{u}^{k+1} = \mathbf{u}^k + \Delta\mathbf{u}^k$

Historical note:

*The Jacobian, **J**, is named after Peter Jacobs who discovered this mathematical structure as a teenager while dabbling around with CFD algorithms. At the time of his discovery, he noted: the "I have discovered a truly remarkable matrix that will speed up all CFD calculations, but alas the code to implement this will not fit in the margin of this workbook." Ever since, CFD coders have tried to write that code, and Peter Jacobs walked away from Jacobian matrices.*

Eilmer4's steady-state solver: Newton method

- Using Newton's method to solve the zero equation, F(u) = 0.
- These steps are called the outer iterations.

$$\text{Taylor expansion: } \mathbf{F}(\mathbf{u}^{k+1}) = \mathbf{F}(\mathbf{u}) + \mathbf{F}'(\mathbf{u}^k)\left[\mathbf{u}^{k+1} - \mathbf{u}^k\right] + \text{h.o.t}$$

- Set $\mathbf{F}(\mathbf{u}^{k+1}) = 0$
- Ignore higher order terms
- Notation substitution...

$$\mathbf{J}(\mathbf{u}^k)\Delta\mathbf{u}^k = -\mathbf{F}(\mathbf{u}^k), \quad \mathbf{u}^{k+1} = \mathbf{u}^k + \Delta\mathbf{u}^k$$

0. Given $\mathbf{u}^0$
1. Repeat for $k = 0$,..., good enough:

    1a. Solve $\mathbf{J}(\mathbf{u}^k)\Delta\mathbf{u}^k = -\mathbf{F}(\mathbf{u}^k)$
    1b. $\mathbf{u}^{k+1} = \mathbf{u}^k + \Delta\mathbf{u}^k$

Historical note, corrected:
*The Jacobian, **J**, is named after Carl Gustav Jacob Jacobi (1804-1851), a German mathematician.*

Eilmer4's steady-state solver: Krylov iterative solver

Solving the linear system $\mathbf{J}(\mathbf{u}^k)\Delta\mathbf{u}^k = -\mathbf{F}(\mathbf{u}^k) \rightarrow \mathbf{Ax} = \mathbf{b}$

- The solution to the matrix equation is solved by an iterative process.
- In particular, the Generalized Minimal RESidual (GMRES) method is used
- On each iteration, a matrix-vector product builds a new trial vector
- The final solution vector is formed from a linear combination of the trial vectors
- The GMRES method is not restricted based on the type of matrix. This makes it good for a wide class of linear systems that results from discretizing PDEs.

Eilmer4's steady-state solver: Jacobian-free algorithm

Look Ma, no hands! Or
How to solve a matrix equation without ever forming the matrix

ALGORITHM **6.9** *GMRES*

1. *Compute* $r_0 = b - Ax_0$, $\beta := \|r_0\|_2$, *and* $v_1 := r_0/\beta$
2. *For* $j = 1, 2, \ldots, m$ *Do:*
3. *Compute* $w_j := Av_j$
4. *For* $i = 1, \ldots, j$ *Do:*
5. $h_{ij} := (w_j, v_i)$
6. $w_j := w_j - h_{ij}v_i$
7. *EndDo*
8. $h_{j+1,j} = \|w_j\|_2$. *If* $h_{j+1,j} = 0$ *set* $m := j$ *and go to 11*
9. $v_{j+1} = w_j/h_{j+1,j}$
10. *EndDo*
11. *Define the* $(m+1) \times m$ *Hessenberg matrix* $\bar{H}_m = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$.
12. *Compute* y_m *the minimizer of* $\|\beta e_1 - \bar{H}_m y\|_2$ *and* $x_m = x_0 + V_m y_m$.

Source: Saad (2003)

Eilmer4's steady-state solver: Jacobian-free algorithm

Look Ma, no hands! Or
How to solve a matrix equation without ever forming the matrix

ALGORITHM **6.9** *GMRES*

1. Compute $r_0 = b - Ax_0$, $\beta := \|r_0\|_2$, and $v_1 := r_0/\beta$
2. For $j = 1, 2, \ldots, m$ Do:
3. Compute $w_j := Av_j$
4. For $i = 1, \ldots, j$ Do:
5. $h_{ij} := (w_j, v_i)$
6. $w_j := w_j - h_{ij}v_i$
7. EndDo
8. $h_{j+1,j} = \|w_j\|_2$. If $h_{j+1,j} = 0$ set $m := j$ and go to 11
9. $v_{j+1} = w_j/h_{j+1,j}$
10. EndDo
11. Define the $(m+1) \times m$ Hessenberg matrix $\bar{H}_m = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$.
12. Compute y_m the minimizer of $\|\beta e_1 - \bar{H}_m y\|_2$ and $x_m = x_0 + V_m y_m$.

- The matrix only ever appears as a matrix-vector product in Krylov algorithms.
- The result of the matrix-vector product is a vector.
- Use the Frechet derivative to find compute the vector result directly without forming the matrix.

Source: Saad (2003)

$$\mathbf{J}\mathbf{v} = \left[\mathbf{F}(\mathbf{u} + \epsilon\mathbf{v}) - \mathbf{F}(\mathbf{u})\right]/\epsilon$$

Using some mathematical black magic, we can treat F(u) as a black box. This is a big win in hypersonic flows where we have to deal with many physcial models.

Eilmer4's steady-state solver: refinements to the algorithm

What I've described so far is the purest form of a Jacobian-free Newton-Krylov (JFNK) method. In practice, we need to refine the algorithm to get a good blend of robustness and computational efficiency.

1st-order spatial reconstruction during start-up:

- increases robustness to use 1st-order in space early, especially while shock positions are establishing
- also helps push the solution towards the region of convergence

pseudo-transient continuation:

- principally, this is a robustness tweak.
- the Newton method can find itself trapped in local valleys.
- pseudo-transient continuation allows us to follow physical transients in an approximate way, which lets the solution "climb hills" in the search for the zero
- the Jacocian is augmented with a time term

$$\mathbf{J} = \frac{1}{\tau} + \frac{\partial \mathbf{F}}{\partial \mathbf{u}}$$

- as convergence drops, timestep is increased. Eventually, 1/tau term drops out.

Eilmer4's steady-state solver: refinements to the algorithm

an inexact Newton method:

- again, when far from the region of convergence it does not pay to solve for the Newton step precisely. It leads to "oversolving" and conergence can stall.
- the Newton step gives you an idea of which direction to move in. Early on in the iterations, it is not a very good estimate of where to move, so why then spend computational effort exactly solving for a poor estimate.
- in the inexact method, a loose tolerance on linear system convergence is used early on. This tolerance is tightened as the region of convergence is approached.

restarted GMRES

- to save on memory, the number of trial vectors in GMRES is capped at a certain number
- when the linear iterative solve does not achieve convergence before filling the memory allocated for all trial vectors, then it restarts with the best guess of the solution as the new starting point
- this degrades convergence, but it allows problems to fit in memory that you might not be able to tackle otherwise

Looking back at the Wish List

- able to treat **F(u)** as a black box
- good for high speed flows (ie. grids with high aspect ratio cells)
- works for both structured and unstructured grids
- avoid the need to derive and code implicit boundary conditions
- easily parallelized and scales well in parallel
- efficient on memory, particularly in 3D

Using a Krylov method, which only requires the result of a matrix-vector product, and the Frechet derivative means that an explicit Jacobian is never formed. This means I don't need to know about the details of how **F(u)** is constructed.

Looking back at the Wish List

- able to treat **F(u)** as a black box
- good for high speed flows (ie. grids with high aspect ratio cells)
- works for both structured and unstructured grids
- avoid the need to derive and code implicit boundary conditions
- easily parallelized and scales well in parallel
- efficient on memory, particularly in 3D

The Frechet derivative allows the solver to feel the "full" Jacobian. This is in contrast to methods that might approximate the Jacobian. This method makes no assumptions about the flow gradient directions and full information from the Jacobian is transmitted to the solution.

Looking back at the Wish List

- able to treat **F(u)** as a black box
- good for high speed flows (ie. grids with high aspect ratio cells)
- works for both structured and unstructured grids
- avoid the need to derive and code implicit boundary conditions
- easily parallelized and scales well in parallel
- efficient on memory, particularly in 3D

Using a Krylov method, which only requires the result of a matrix-vector product, and the Frechet derivative means that an explicit Jacobian is never formed. This means I don't need to know about the details of how **F(u)** is constructed.

Looking back at the Wish List

- able to treat **F(u)** as a black box
- good for high speed flows (ie. grids with high aspect ratio cells)
- works for both structured and unstructured grids
- avoid the need to derive and code implicit boundary conditions
- easily parallelized and scales well in parallel
- efficient on memory, particularly in 3D

The numerical differentiation involved with the Frechet derivative takes the hassle out of forming implicit boundary conditions.

Looking back at the Wish List

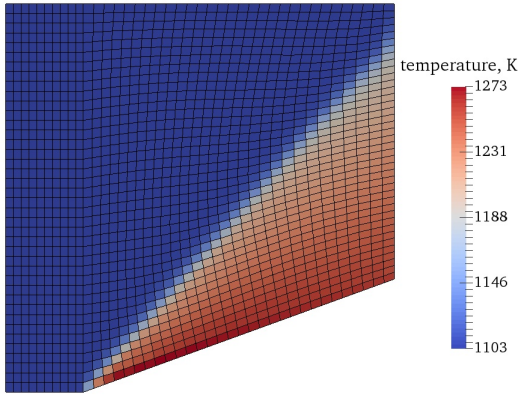
- able to treat **F(u)** as a black box
- good for high speed flows (ie. grids with high aspect ratio cells)
- works for both structured and unstructured grids
- avoid the need to derive and code implicit boundary conditions
- easily parallelized and scales well in parallel
- efficient on memory, particularly in 3D

The Newton-Krylov methods have been designed to scale well for use on large supercomputers. The present implentation was extended to shared-memory parallel mode with very little extra code.

Demonstration cases

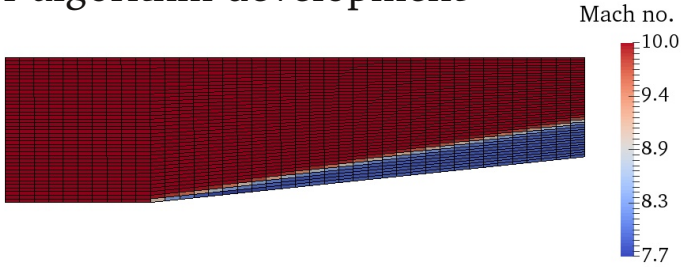
Mach 1.5 flow over a 20-deg cone

- axisymmetric, inviscid, low supersonic Mach number
- the classic "intro to eilmer" test case



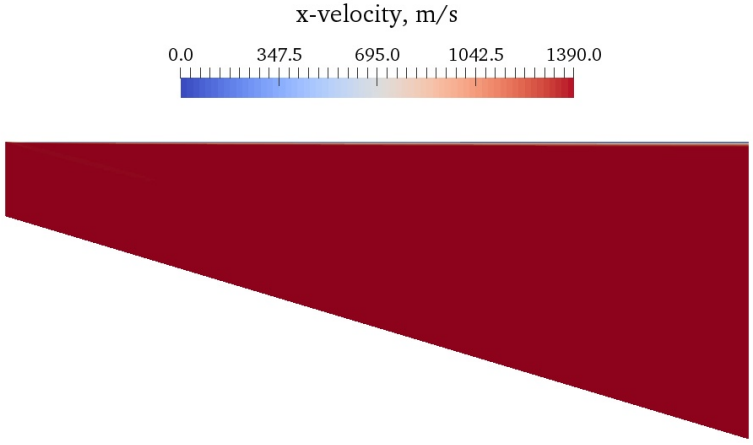
Mach 10 flow over a 6-deg wedge

- 2D planar, inviscid, high Mach number
- conditions relevant to SPARTAN flight
- principal test case for algorithm development



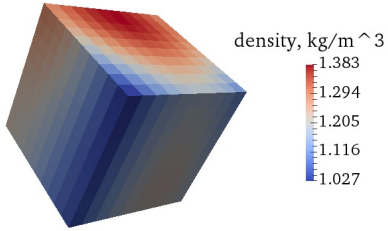
Mach 4 laminar flow over a flat plate

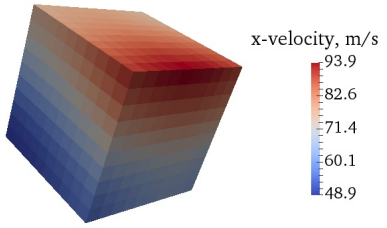
- 2D planar, viscous
- high aspect ratio cells



3D Method of Manufactured Solutions

- unstructured grid in 3D
- multiple block
- parallel
- viscous terms
- user-defined B.C.s
- user-defined source terms

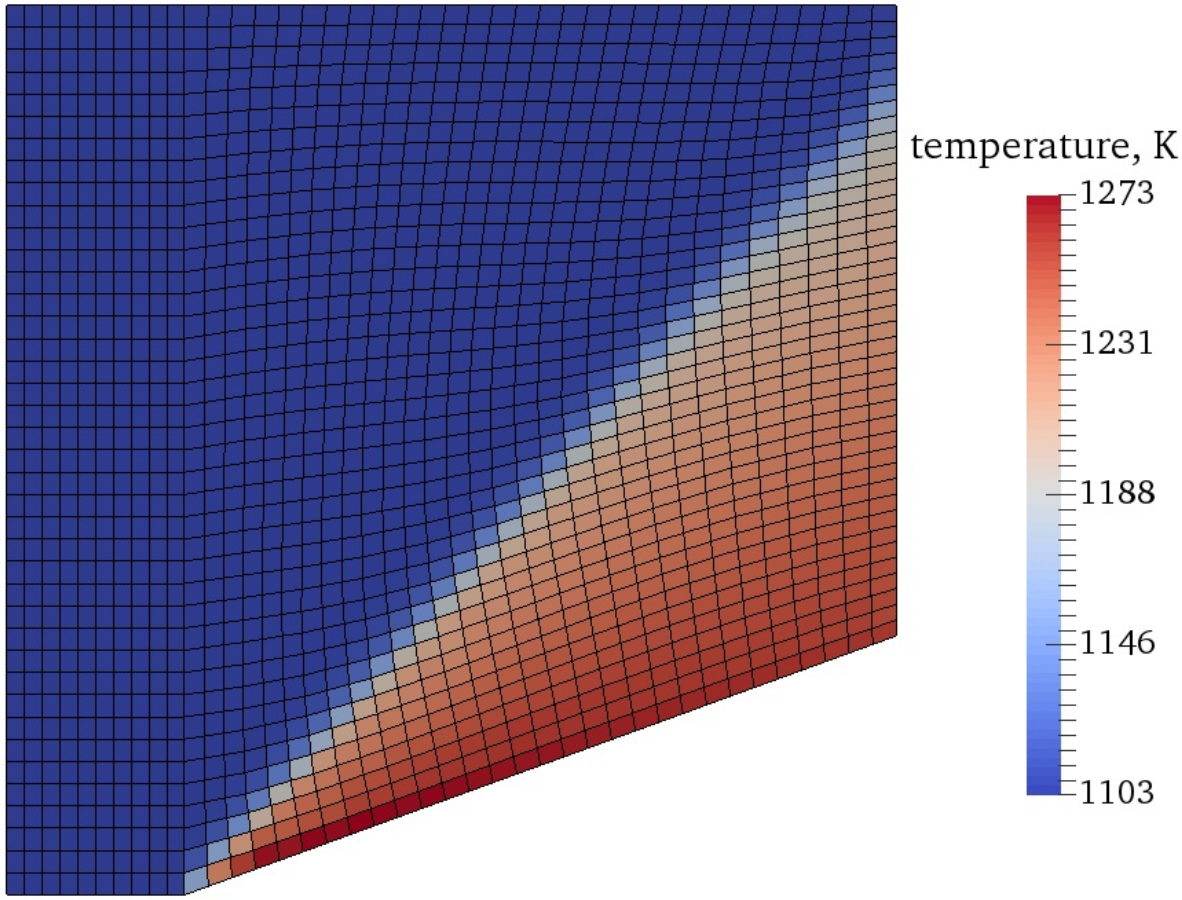




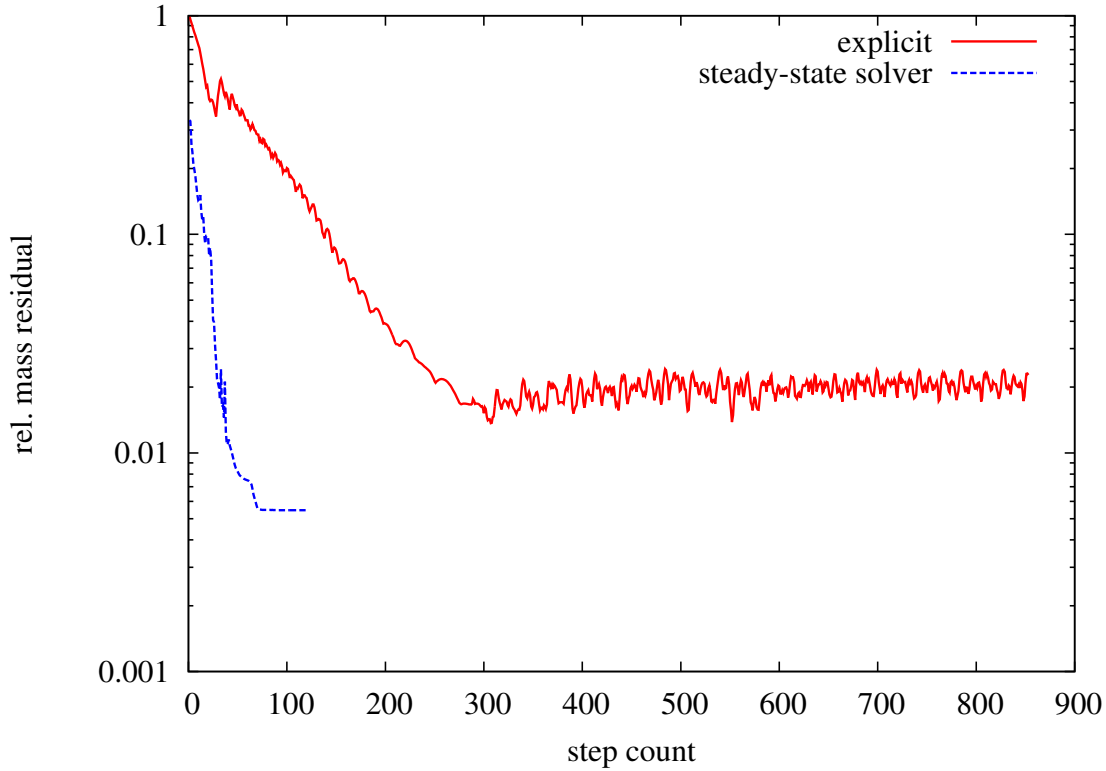
A note on residuals

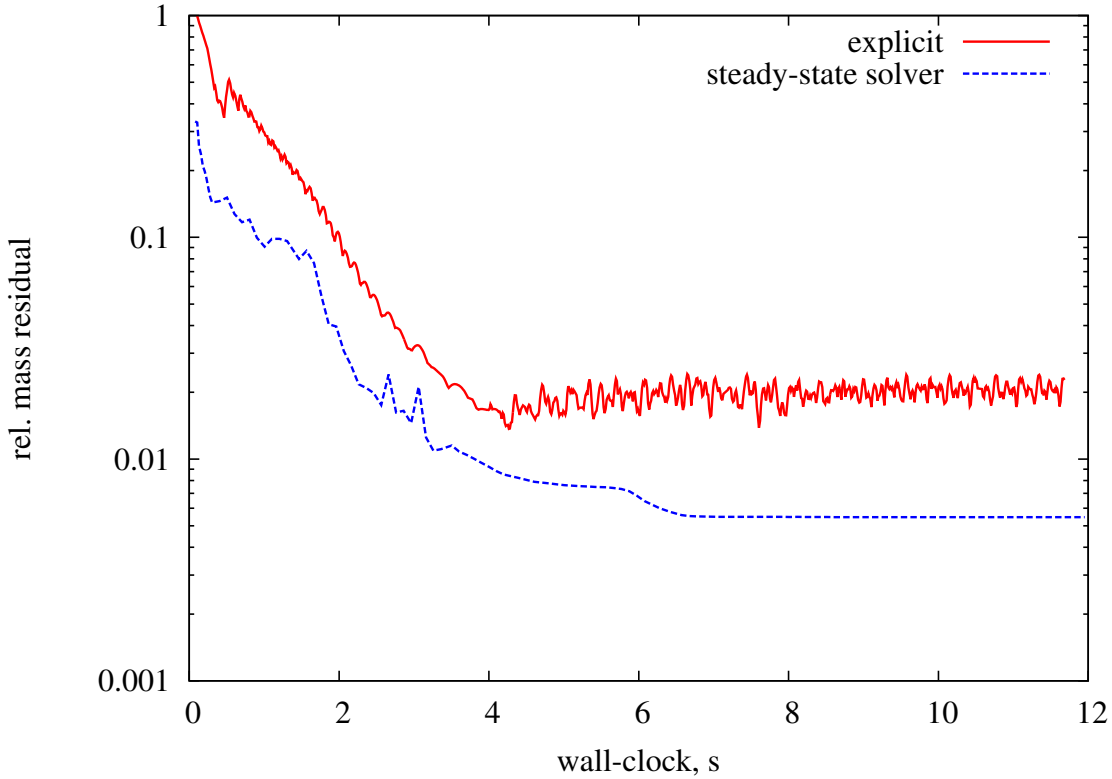
- Residuals give a measure of how well we have balanced the conservation equations
- The lower the residual value, the better we are doing at balancing (solving) the conservation equations
- There are many ways to report residuals.
- I will show mass residual in all these demo cases and report the worst case mass residual in the grid

$$\max\left(\left|\frac{\partial \rho_i}{\partial t}\right|\right)$$



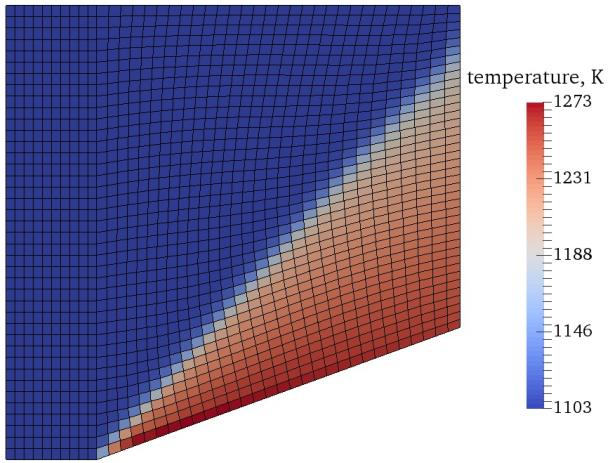
Mach 1.5 flow over 20-deg cone



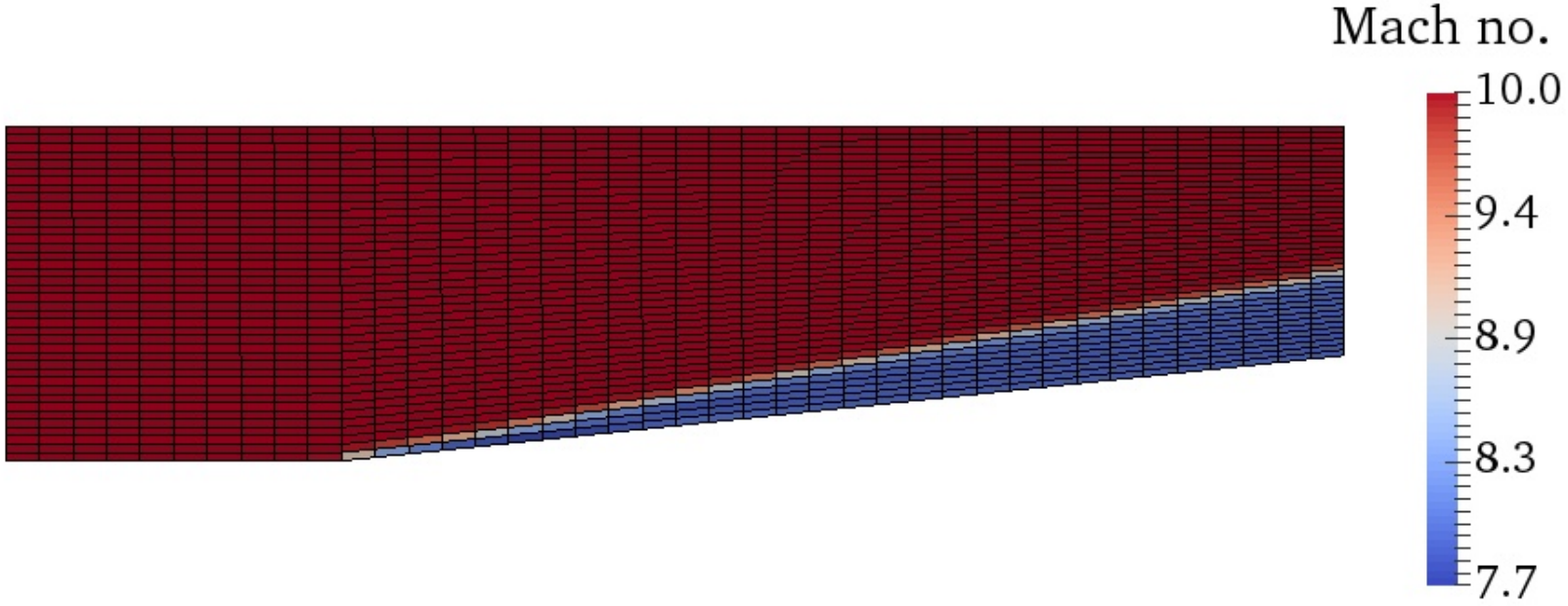


Notes:

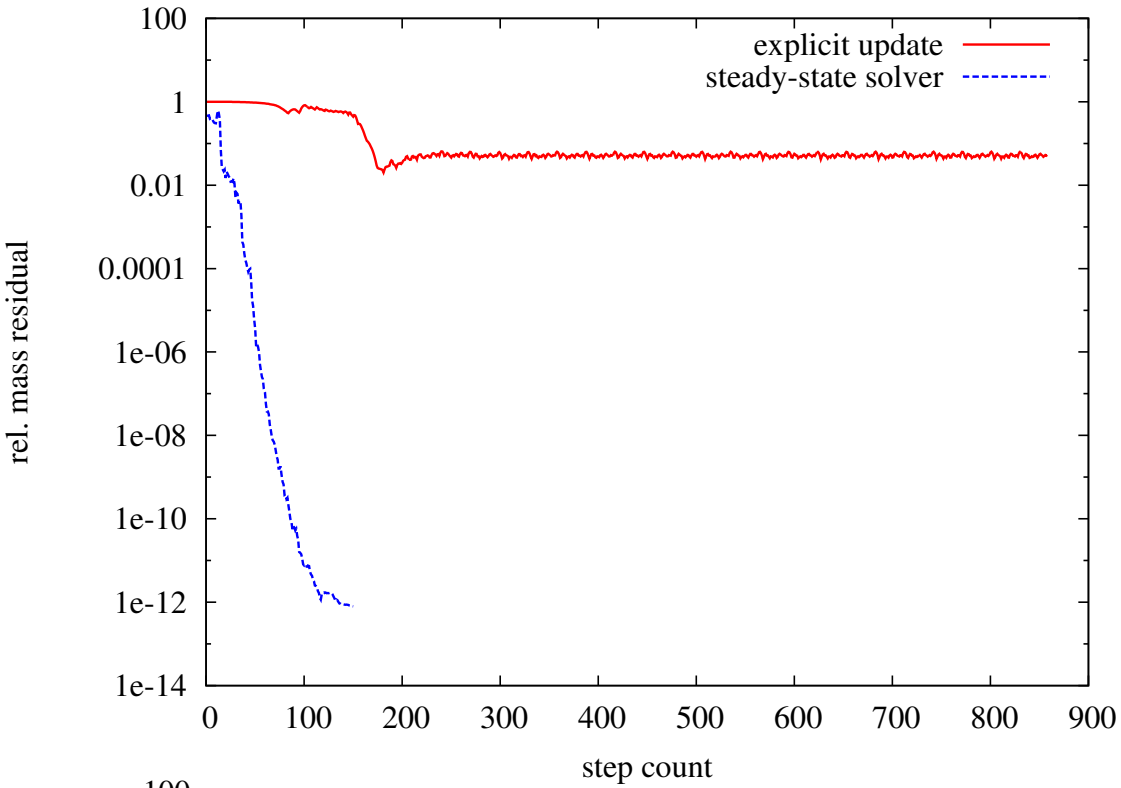
- Flux calculator: adaptive
- Geometry: axisymmetric
- Compute cores: 2 x Intel Core i3-4170

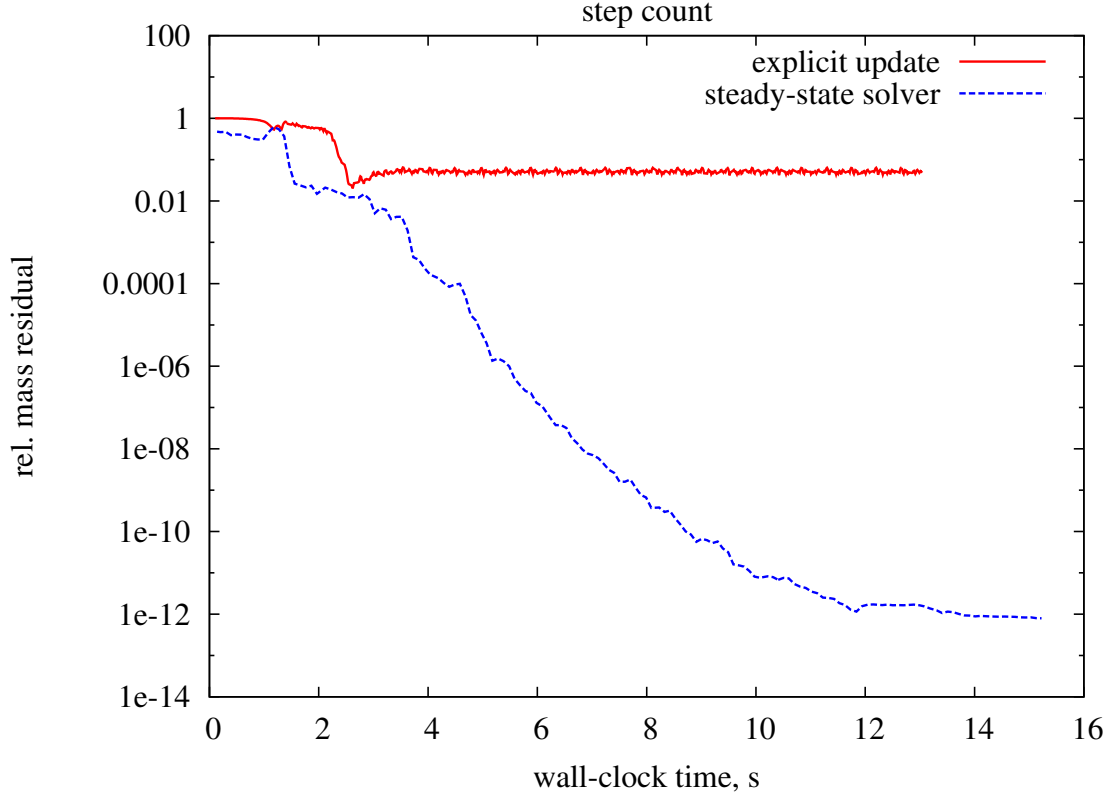


Mach 10 flow over a 6 deg wedge



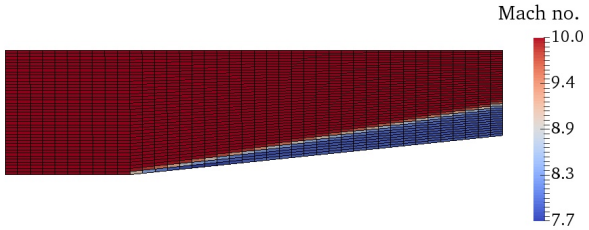
Mach 10 flow over a 6 deg wedge





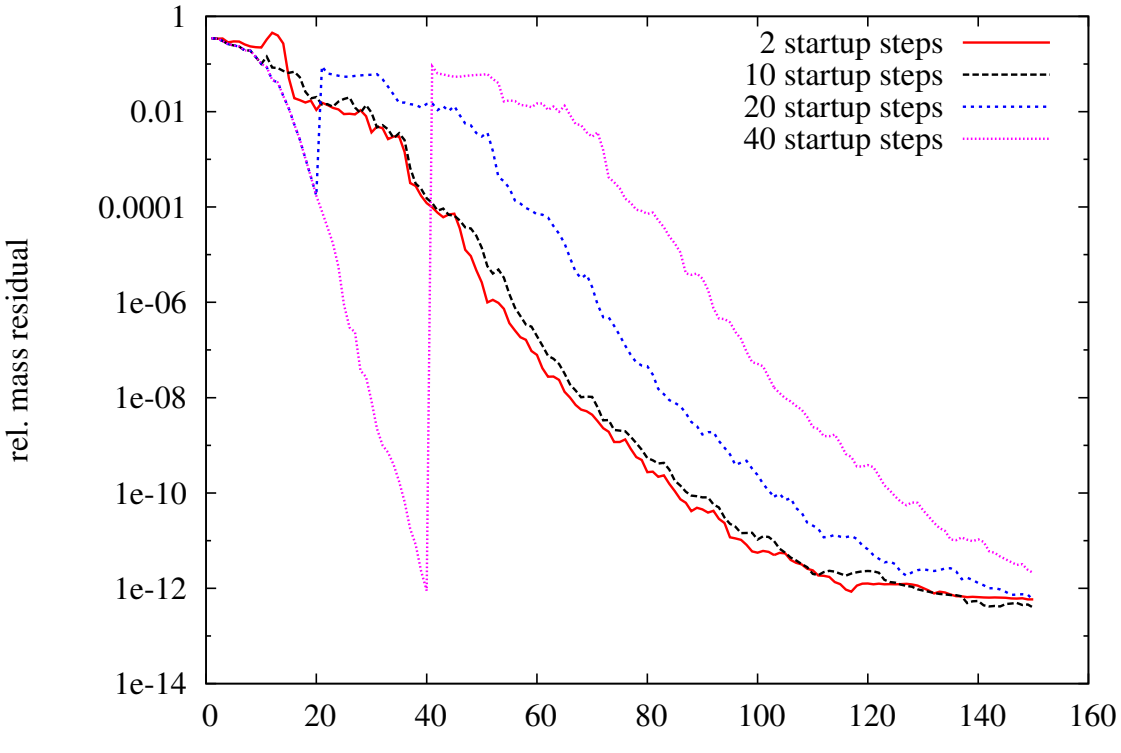
Notes:

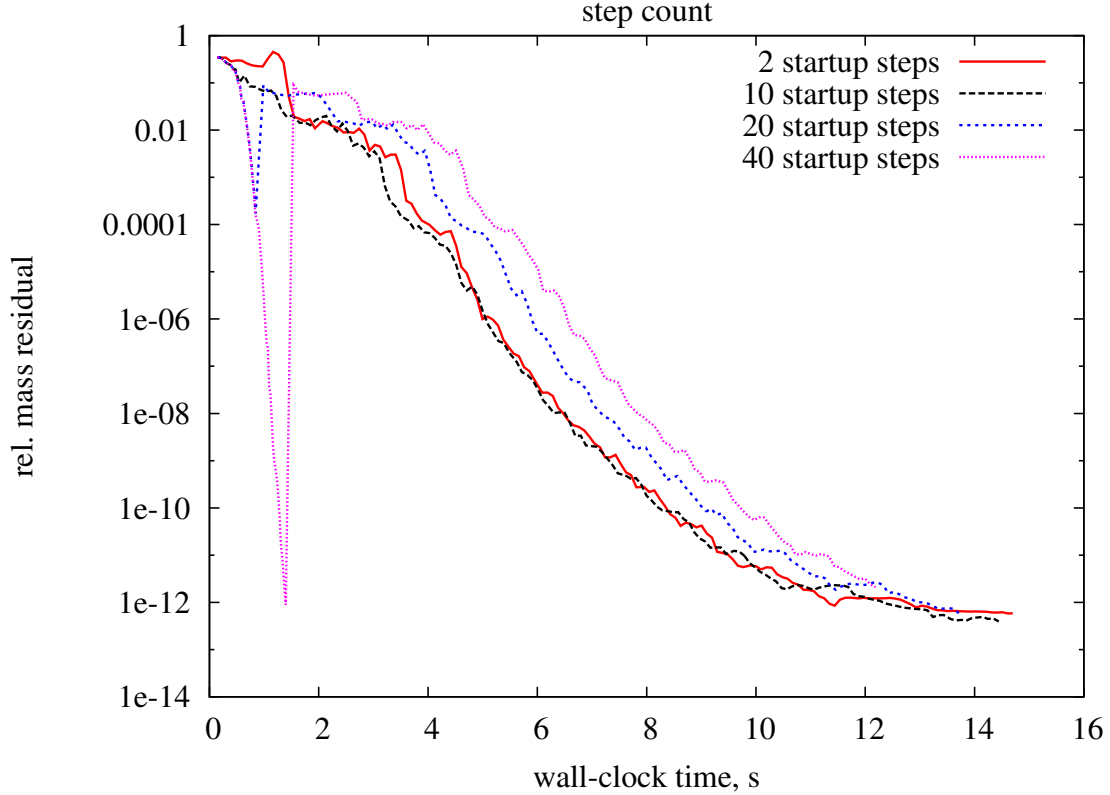
- Flux calculator: ausmdv
- Geometry: 2D planar
- Steady-state solver can achieve 12-orders of magnitude residual drop on relatively poor grid
- Explict updates stall early
- Compute core: 1 x Intel Core i3-4170



Mach 10 flow over a 6 deg wedge

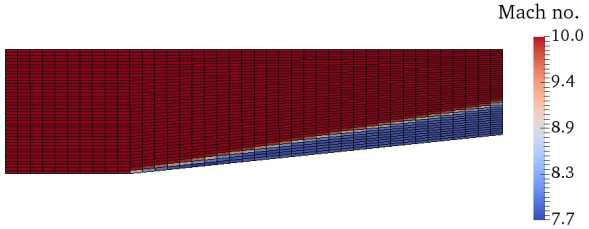
The effect of number of 1st-order startup steps





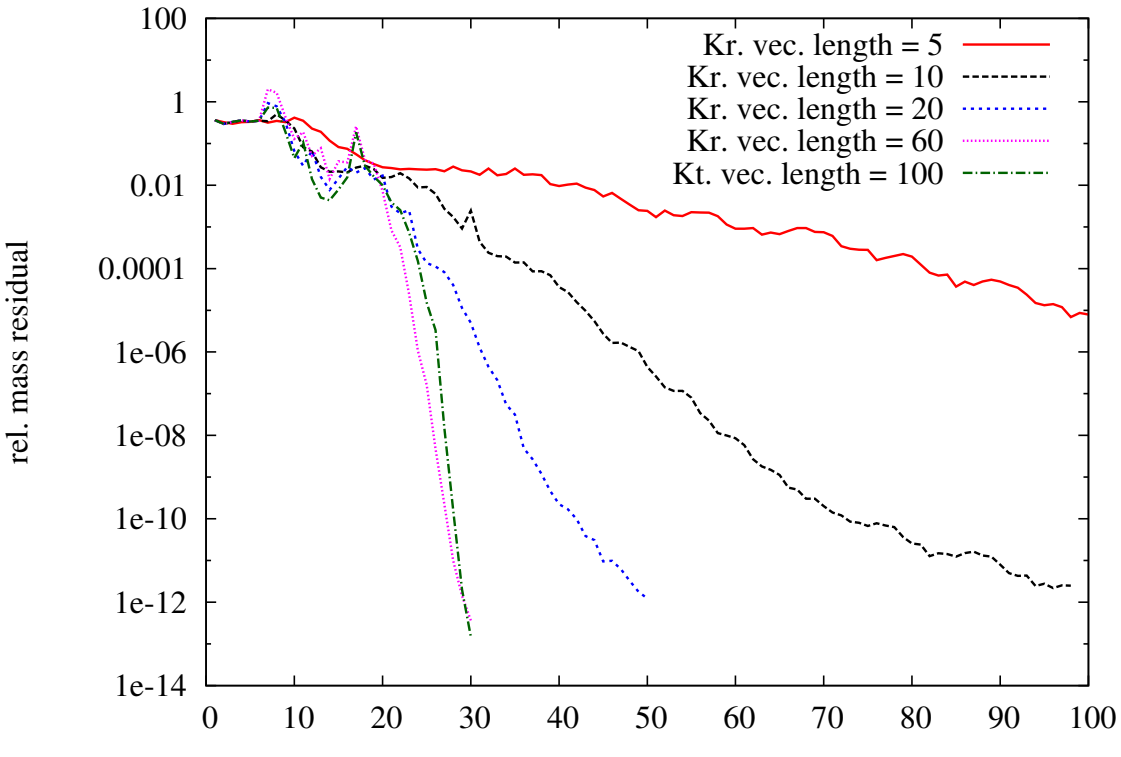
Notes:

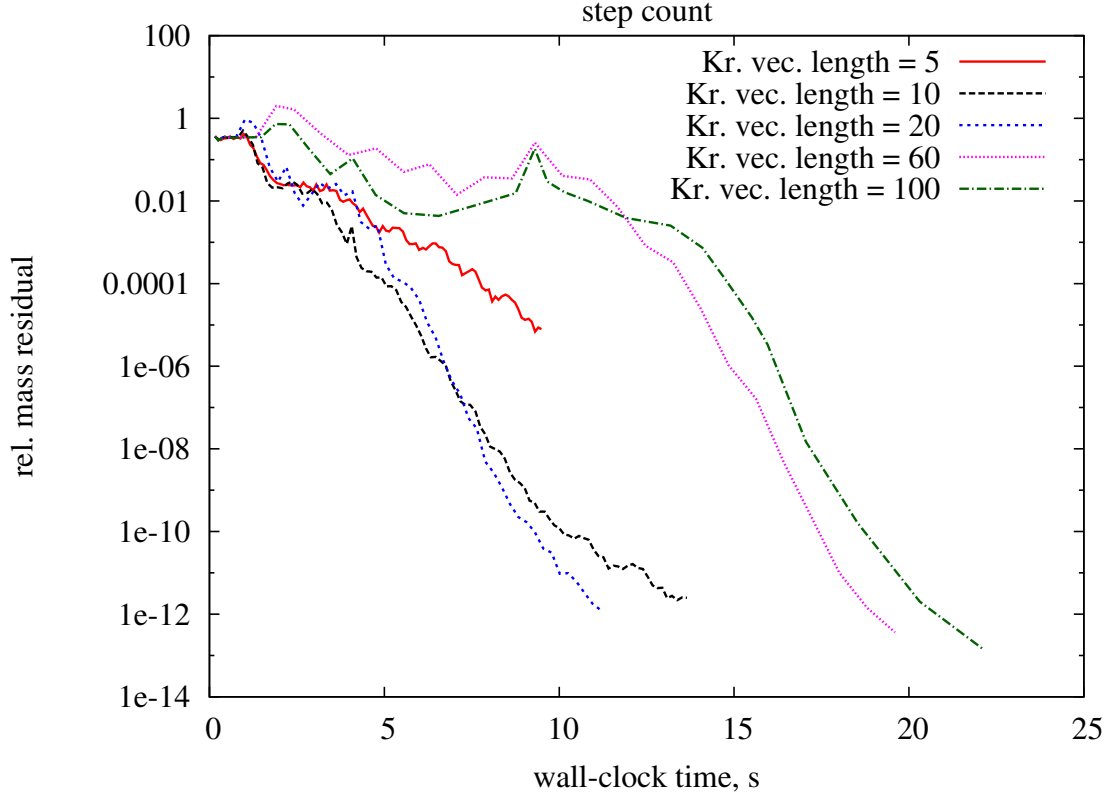
- This simple case is robust even without 1st-order steps
- 1st-order steps may help to push the solution towards region of convergence
- No benefit to going for deep convergence with 1st-order reconstruction



Mach 10 flow over a 6 deg wedge

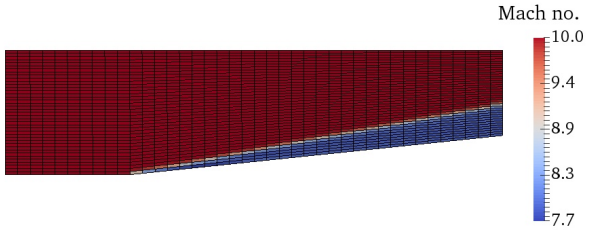
The effect of number of trial vectors



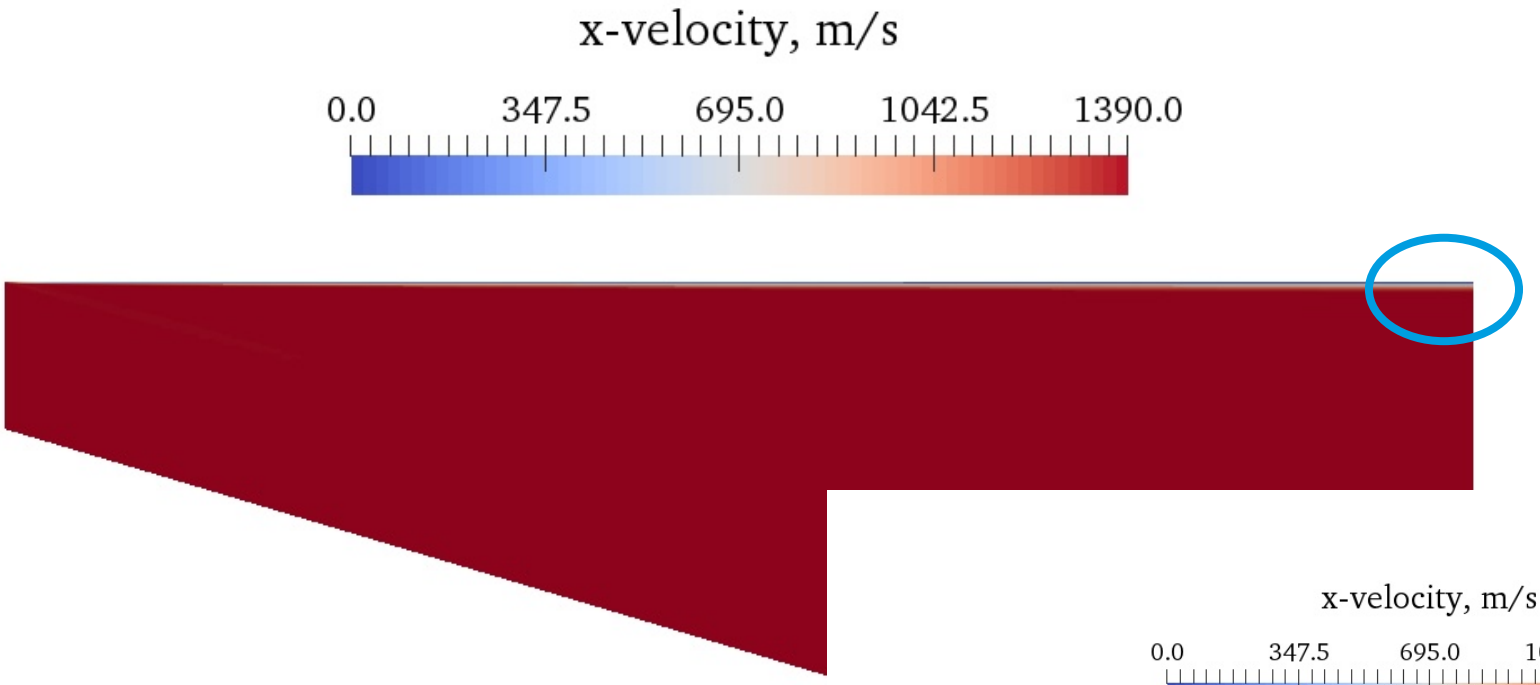


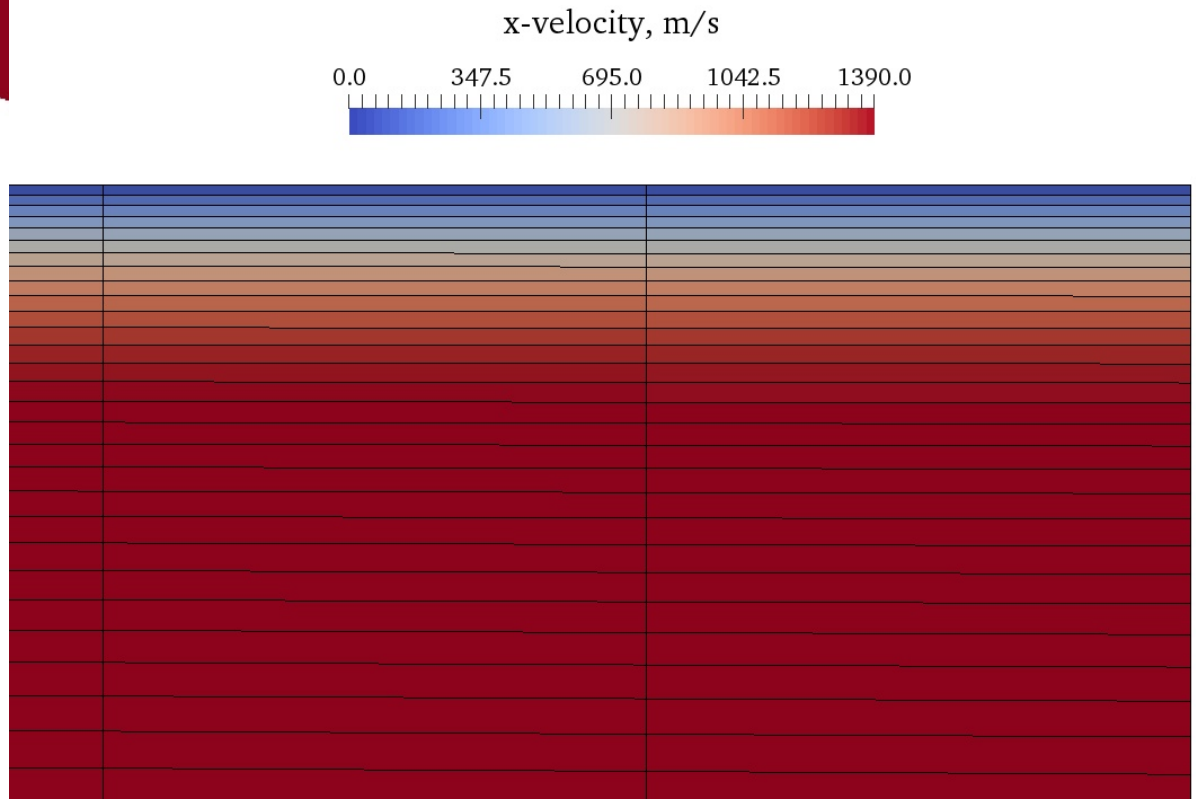
Notes:

- For large numbers of Krylov trial vectors, achieve "textbook" quadratic convergence.
- However, this is not a win in terms of wall-clock to final solution

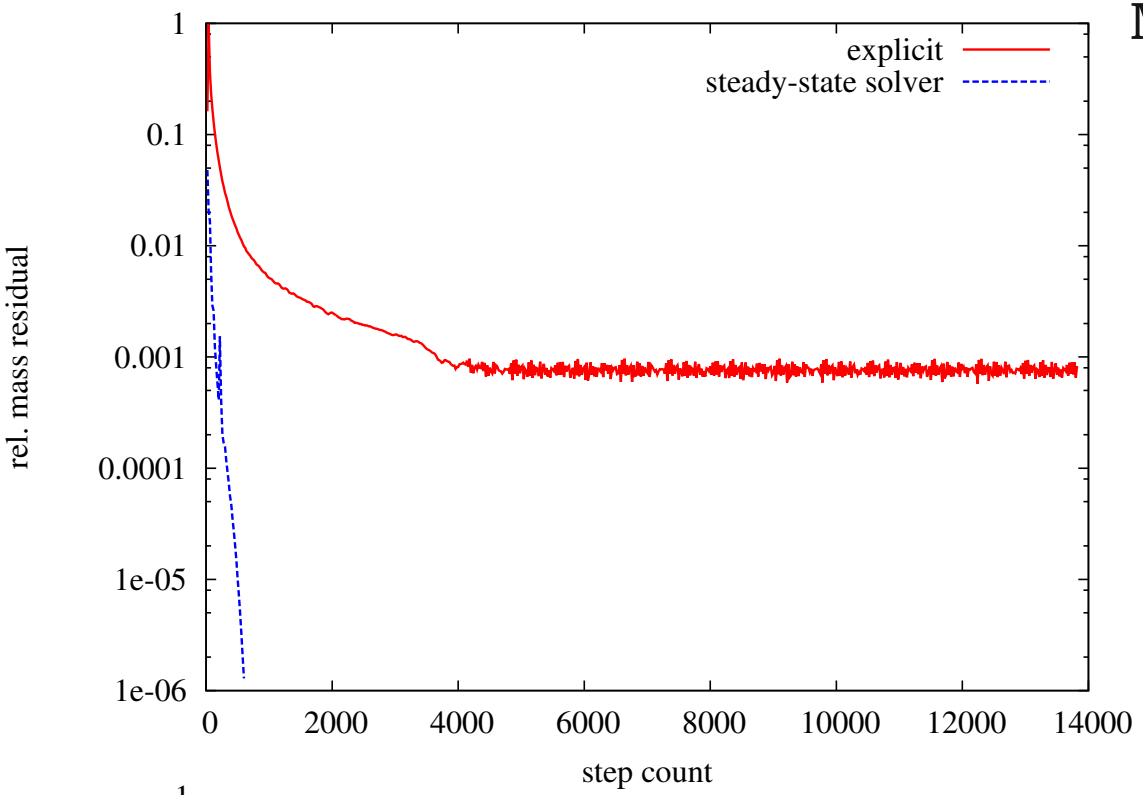


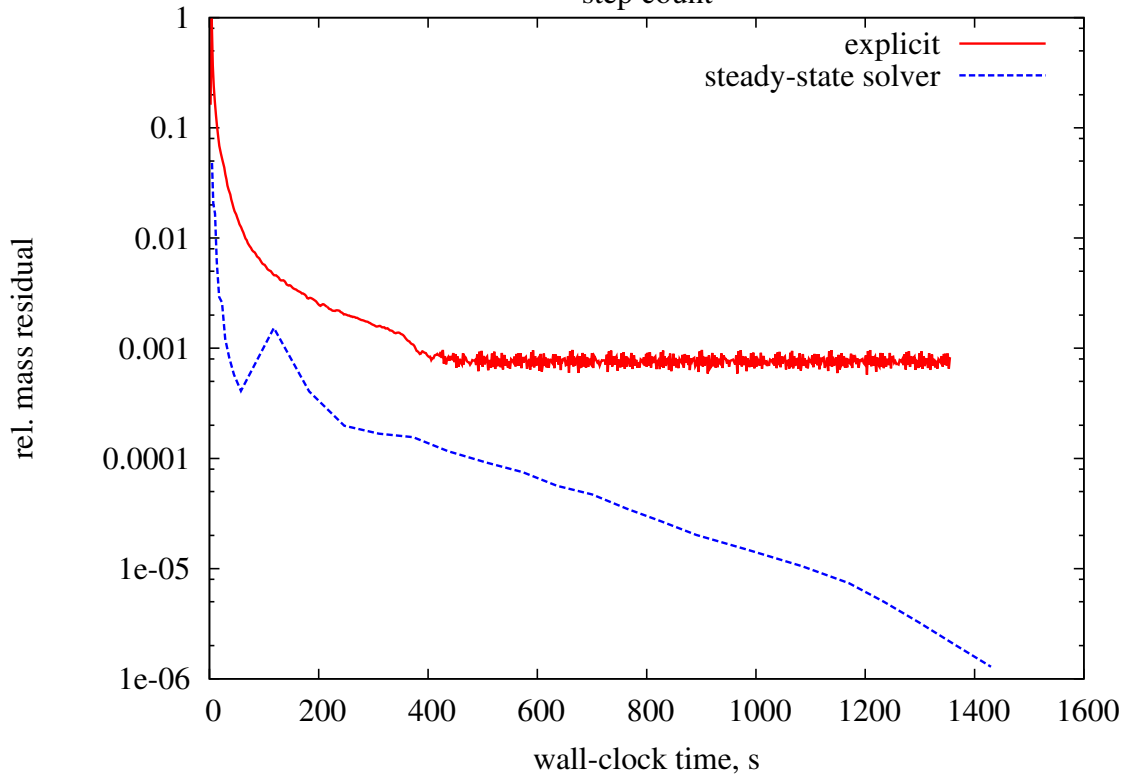
Mach 4 laminar flow over a flat plate





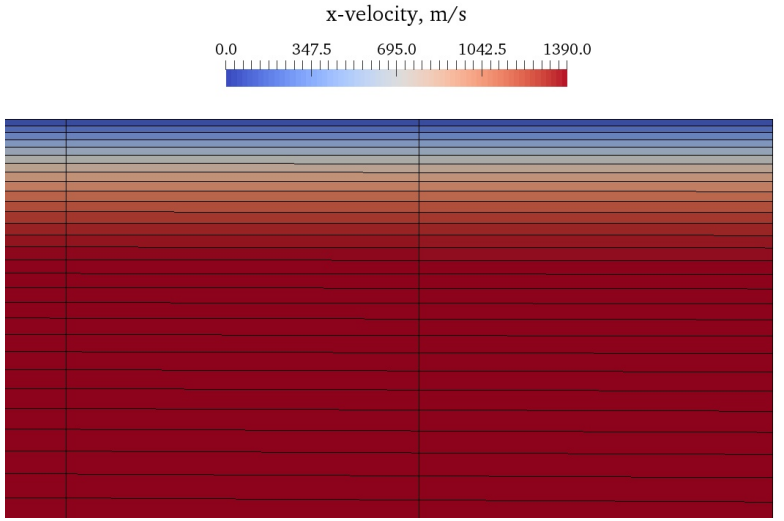
Mach 4 laminar flow over flat plate





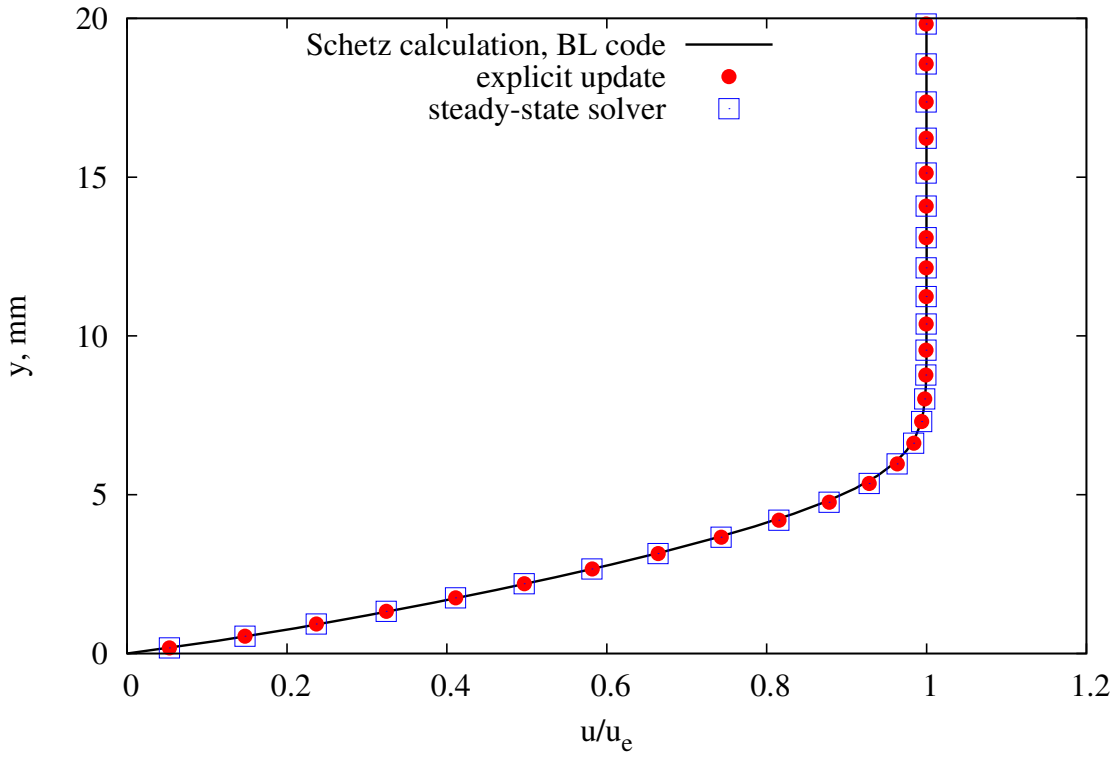
Notes:

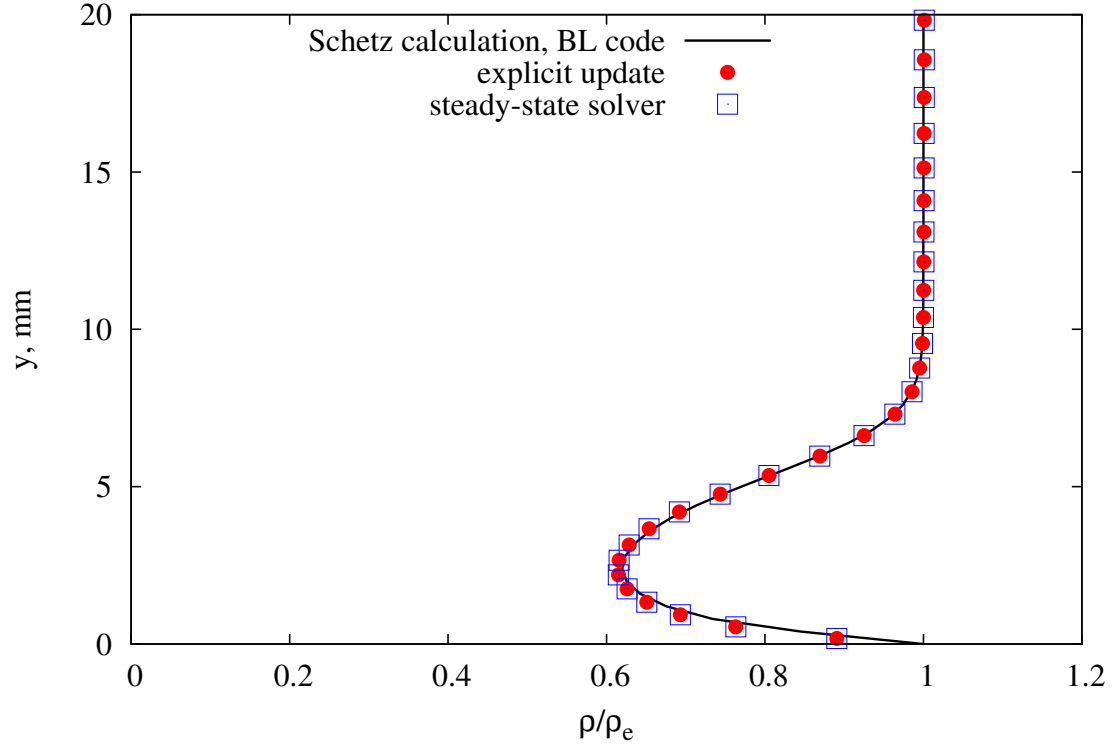
- Flux calculator: ausmdv
- Geometry: 2D planar
- Half-resolution of standard eilmer4 examples
- Explicit updates stalls at about 3 orders of residual drop; fine for engineering work
- Compute cores: 4 x Intel Core i3-4170



Mach 4 laminar flow over flat plate

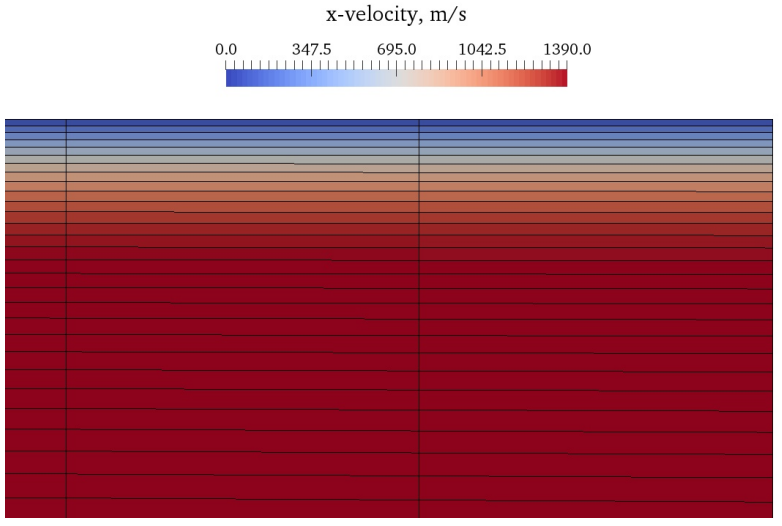
comparison to boundary layer code



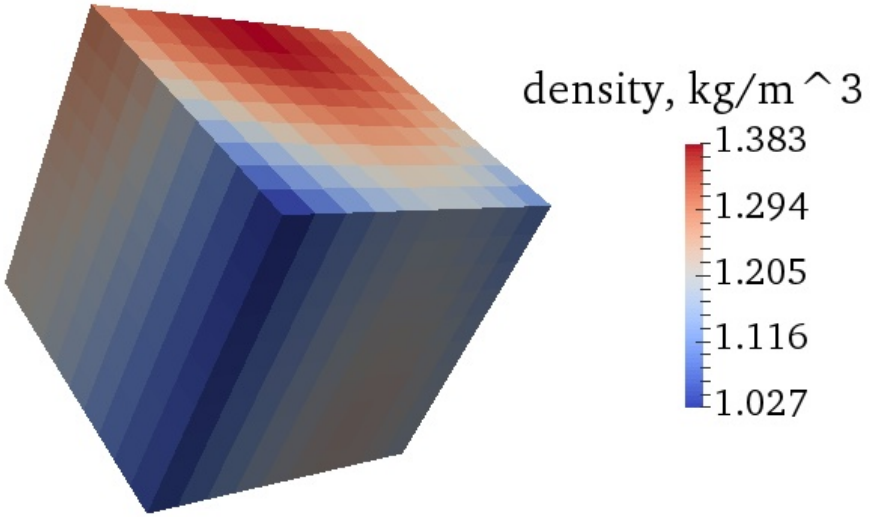


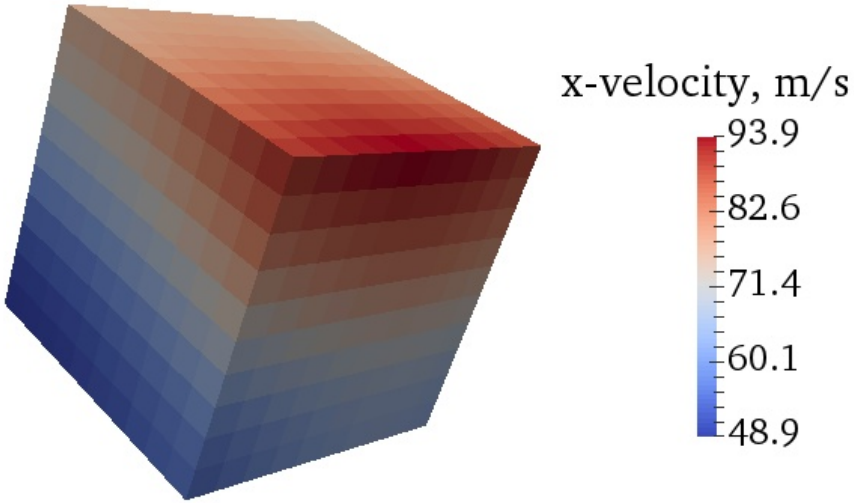
Notes:

- Explicit update matches with only 3 orders of magnitude
- In other words, could terminate steady-state solver early and achieve same match



3D Method of Manufactured Solutions

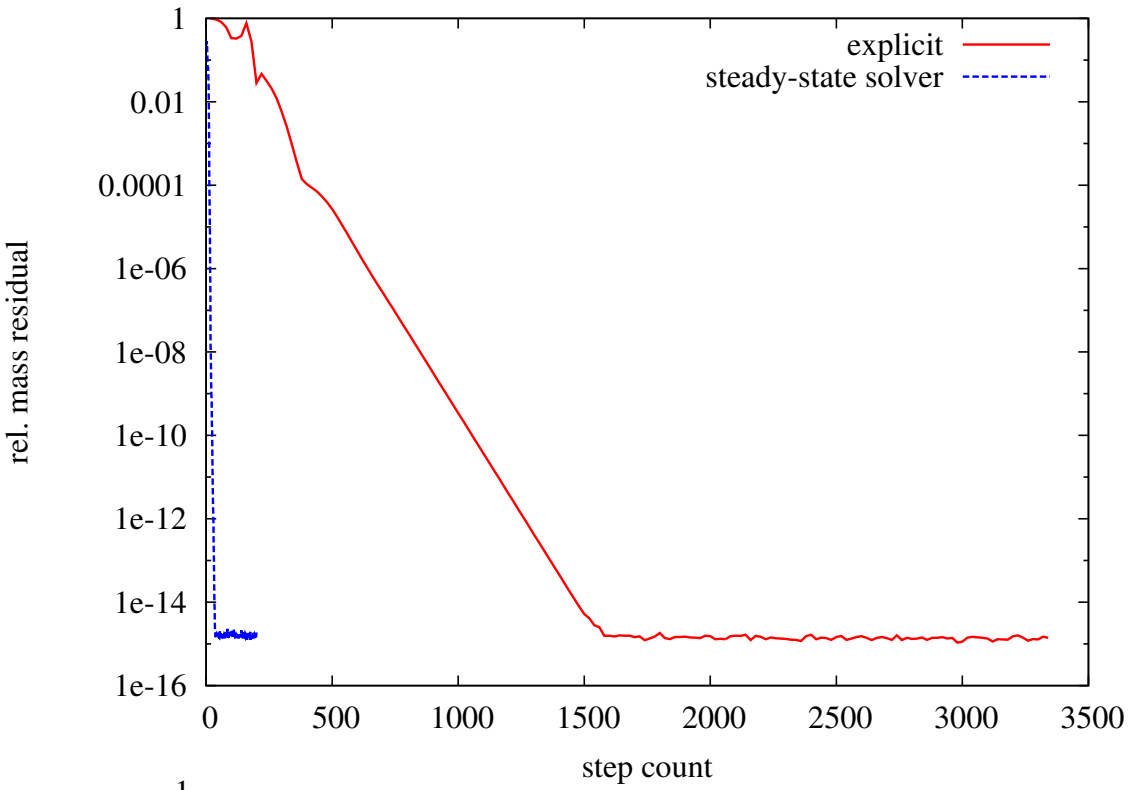


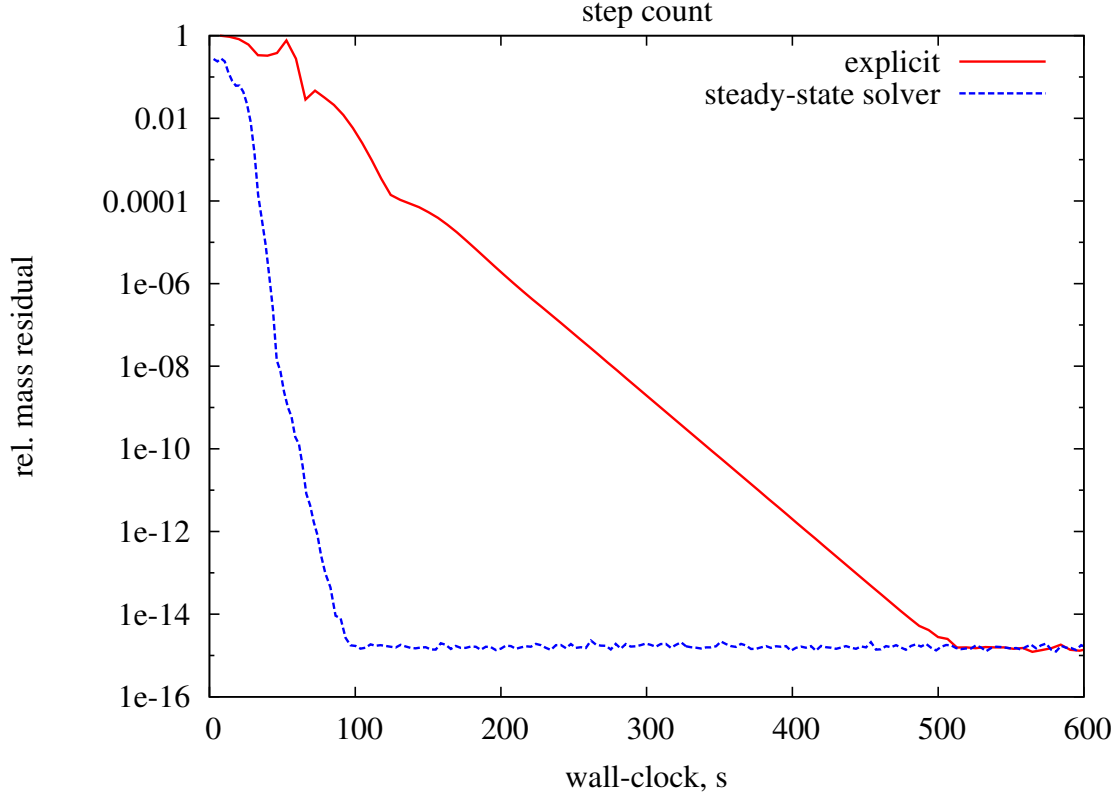


Throwing in the kitchen sink...

- unstructured grid in 3D
- multiple block
- parallel
- viscous terms
- user-defined B.C.s
- user-defined source terms

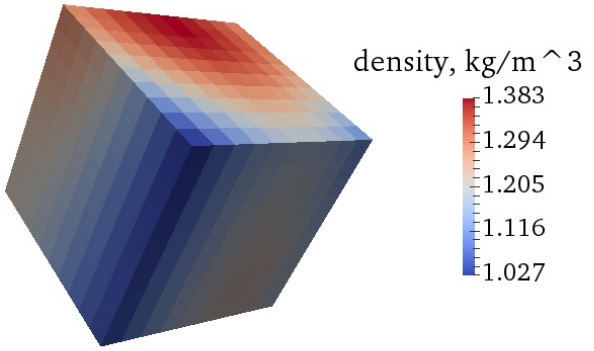
3D Method of Manufactured Solutions

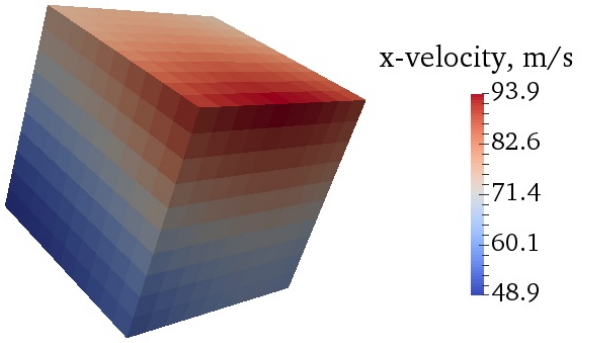




Notes:

- Compute cores: 8 x AMD Opteron 6378





Concluding Remarks

- A steady-solver mode is available and ready for business in Eilmer4
- The Newton-Krylov algorithm is an excellent choice when the code needs to handle several variations of physical modelling, because the details of Jacobian formation are abstracted away.
- In practical application, the Newton-Krylov algorithm has a few knobs to turn in order to get the best wall-clock performance. The settings on those knobs are problem dependent, but they seem to hold well across a family of like flow situations.
- Next steps:
  + A distributed-memory implementation so that we can solve problems across multiple nodes
  + Robustness and performance testing for 3D unstructured grids at large scale (millions of cells)