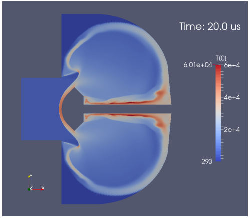
Implementation of a compressible-flow simulation code in the D programming language

Peter Jacobs, Rowan Gollan and Anand V.
Co-chief Gardeners of the CFCFD Code Collection
School of Mechanical Engineering, UQ

26 May 2016

Eilmer in a nutshell



- Eulerian/Lagrangian description of the flow (finite-volume, 2D axisymmetric or 3D).
- Transient, time-accurate, optionally implicit updates for steady flow.
- Shock capturing plus shock fitting boundary.
- Multiple block, structured and unstructured grids.
- Parallel computation on a cluster computer, using MPI in Eilmer2,3 and shared memory in dgd/Eilmer.
- High-temperature nonequilibrium thermochemistry (GPU).
- Dense-gas thermodynamic models and rotating frames of reference for turbomachine modelling.
- Turbulence models: Baldwin-Lomax and k-ω.
- Coupling to radiation and ablation codes for aeroshell flows.
- ...plus conjugate heat transfer and MHD

Origins

- in the late 1980s, the state of the art for scramjet simulations involving reactive flow was JP Drummond *SPARK* code
- Flow solver component based on Bob McCormack's (1969) finite-difference shock-capturing technique.
- All configuration hard-coded into the Fortran source code and compiled to run on a Cray supercomputer.
- In the 1980s, a new CFD technology (upwind flux) was being developed by the applied mathematics people and parallel computing environments were being developed by the computer science people (cluster computers).
- Dec 1990: following a CFD lesson on the chalk-board from Bob Walters and Bernard Grossman, *cns4u* was started with the intention to be like SPARK but with new technology

Development of Eilmer

- 1993 built *sm3d*, a space-marching code for 3D scramjet flows
- 1995 through 1999: the postgrad years expanded scope of experimentation and application
- 1996: code reformulation around fluxes (frequent discussions with Mike Macrossan); all code still in C with a preprocessor having a little command interpreter built in.
- 1997: discovered scripting languages Tcl and Python
- May 2003: *scriptit.tcl* provided fully programmable environment for simulation-preparation.
- Aug 2004: *Elmer* began as a hybrid code using Python and C.
- Jun 2005: rewrite of *Elmer(2)* in C alone so that Andrew Denman could get on with his thesis
- Jul 2006: rewrite *Elmer2* in C++ and, in 2008, call it *Eilmer3*. The class-based implementation was easier to extend and maintain.

Eilmer – Let's do it right, again.

Fred Brooks, in the "Mythical Man-Month: Essays on software engineering"

> *Sooner or later the first system is finished, and the architect, with firm confidence and a demonstrated mastery of the class of systems, is ready to build a second system. ...*
> *This second is the most dangerous system a man ever designs. ...*
> *The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one.*

We're OK, this is not our second system.
cns4u, mbcns, mbcns2, Elmer, Elmer2, Eilmer3 ... Eilmer4.

Eilmer4 – think big!

- Heather Muir has been working on the unstructured-grid generator. based on the paving algorithm.

Mathematical gas dynamics (in differential form)

Conservation of mass:

$$\frac{\partial}{\partial t}\rho + \nabla \cdot \rho \mathbf{u} = 0 \qquad (1)$$

Conservation of species mass:

$$\frac{\partial}{\partial t}\rho_i + \nabla \cdot \rho_i \mathbf{u} = -\left(\nabla \cdot \mathbf{J}_i\right) + \dot{\omega}_i \qquad (2)$$

Conservation of momentum:

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

Conservation of total energy:

$$\frac{\partial}{\partial t}\rho E + \nabla \cdot (e + \frac{p}{\rho})\mathbf{u} = \nabla \cdot [k\nabla T + \sum_{s=1}^{N_v} k_{v,s}\nabla T_{v,s}] + \nabla \cdot \left[\sum_{i=1}^{N_s} h_i \mathbf{J}_i\right]$$
$$- \left(\nabla \cdot \left[\left\{-\mu(\nabla \mathbf{u} + (\nabla \mathbf{u})^{\dagger}) + \tfrac{2}{3}\mu(\nabla \cdot \mathbf{u})\boldsymbol{\delta}\right\} \cdot \mathbf{u}\right]\right) - Q_{\mathrm{rad}} \qquad (4)$$

Conservation of vibrational energy:

$$\frac{\partial}{\partial t}\rho_i e_{v,i} + \nabla \cdot \rho_i e_{v,i}\mathbf{u} = \nabla \cdot [k_{v,i}\nabla T_{v,i}] - \nabla \cdot e_{v,i}\mathbf{J}_i + Q_{T-V_i} + Q_{V-V_i} + Q_{\mathrm{Chem}-V_i} - Q_{\mathrm{rad}_i} \qquad (5)$$

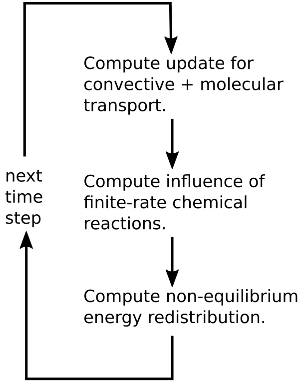
More maths...

Thermodynamic model of the gas...
Finite-rate chemical kinetics...
Radiation energy exchange...
Boundary conditions...
Features:

- 3D from the beginning, 2D as a special case
- structured- and unstructured-meshes for complex geometries
- refined thermochemistry
- moving meshes (Jason Qin and Kyle Damm)
- simplified and generalized boundary conditions
- coupled heat transfer
- shared-memory parallelism for multicore workstation use
- block-marching for speed (nenzfr and nozzle design)

Code structure

- D language data storage and solver, with embedded Lua interpreters for preprocessing, user-controlled run-time configuration in boundary conditions and source terms and thermochemical configuration.



for s=1 to n do:

- clear flux data
- apply pre-reconstruction action
- detect shock points
- reconstruct flow data at cell interfaces
- compute convective fluxes
- apply pre-spatial-derivative action
- compute spatial derivatves
- apply post-differential flux action
- add source terms if any
- compute time derivatives of conserved quantities
- update cell-average conserved quantities for stage s
- decode conserved quantities to all flow quantities

Collecting the low-hanging fruit of parallelism

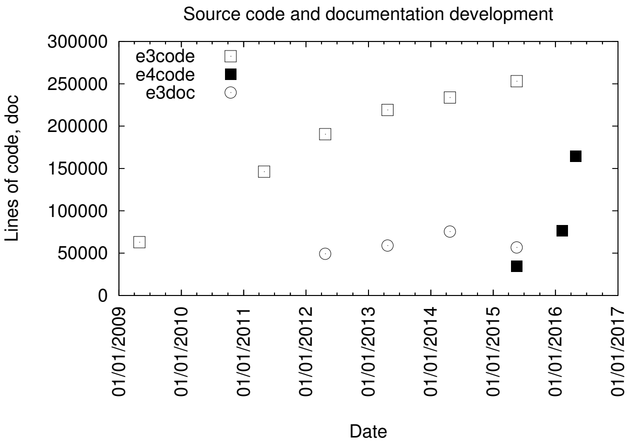
```
    // First-stage of gas-dynamic update.
    shared int ftl = 0; // time-level within the overall convective-update
    shared int gtl = 0; // grid time-level remains at zero for the non-moving grid
    if (GlobalConfig.apply_bcs_in_parallel) {
        foreach (blk; parallel(gasBlocks,1)) {
            if (blk.active) { blk.applyPreReconAction(sim_time, gtl, ftl); }
        }
    } else {
        foreach (blk; gasBlocks) {
            if (blk.active) { blk.applyPreReconAction(sim_time, gtl, ftl); }
        }
    }
```

Notes:

- Need to keep most data thread local.
- D Compiler expands “parallel” into code that hands out tasks to the default ThreadPool.

How far have we gone, in lines of source code.

At 60 lines per page,
the collection is equivalent to a 7500 page document.



Verification and Validation Examples

Verification:

- Are we solving the equations correctly?
- Compare with numerical solutions from other codes.
- Manufactured solution that we must match (using special source terms and BCs).

Validation:

- Are we solving the correct gas-dynamic equations?
- Compare with experimental measurements.

Example 1: sharp-nosed projectile

- Original Zucrow & Hoffman; also Anderson's Hypersonics text
- Shape of surface defined by polynomial equation
- Can compare numerical solutions

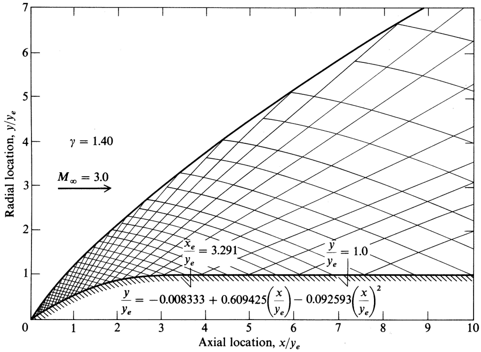
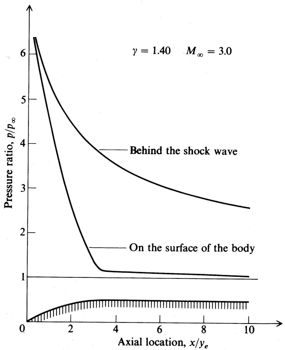


FIGURE 5.5
A typical characteristics mesh. (*From Zucrow and Hoffman, Ref. 53.*)



Input script – gas model and flow

```lua
-- sharp.lua
config.title = "Mach 3 flow over a sharp 2D body"
print(config.title)

nsp, nmodes = setGasModel('ideal-air-gas-model.lua')
print("GasModel set to ideal air. nsp= ", nsp, " nmodes= ", nmodes)
initial = FlowState:new{p=5955.0, T=304.0, velx=0.0, vely=0.0}
inflow = FlowState:new{p=95.84e3, T=1103.0, velx=2000.0, vely=0.0}

```

Notes:

- user's input script is Lua source code
- arguments to function calls delimited by ()
- tables delimited by {}
- object model by convention as described in Ierusalimschy's book "Programming in Lua"

Input script – user-defined functions

```lua
-- Geometry of flow domain.
function y(x)
   -- (x,y)-space path for x>=0
   if x <= 3.291 then
      return -0.008333 + 0.609425*x - 0.092593*x*x
   else
      return 1.0
   end
end

function xypath(t)
   -- Parametric path with 0<=t<=1.
   local x = 10.0 * t
   local yval = y(x)
   if yval < 0.0 then
      yval = 0.0
   end
   return {x=x, y=yval}
end
```

Notes:

- global variables unless stated otherwise
- can return tables

Input script – geometry definition

```
a = Vector3:new{x=-1.0, y=0.0}; b = Vector3:new{ x=0.0, y=0.0}
c = Vector3:new{x=10.0, y=1.0}; d = Vector3:new{x=10.0, y=7.0}
e = Vector3:new{ x=0.0, y=7.0}; f = Vector3:new{x=-1.0, y=7.0}
-- lower boundary including body surface
ab = Line:new{p0=a, p1=b}; bc = LuaFnPath:new{luaFnName="xypath"}
-- upper boundary
fe = Line:new{p0=f, p1=e}; ed = Line:new{p0=e, p1=d}
-- vertical lines
af = Line:new{p0=a, p1=f}; be = Line:new{p0=b, p1=e}
cd = Line:new{p0=c, p1=d}
-- Mesh the patches, with particular discretisation.
ny = 60
clustery = RobertsFunction:new{end0=true, end1=false, beta=1.3}
clusterx = RobertsFunction:new{end0=true, end1=false, beta=1.2}
grid0 = StructuredGrid:new{psurface=makePatch{north=fe, east=be, south=ab, west=af},
                           cfList={east=clustery, west=clustery},
                           niv=17, njv=ny+1}
grid1 = StructuredGrid:new{psurface=makePatch{north=ed, east=cd, south=bc, west=be},
                           cfList={north=clusterx,south=clusterx,west=clustery},
                           niv=81, njv=ny+1}
```

Notes:

- Table entries are mostly named. (new behaviour) This is an advantage for large numbers of parameters.
- Also, could import grids. Good for complex geometries because you may have your favourite gridding tool.

Input script – flow domain with boundary conditions

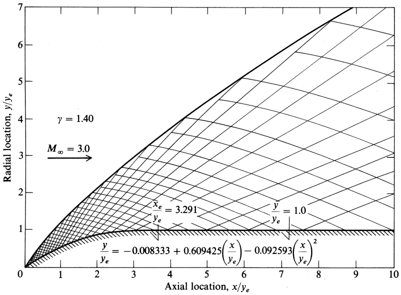
```
-- Define the flow-solution blocks.
blk0 = SBlock:new{grid=grid0, fillCondition=inflow}
blk1 = SBlock:new{grid=grid1, fillCondition=initial}
-- Set boundary conditions.
identifyBlockConnections()
blk0.bcList[west] = InFlowBC_Supersonic:new{flowCondition=inflow}
blk1.bcList[east] = OutFlowBC_Simple:new{}

config.max_time = 15.0e-3  -- seconds
config.max_step = 2500
config.dt_init = 1.0e-6
```

Notes:

- We have separated block definition from grid generation.
- fillCondition could be given as a (user-defined) function of position (x,y,z).
- Also, could provide lists of boundary conditions to the block constructors.

Result – pressure field



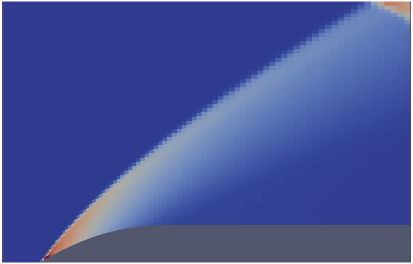
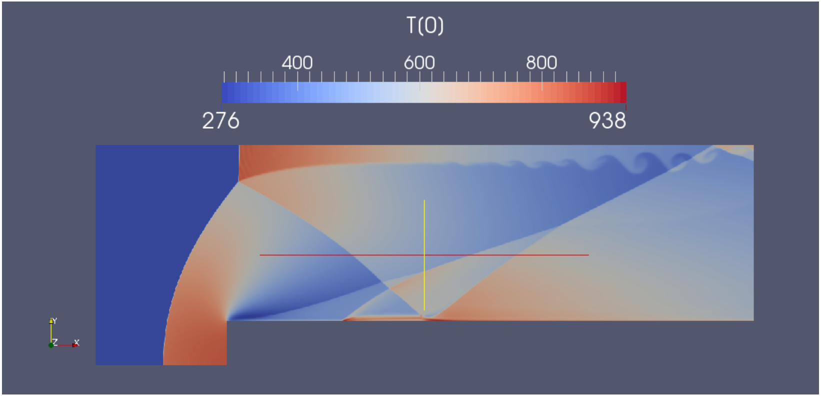


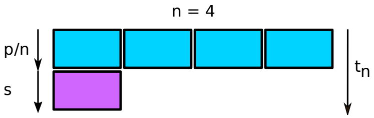
FIGURE 5.5
A typical characteristics mesh. (*From Zucrow and Hoffman, Ref. 53.*)

Example 2: supersonic flow over a forward-facing step



- To make good use of all of those processing cores, divide the flow domain into 21 blocks.
- There is an animation if we have time.

Scaling of run times when using multiple CPUs



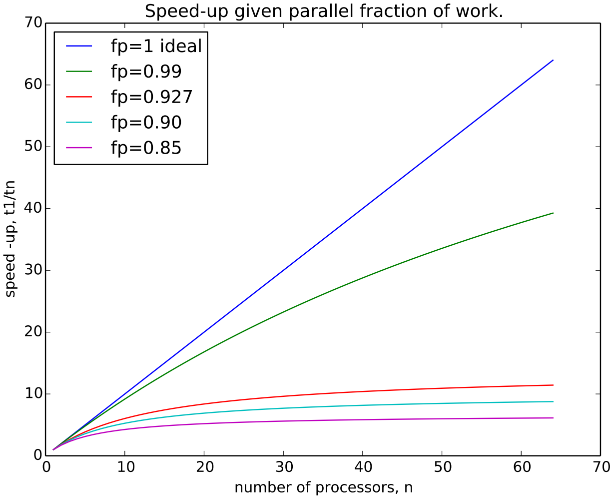

```
/* parallel fraction calculation, pj, 2016-05-24 */
eq0: p + s = t1;
eq1: p/na + s = ta;
eq2: p/nb + s = tb;
solve([eq0, eq1, eq2], [t1, p, s]);

```

```python
# fparallel.py
# Compute fraction of work done in parallel.
na = 3; ta = 3214.0-518.0
nb = 7; tb = 1904.0-453.0
p = -(na*nb*ta - na*nb*tb)/(na - nb)
s = (na*ta - nb*tb)/(na - nb)
t1 = p + s
fp = p/t1
print("p=", p, "s=", s, "t1=", t1, "fp=", fp)
```

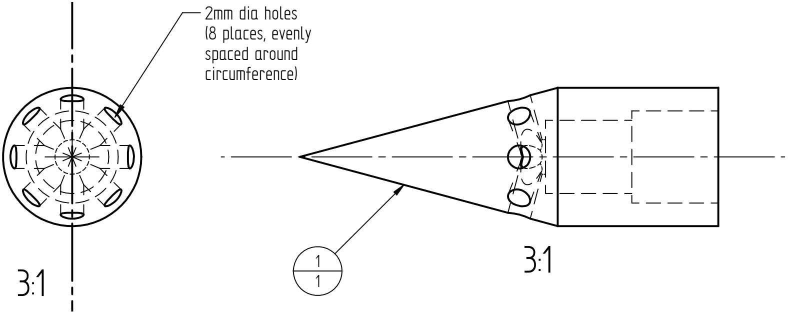
- Amdahl's model for serial and parallel work components with n processors.
- fp=0.922 for dx=2.5mm
- fp=0.927 for dx=1.25mm

Amdahl's scaling for parallel calculations



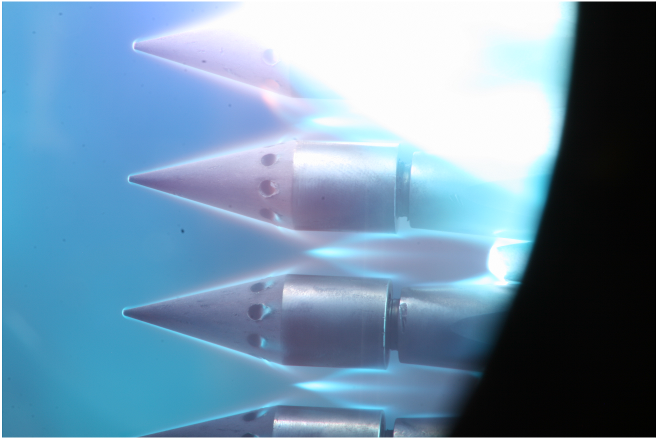
- Anand estimated fp=0.99 for Eilmer3, MPI, chemistry.
- fp=0.927 best so far for Eilmer4 with a 2D, inviscid flow.

Example 3: blunt cone-probe for expansion-tube flows



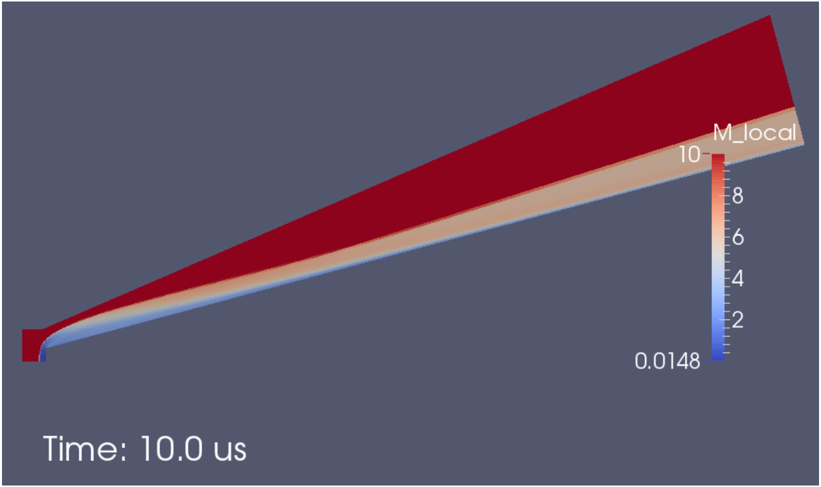
- Nice idea based on reducing the pressure to something less than a Pitot probe
- but Pierpaolo shows the measured pressure values to be something like half of the expected values.

Blunt-cone-probe in use in X2



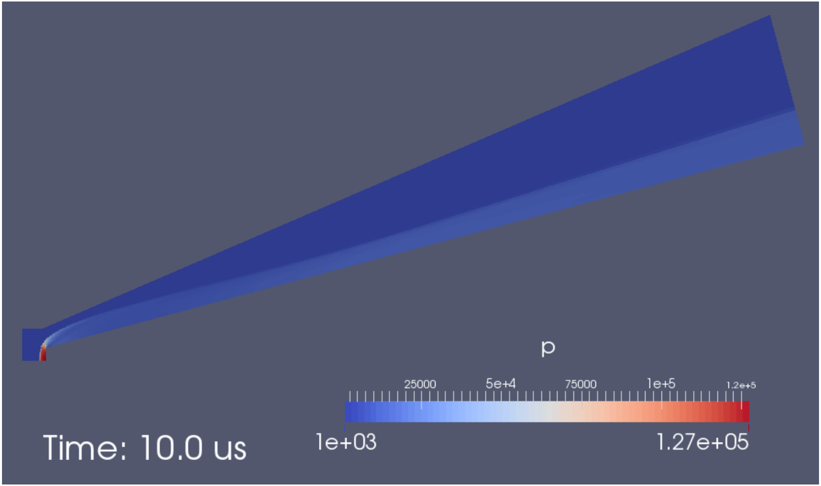
- Photograph from Steven Lewis, yesterday.

Blunt-cone-probe inviscid flow field – Mach number



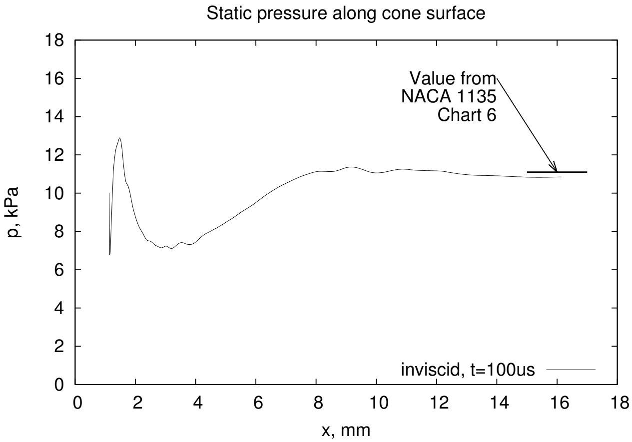
- After about 800 seconds, we have a computed flowfield...

Blunt-cone-probe inviscid flow field – pressure



- Note that the shock is far from conical
- so, the surface pressure may not be the Taylor-Maccoll value.

Blunt-cone-probe – surface pressure



- We should repeat this analysis with viscous effects included.
- Now, hurry up and show the animation.