Progress of the Eilmer 4 transient flow solvers

Peter Jacobs, Rowan Gollan, Kyle Damm

The University of Queensland

29 Aug 2019

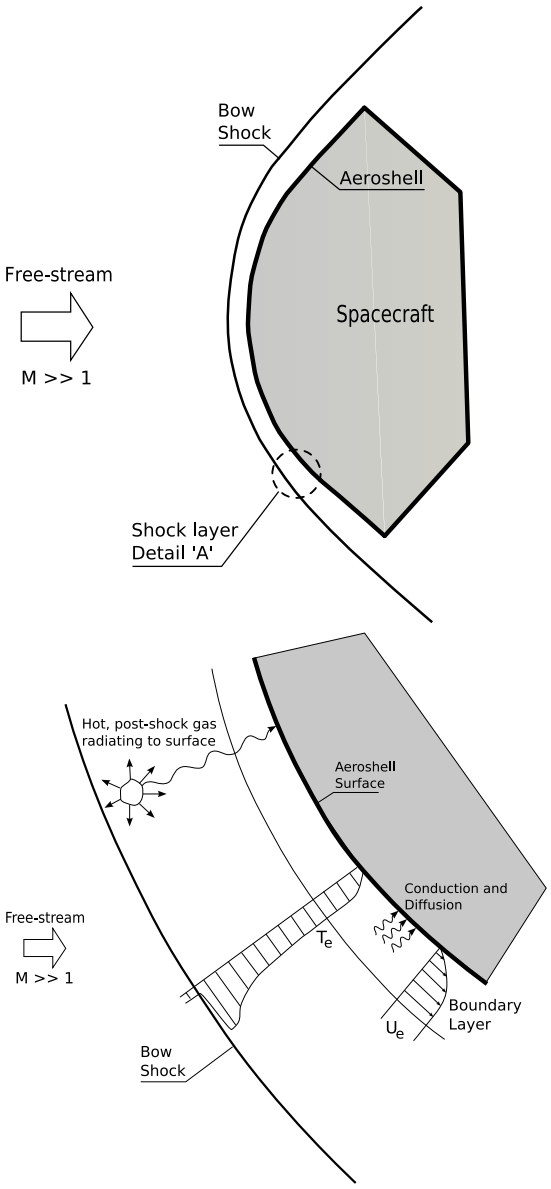
Background CFD ideas (for new members of CfH)

- When and why?
- How?

Progress 2018-2019

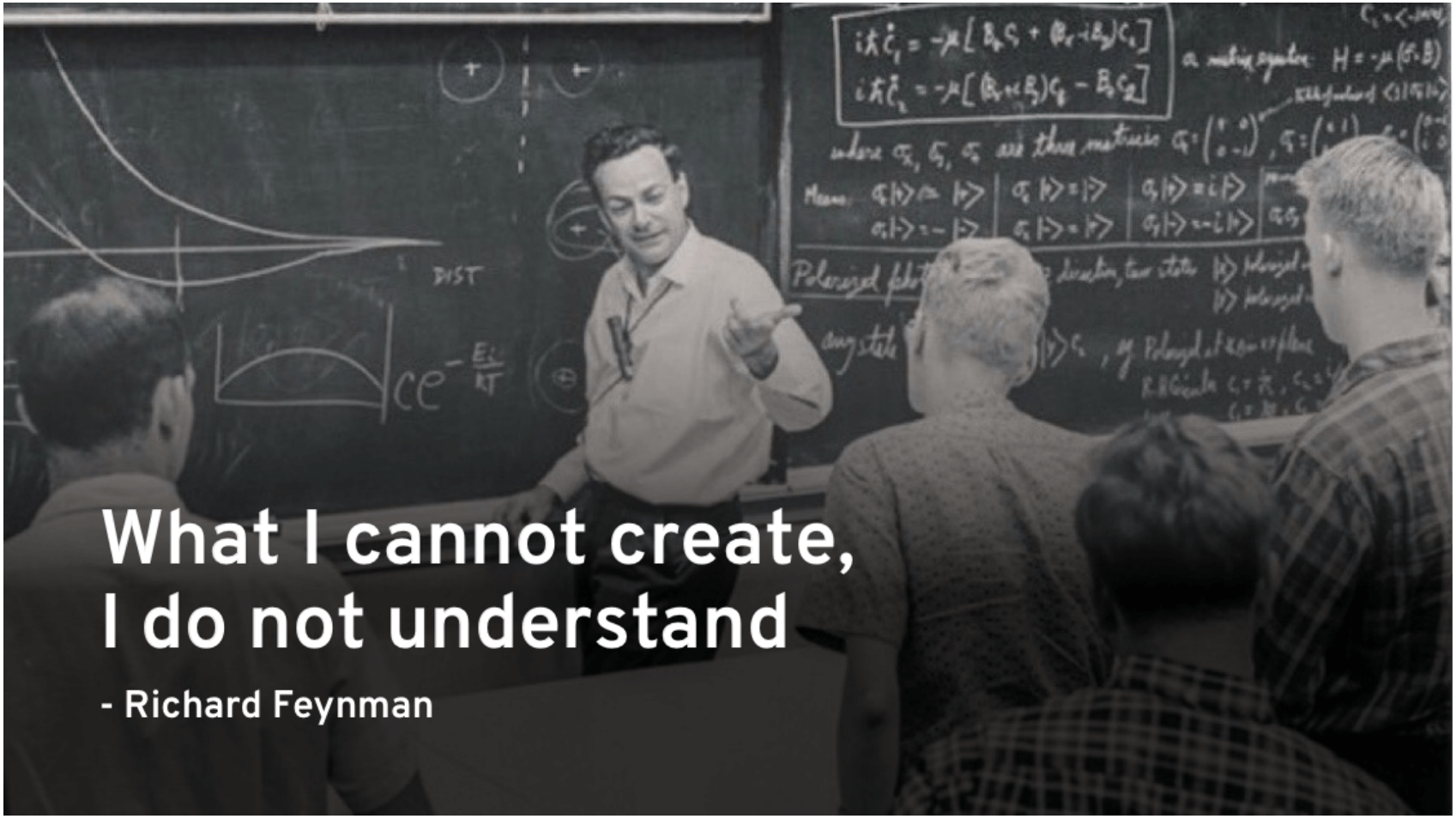
- MPI flavour of transient solver
- Complex numbers
- One-sided flux calculator and moving grids
- Complex 2D (unstructured) grids

When should we compute hypersonic flows?



- The flow physics are modelled well, but interactions are complex.
- Physical experimentation provides insights but has limitations, such as: scaling (time and length), boundary conditions, quantification of uncertainties, expense.
- Computer simulation complements physical experiments, and vice versa.
- Analysis via computer simulation (might) substitute when we don't have suitable experience.
- Computer analysis is good for 'what-if' studies, and design.

What I cannot simulate, I do not understand.



- We have to understand nature, just enough to deal with it. – Ray Stalker.

Compressible flow simulation via finite volumes

- What we compute: solution to the conservation equations for a viscous compressible flow
- How we compute: discretise in space and time
- Start with a known state and boundary conditions, then use an update algorithm
  - Convective fluxes: reconstruction-evolution approach, interpolation order, flux calculators, limiters
  - Viscous-transport fluxes: gradient estimation
  - Time integration

Form of the equations to solve

Integral form of a conservation law

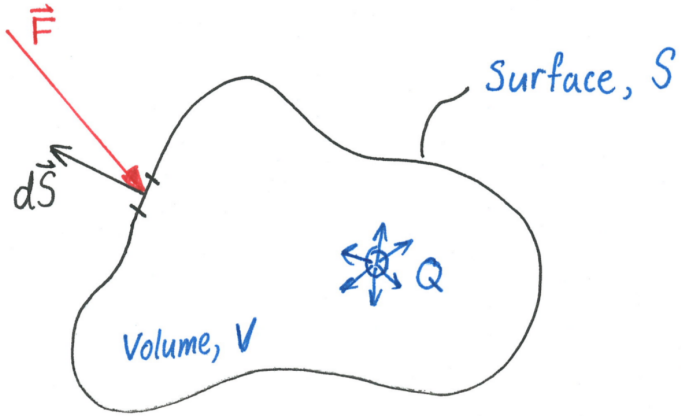
A general conservation law for quantity U is written in integral form as

$$\frac{\partial}{\partial t}\int_V U dV = -\oint_S \left(\vec{F}_c - \vec{F}_d\right)\cdot \hat{n}\, dA + \int_V Q dV\ , \qquad (1)$$

where S is the bounding surface and $\hat{n}$ is the outward-facing unit normal of the control surface.

What are the quantities U?

The conserved quantities in a compressible flow are **mass**, **momentum** and **energy**. In two dimensions, we can group these conservation equations with vector notation.



Integral form of conservation laws in vector form

For an ideal gas in two dimensions, the vector of conserved quantities is:

$$U = \begin{bmatrix} \rho \\ \rho u_x \\ \rho u_y \\ \rho E \end{bmatrix} , \tag{2}$$

with convective flux vector

$$\vec{F}_c = \begin{bmatrix} \rho u_x \\ \rho u_x^2 + p \\ \rho u_y u_x \\ \rho E u_x + p u_x \end{bmatrix} \hat{i} + \begin{bmatrix} \rho u_y \\ \rho u_x u_y \\ \rho u_y^2 + p \\ \rho E u_y + p u_y \end{bmatrix} \hat{j} , \tag{3}$$

and diffusive flux vector

$$\overline{F}_d = \begin{bmatrix} 0 \\ \tau_{xx} \\ \tau_{yx} \\ \tau_{xx} u_x + \tau_{yx} u_y + q_x \end{bmatrix} \hat{i} + \begin{bmatrix} 0 \\ \tau_{xy} \\ \tau_{yy} \\ \tau_{xy} u_x + \tau_{yy} u_y + q_y \end{bmatrix} \hat{j} . \tag{4}$$

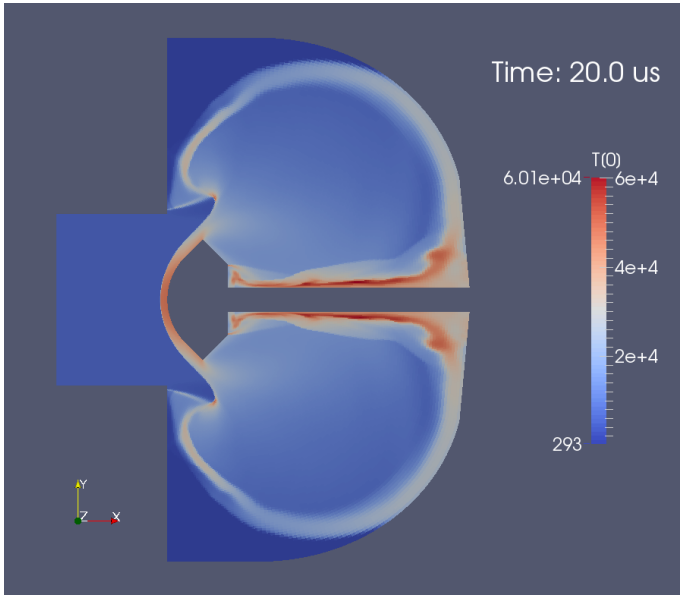
The vector of sources Q is typically zero.

Other necessary (and optional) pieces...

Thermodynamic model of the gas
Finite-rate chemical kinetics
Radiation energy exchange (yet to port)
Boundary conditions
Features:

- 3D from the beginning, 2D as a special case
- structured- and unstructured-meshes for complex geometries
- moving meshes
- coupled heat transfer
- shared-memory parallelism for multicore workstation use
- block-marching for speed (nenzfr and nozzle design)
- GPGPU processing for thermochemistry
- MPI distributed-memory parallelism

Features – 1/2



- 2D/3D compressible flow simulation.
- Gas models include ideal, thermally perfect, equilibrium (LUT).
- Finite-rate chemistry.
- Multi-temperature and state-specific thermochemistry.
- Inviscid, laminar, turbulent (k-ω) flow.
- Solid domains with conjugate heat transfer in 2D.
- User-controlled moving grid capability, with shock-fitting method for 2D geometries.
- Dense-gas thermodynamic models and rotating frames of reference for turbomachine modelling.

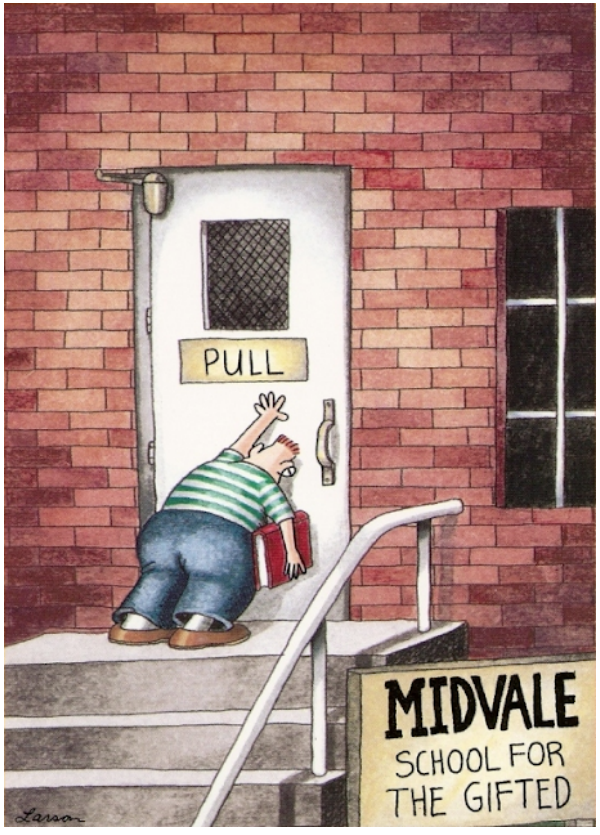
Features – 2/2

- Transient, time-accurate, using explicit Euler, PC, RK updates.
- Alternate steady-state solver with implicit updates using Newton-Krylov method.
- Parallel computation via shared-memory on workstations, and using MPI on a cluster computer.
- Multiple block, structured and unstructured grids.
- Native grid generation and import capability.
- Unstructured-mesh partitioning via Metis.

- `en.wikipedia.org/wiki/Eilmer_of_Malmesbury`
- Gas model calculator and compressible flow relations.

Documentation

- Web site: `cfcfd.mechmining.uq.edu.au/eilmer`
- Source code: `bitbucket.org/cfcfd/dgd`



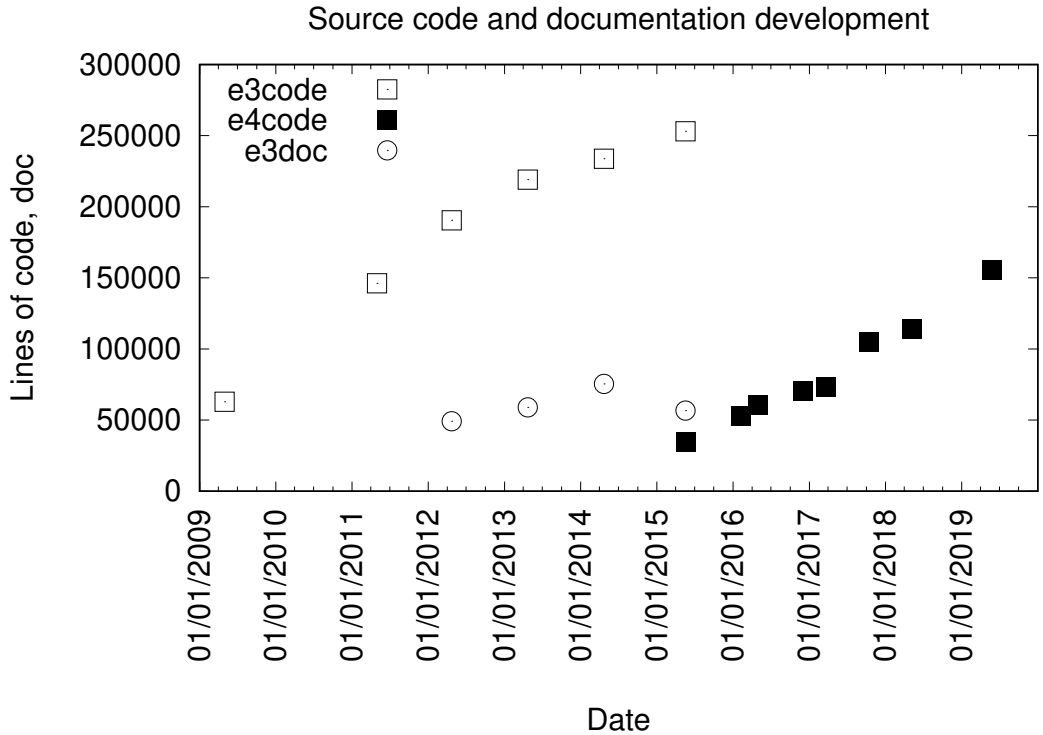
Documentation in the Eilmer 4.0 guides:

- Guide to the transient flow solver
- Guide to the basic gas models package
- Guide to the geometry package
- Formulation of the transient flow solver
- Reacting gas thermochemistry

Gary Larson, *The Far Side*

Development progress, in lines of source code.

Margaret Hamilton (1969) with Apollo Guidance Computer source code, assembly, ≈11,000 pages.



At 60 lines per page, the Eilmer4 code is equivalent to a 2500 page document.

Development progress, maturity of code.

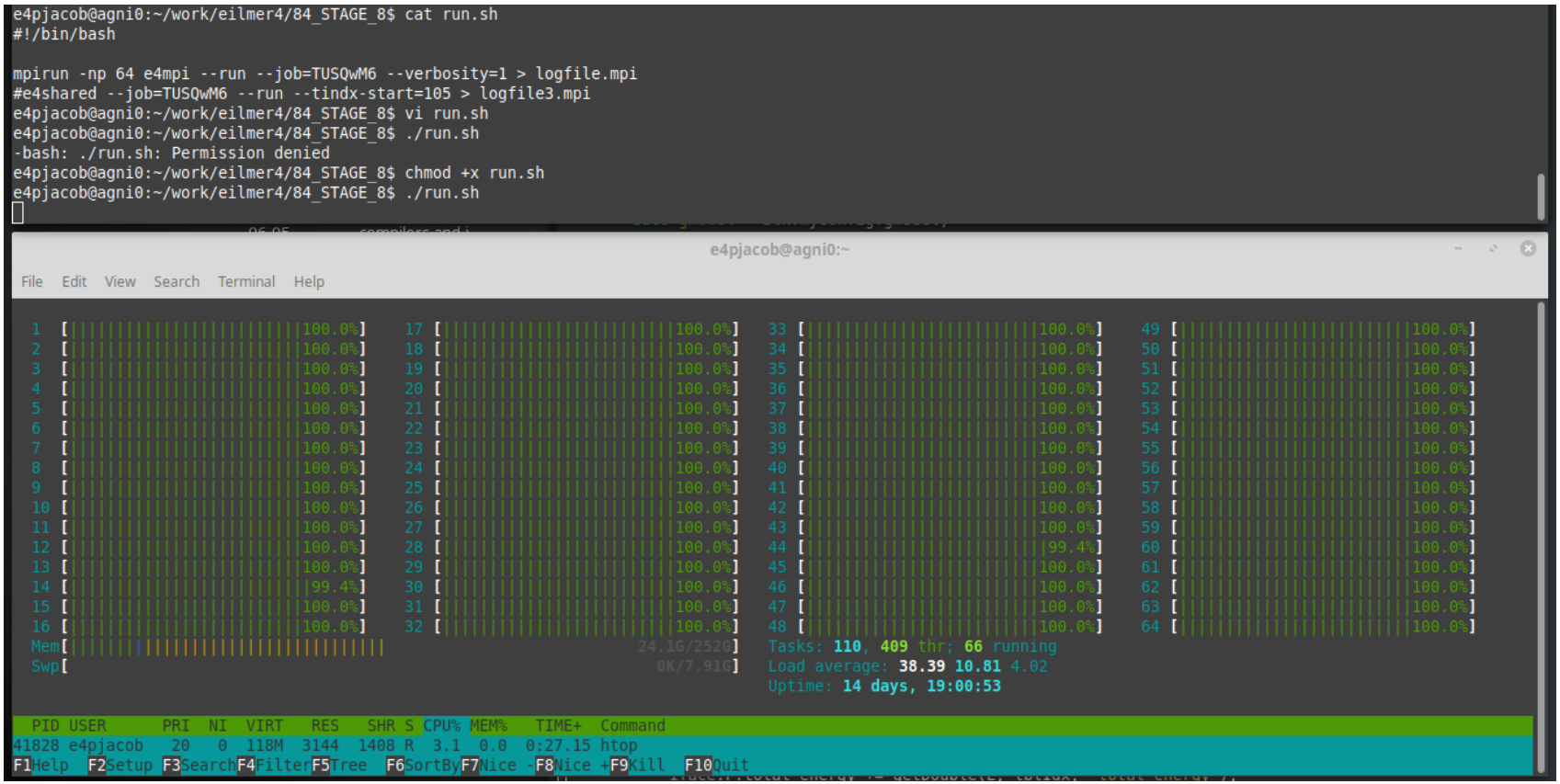
- Memory clean-up; but we still have some margin with dynamic arrays.
- Code can be built with reduced-capabilities.

```
make DMD=ldmd2 FLAVOUR=fast WITH_MPI=1 \
MULTI_SPECIES_GAS=0 MULTI_T_GAS=0 \
MHD=0 KOMEGA=0 install
```

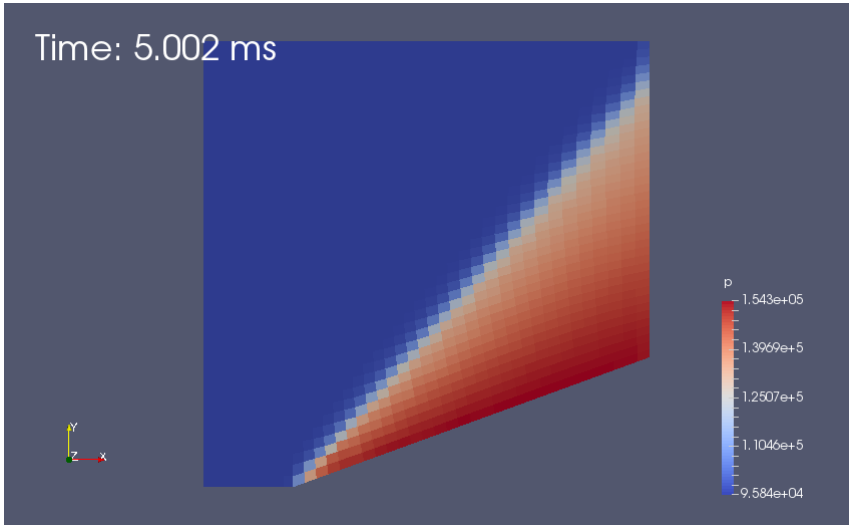
- Most core code is now @nogc. This reduces the number of temporary copies.
- Improved run time. Eilmer4 is fast as, or faster than the Eilmer3 code that was written in C++.
- We catch and handle (more carefully) a larger range of exceptional situations than we did a year ago.

Parallel calculation with MPI

- First of two big development items in the transient flow solver for 2018.
- Blocks partition the flow domain into connected pieces and compute the flow solution in the blocks in parallel.
- This is how we like to keep our workstations busy in 2019.



Simulation run times across the decades.



- 1991 SUN workstation with Sparc-2 processor
  - 5.36 hours for 2410 steps on a 100×100 mesh
  - 0.8 milliseconds/cell/predictor-corrector-update
- 2019 Dell Optiplex 990 with Intel i7 (4 cores, 8 threads)
  - 37.2 seconds for 2080 steps on a 100×100 mesh
  - 1.79 microseconds/cell/predictor-corrector-update
  - 0.7 seconds for 820 steps on a 40×40 mesh (usual test case)
  - 0.53 microseconds/cell/Euler-update (benefit of cache?)

Complex numbers are simple in D

- Second of the big development items for the transient flow solver in 2018.
- The malleability of D code allowed us to experiment with a redefinition of our numbers.

```
 6  module nm.number;
 7
 8  import std.math;
 9  import nm.complex;
10
11  version(complex_numbers) {
12      alias number = Complex!double;
13  } else {
14      alias number = double;
15  }
```

Using complex numbers

- Once numbers and operations/functions are defined, proceed as usual.
- Derivative can be evaluated a $\frac{\partial f}{\partial x} = \mathrm{Im}(f(x + \mathrm{i}h))/h$, without concern for round-off error as h becomes small (say, 10^{-20}).
- Sample code from fvcell.d below.
- Rowan and Kyle will tell you how to use these derivatives in the steady-state and adjoint solvers.

```
705         // Time-derivative for Mass/unit volume.
706         number integral = 0.0;
707         foreach(i; 0 .. nf) { integral -= myF[i].mass*area[i]; }
708         my_dUdt.mass = vol_inv*integral + Q.mass;
709
710         // Time-derivative for Momentum/unit volume.
711         number integralx = 0.0; number integraly = 0.0; number integralz = 0.0;
712         foreach(i; 0 .. nf) {
713             integralx -= myF[i].momentum.x*area[i];
714             integraly -= myF[i].momentum.y*area[i];
715             if ((myConfig.dimensions == 3) || ( myConfig.MHD )) {
716                 // require z-momentum for MHD even in 2D
717                 integralz -= myF[i].momentum.z*area[i];
718             }
719         }
```

One-sided flux calculator

- Main development item (for PJ) in 2019, prompted by moving-grid simulations.
- Historically, code had used ghost-cells to implement all boundary conditions because it simplified the interpolation and flux calculation code.
- With the natural law of *conservation of evil*, this caused the boundary conditions to become more complex.
- So, let's not pretend that there is gas where there is none.
- This makes the code for interpolation a bit more complicated because it needs to adjust to lower order (with fewer data points) as an external boundary is approached.

One-sided flux calculator – theory 1 of 2

- Consider local flow region (R), close to a solid boundary on the left.
- One-dimensional flow, normal to boundary, is determined by a simple wave process into the (ideal) gas on the right.
- Given state of near-wall gas and wall velocity, v^*, assume an isentropic wave to compute pressure at the wall.

$$p^* = \left[\left(v^* - \bar{U}_R \right) \frac{\gamma - 1}{2\sqrt{\gamma}} \left(\frac{\rho_R}{p_R^{1/\gamma}} \right)^{\frac{1}{2}} \right]^{2\gamma/(\gamma-1)}$$

where $\bar{U}_R$ is the Riemann invariant $\bar{U}_R = v_R - \frac{2a_R}{\gamma-1}$.

- This happens to be the core of gas-dynamic calculation within the L1d flow solver.

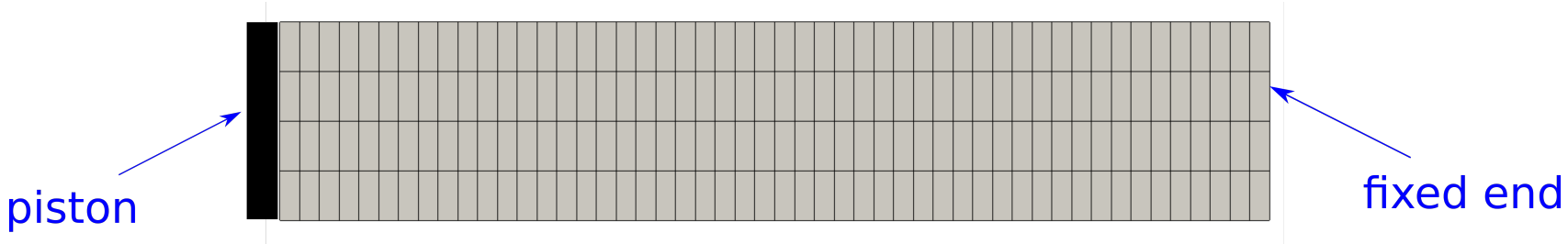
One-sided flux calculator – theory 2 of 2

- If there is a pressure rise, assume that a shock exists and solve the shock relations iteratively to get p^*.
- Given conditions at the boundary, p^*, v^*, write the mass, momentum and energy fluxes as

$$0, \quad p^*, \quad p^*.v^*$$

simples.

Example 1 – constant-velocity piston pushing gas



- This is the simplest moving-grid simulation; a simple grid motion is imposed and there is no feedback from the gas flow.
- With the use of recently-code features, we can simplify Rowan's piston example from 2018 seminar.
  - The one-sided flux calculators have direct application at all walls, moving or stationary, so that we no longer need to define our own flux calculation at the piston face.
  - We have more convenient functions for interpolating grid velocities.

Constant-velocity piston pushing gas – input script

```
19  -- Geometry, grid and block setup.
20  L = 0.5; H = 0.1
21  -- Gas region that drives piston.
22  patch0 = CoonsPatch:new{p00=Vector3:new{x=0, y=0},
23                          p10=Vector3:new{x=L, y=0},
24                          p11=Vector3:new{x=L, y=H},
25                          p01=Vector3:new{x=0, y=H}}
26  grid0 = StructuredGrid:new{psurface=patch0, niv=51, njv=3}
27  blk0 = FluidBlock:new{grid=grid0, initialState=initial,
28                        bcList={west=WallBC_WithSlip1:new{}}
29  }
30
31  -- Simulation control parameters
32  config.gasdynamic_update_scheme = "moving_grid_1_stage"
33  config.grid_motion = "user_defined"
34  config.udf_grid_motion_file = "grid-motion.lua"
```

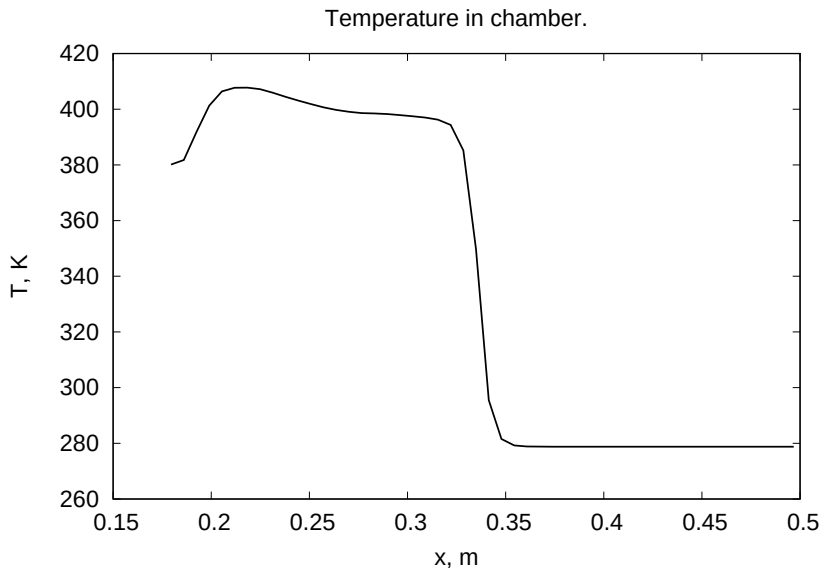
- WallBC_WithSlip1 is a boundary condition that does not have ghost cells. WallBC_WithSlip is an alias.
- WallBC_WithSlip0 uses ghost cells (assuming zero grid speed).

Constant-velocity piston pushing gas – grid motion

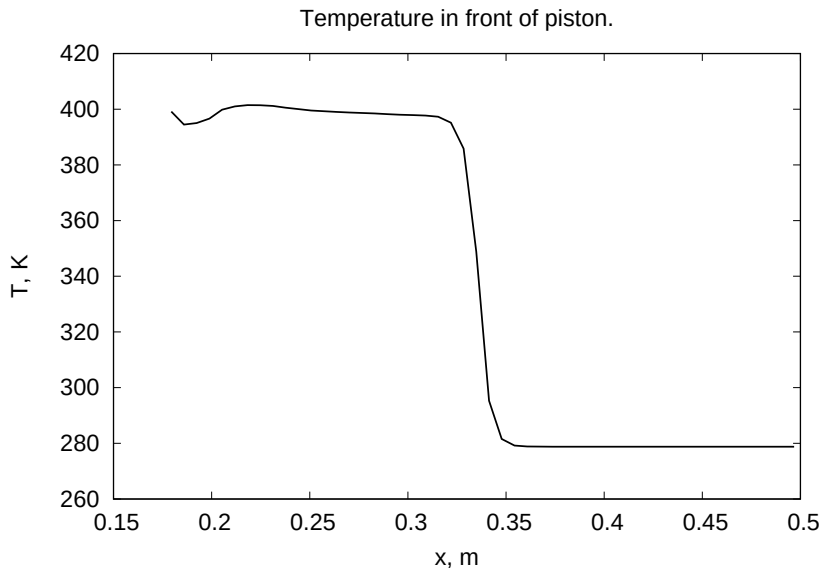
```
1  -- Authors: Rowan J. Gollan & Peter J.
2  -- Dates: 2017-01-04 -- 2019-05-21
3  --
4  zero = Vector3:new{x=0, y=0}
5  pSpeed = Vector3:new{x=293.5, y=0}
6
7  function assignVtxVelocities(t, dt)
8     local blkId = 0
9     -- Corner labels -                p00     p10   p11   p01
10    setVtxVelocitiesByCorners(blkId, pSpeed, zero, zero, pSpeed)
11 end
```

- Several functions available to interpolate grid velocities to all vertices.
- If none of those are suitable, you may set the velocity of individual vertices as Rowan had done last year.

Constant-velocity piston pushing gas – result at t=0.6ms



Temperature profile for user-defined flux calculator that uses cell pressure.



Temperature profile for one-sided flux calculator. Smaller wobbles are better!

Example 2 – piston in tube, pushed by gas

- Put piston on right end of gas slug and let the gas expand to the right.
- Add piston dynamics so that the simple grid motion responds to the gas flow.
- Need the gas force on the piston face to compute piston acceleration.
- Retain the dynamic state (x, v) of the piston which is updated by the user-defined supervisory function at the start of each time step.
- Communicate the piston state to the user-defined function that sets the grid motion.

Piston in tube – input script

```
22  -- Geometry, grid and block setup.
23  -- Gas region that drives piston.
24  driver_patch = CoonsPatch:new{p00=Vector3:new{x=0,  y=0},
25                                p10=Vector3:new{x=L1, y=0},
26                                p11=Vector3:new{x=L1, y=H},
27                                p01=Vector3:new{x=0,  y=H}}
28  grid0 = StructuredGrid:new{psurface=driver_patch, niv=101, njv=3}
29
30  blk0 = FluidBlock:new{
31     grid=grid0, initialState=initial,
32     bcList={east=WallBC_WithSlip1:new{group='pistonUpstream'}}
33  }
43  -- Calculate the projectile dynamics in user-defined functions
44  -- but using loads computed by the flow solver.
45  config.udf_supervisor_file='udf-supervisor.lua'
46  config.compute_run_time_loads = true
47  config.run_time_loads_count = 1
48  run_time_loads={
49     {group="pistonUpstream", moment_centre=Vector3:new{x=0, y=0}},
50  }
51  -- Dimension userPad for storing piston position and velocity.
52  config.user_pad_length = 2
53  user_pad_data = {0, 0}
```

Piston in tube – run-time-supervisor script

```
 5  pMass = 1.0 -- kg
 6
 7  function atTimestepStart(sim_time, steps, dt)
 8     -- Unpack current piston state.
 9     local x = userPad[1]
10     local xdot = userPad[2]
11     -- Get the surface loads on the piston.
12     local upstreamForce, upstreamMoment = getRunTimeLoads("pistonUpstream")
13     -- Acceleration of the piston.
14     local xdotdot = upstreamForce.x*2*math.pi / pMass
15     -- Update piston state using simple Euler update.
16     x    = x + xdot * dt
17     xdot = xdot + xdotdot * dt
18     -- Save data to userPad for vtxSpeed Assignment in grid-motion.
19     userPad[1] = x
20     userPad[2] = xdot
21     return
22  end
```

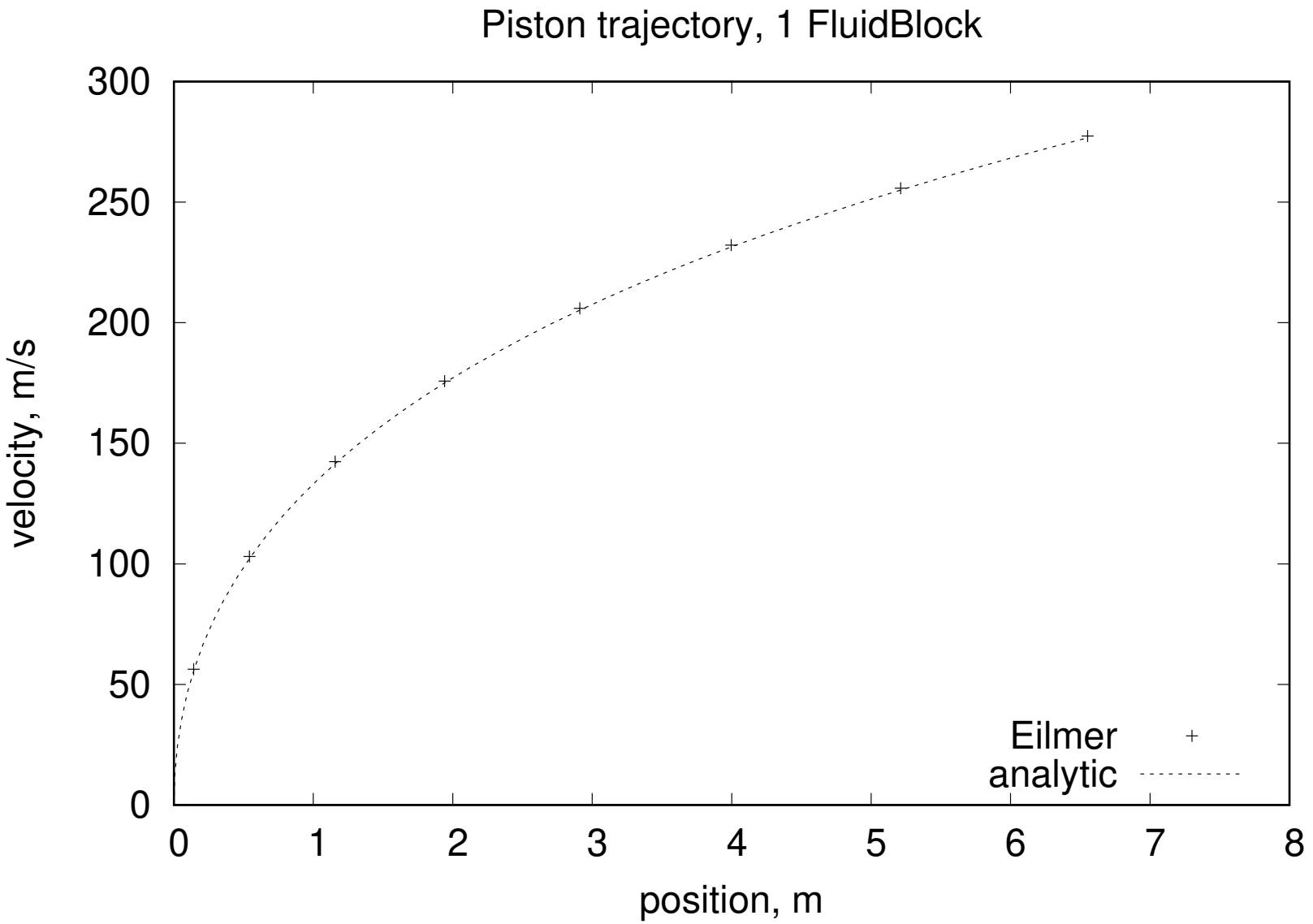
- The loads on the piston face are computed internal to the gas-dynamics code and can be called up by group name.
- Use $a = F/m$ and an Euler update for piston state.
- Put the new piston state into the userPad array.

Piston in tube – grid motion

```
1  -- Authors: Rowan J. Gollan, Fabian Zander, Peter J. Ingo J.
2  -- Date: 2017-01-04 -- 2019-05-21
3  --
4  function assignVtxVelocities(t, dt)
5     local xdot = userPad[2]
6     local zeroVel = Vector3:new{x=0, y=0}
7     local pSpeedVec = Vector3:new{x=xdot, y=0}
8     local blkId = 0
9     setVtxVelocitiesByCorners(blkId, zeroVel, pSpeedVec, pSpeedVec, zeroVel)
10 end
```

- Recover the current value of piston velocity from the userPad array.
- As noted in the comments, there were a lot of cooks in the kitchen.

Piston in tube – result over 40ms

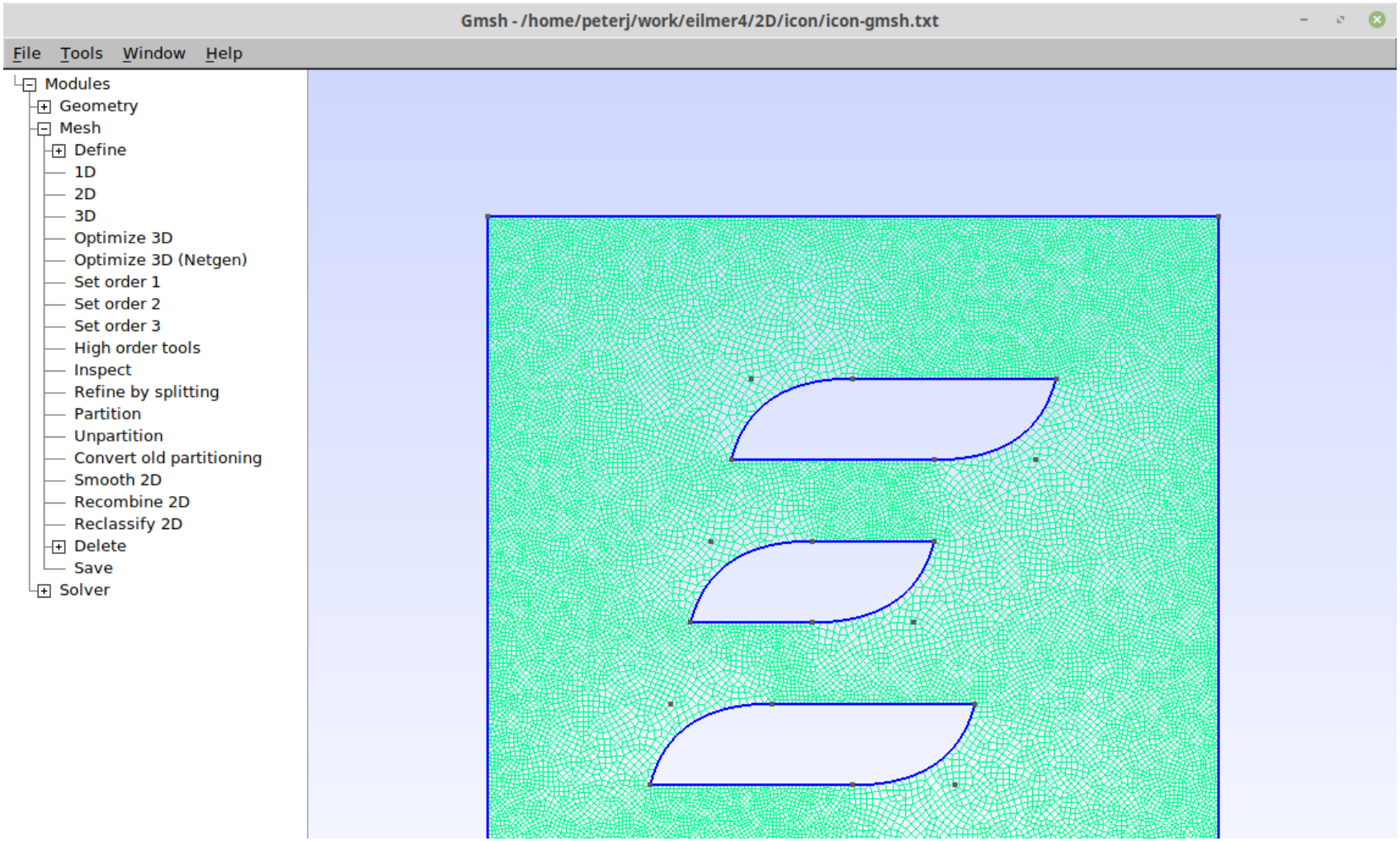


Complex 2D grid – Defining the flow region with SVGPath

```
23  labels = {-- "bounding box",
24            "TOP_BAR", "MIDDLE_BAR", "LOWER_BAR"}
25  svgStrings = {
26     -- "M-2.0,-2.0;h9.0;v9.0;h-9.0;Z",
27     "M1.0,4.0;h2.5;q1.25,0.0 1.5,1.0;h-2.5;q-1.25,0.0 -1.5,-1.0;Z",
28     "M0.5,2.0;h1.5;q1.25,0.0 1.5,1.0;h-1.5;q-1.25,0.0 -1.5,-1.0;Z",
29     "M0.0,0.0;h2.5;q1.25,0.0 1.5,1.0;h-2.5;q-1.25,0.0 -1.5,-1.0;Z"
30  }
31  pTag, cTag, lTag = 4, 4, 1
32  for i,svgStr in ipairs(svgStrings) do
33     svgPth = SVGPath:new{path=svgStr}
34     pTag, cTag, lTag, gmshStr = svgPth:toGmshString(pTag, cTag, lTag
35     f:write(gmshStr)
36  end
```

- Little language to define paths in "Scalable Vector Graphics"
- M x,y move absolute; Z close path
- h x horizontal line relative to start point
- q x1,y1 x2,y2 quadratic curve relative to start point
- So far, have implemented MmLlHhVvQqCcZ

Complex 2D grid – delegate grid generation to Gmsh



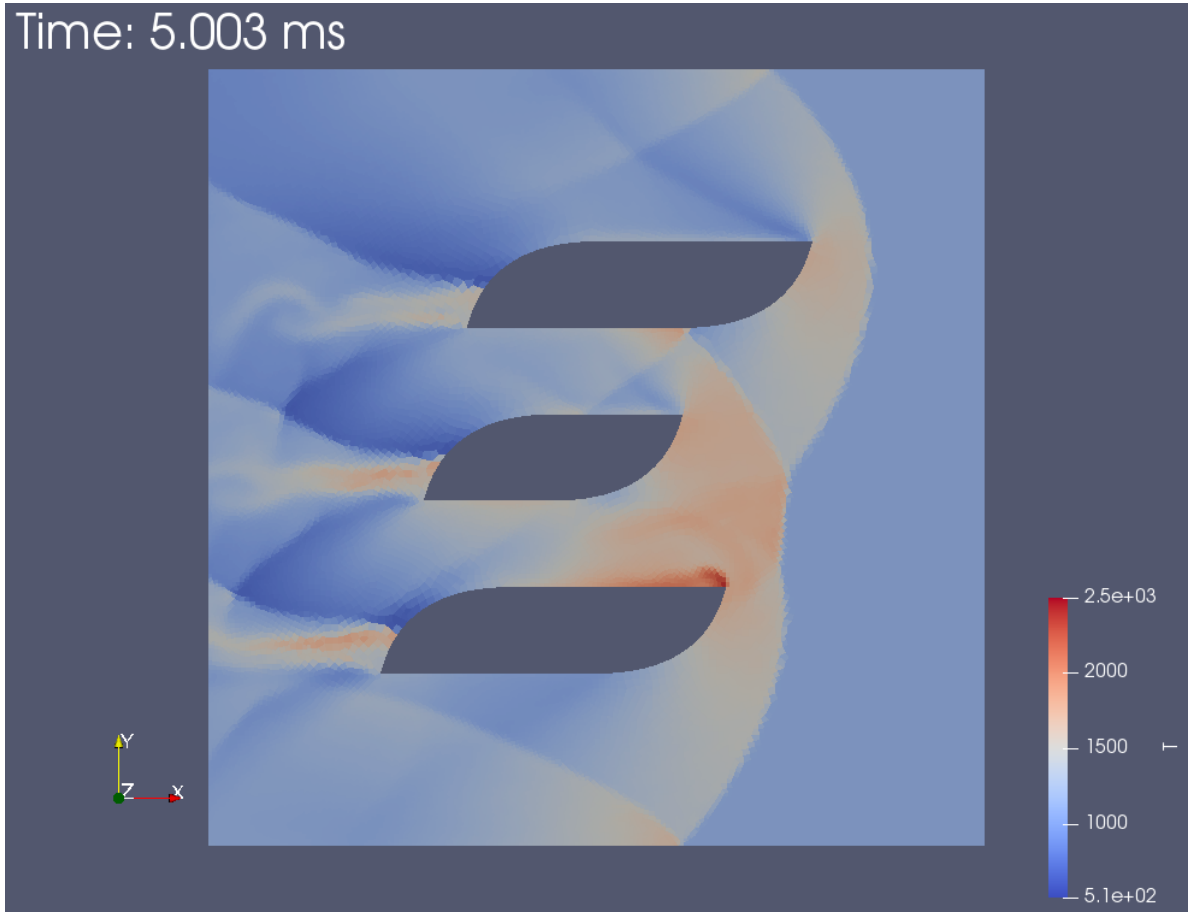
- Mesh ⇒ Define ⇒ 2D to get mesh of triangle elements
- Mesh ⇒ Define ⇒ Recombine 2D to get mesh of quads

Complex 2D grid – simulation script

```
19  -- Define the flow domain using an imported grid
20  nblocks = 4
21  grids = {}
22  for i=0,nblocks-1 do
23     fileName = string.format("block_%d_icon-gmsh.su2", i)
24     grids[i] = UnstructuredGrid:new{filename=fileName, fmt="su2text", scale=1.0}
25  end
26
27  my_bcDict = {INFLOW=InFlowBC_Supersonic:new{flowState=inflow,
    label="inflow-boundary"},
28            OUTFLOW=OutFlowBC_Simple:new{label="outflow-boundary"},
29            SLIP_WALL=WallBC_WithSlip:new{},
30            METIS_INTERIOR=ExchangeBC_MappedCell:new{cell_mapping_from_file=true}}
31
32  blks = {}
33  for i=0,nblocks-1 do
34     blks[i] = FluidBlock:new{grid=grids[i], initialState=inflow, bcDict=my_bcDict}
35  end
```

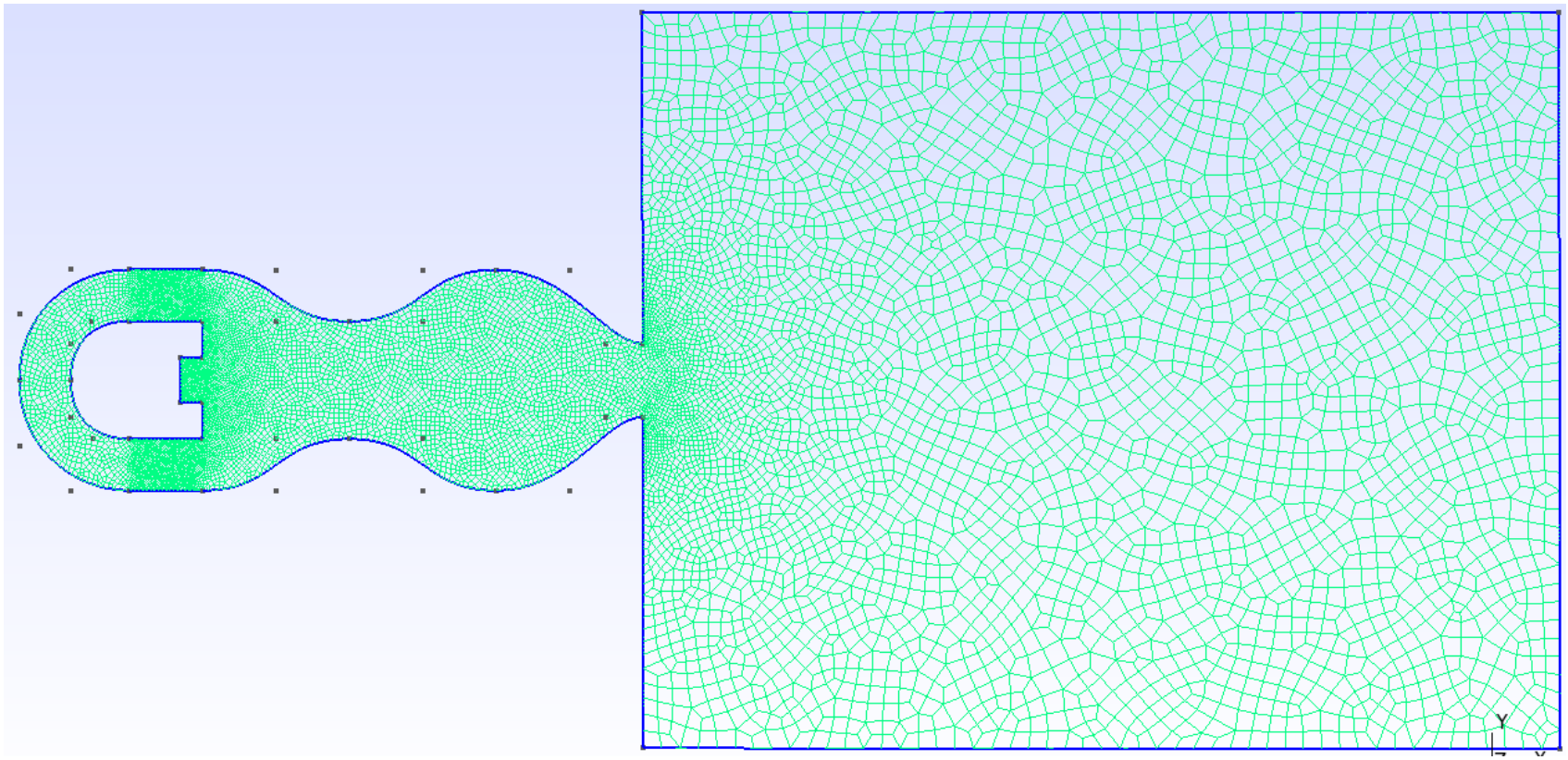
- Have divided the Gmsh grid into 4 blocks using Kyle's Metis partitioner.
- Unstructured grids are imported as SU2 files.

Complex 2D grid – simulation result after 40ms



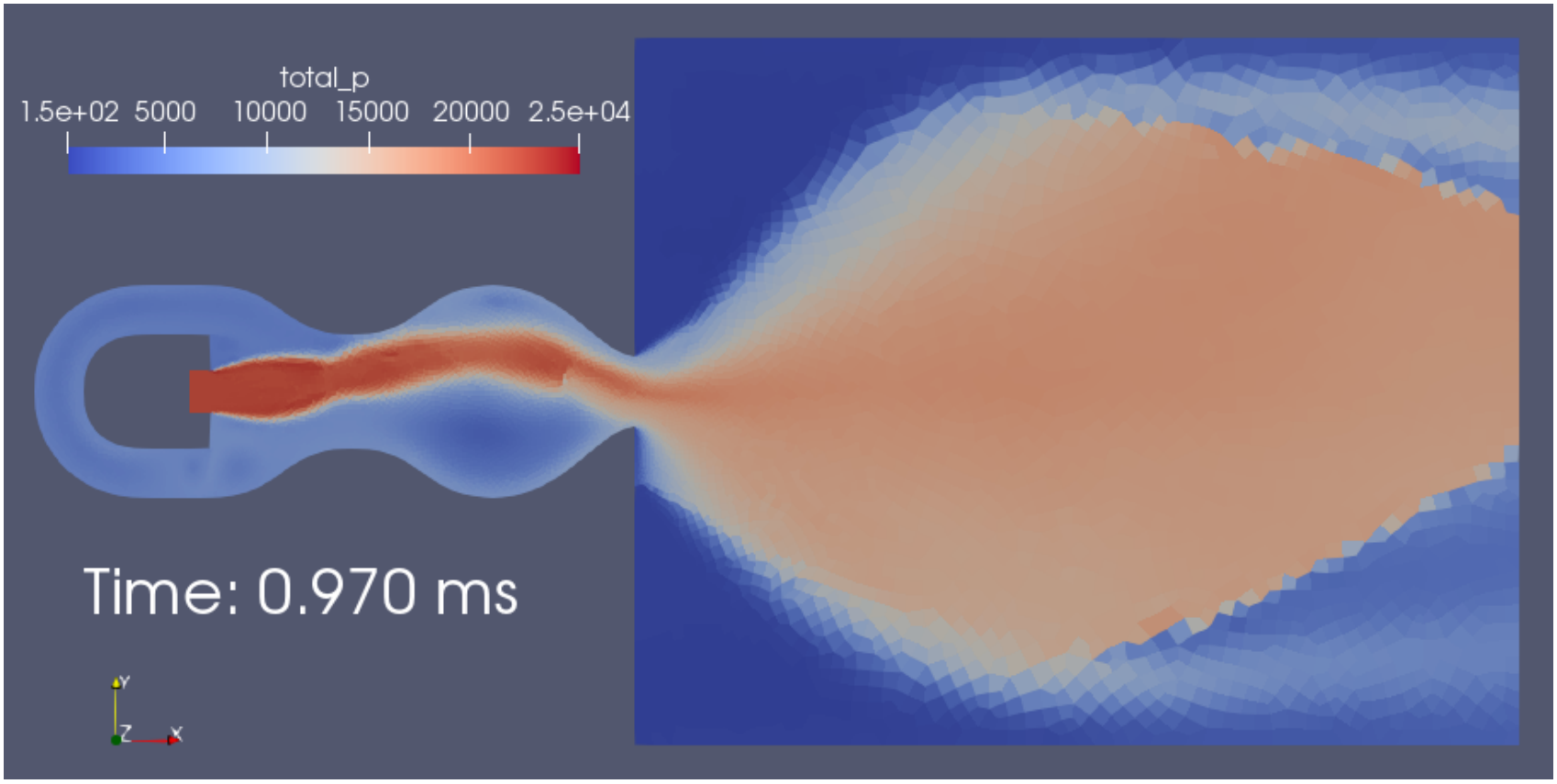
- Run time 94.3 seconds, 660 steps, on Dell Optiplex 990, 4 cores.

Complex 2D grid – Fluidic oscillator, fuel injector



- US Patent 4231519A Fluidic oscillator with resonant inertance and dynamic compliance circuit. Inventor: Peter Bauer
- Original application to the production of liquid sprays.
- Might allow a fuel jet to penetrate more widely.

Complex 2D grid – Fluidic oscillator, fuel injector



- This is a 2D simulation that might be relevant to a 3D slot injector where the slot is aligned with the main flow path.
- Animation is more informative for this transient flow.

Conclusion: Keep calm and putt.