Scientific Programming: Miscellaneous tricks, tips and lessons learned

Nick N. Gibbons,
The University of Queensland, Brisbane, Queensland 4072, Australia

February 28, 2020

Outline: Scientific Computing Group Seminar

- Linux Core Dev Toolkit
  - grep
  - The Editor Wars
  - history searching
  - rsync
- Advanced Paraview
  - Programmable Filters
  - Scripting interface
- numpy
  - ndarrays
  - logical operations
  - einsum
  - linear algebra
- Embedding C code in Python
  - A simple example: Particle Motion
  - A complicated example: Vortex Visualisation

Linux Core Dev Toolkit: grep

- Search text, count things, evaluate regular expressions

```
$ cat *.log | grep -c grep | awk '{ print $1/1035 }'
4.61256
```

- Normal Usage:

```
$ grep with_k_omega *.d
    shape_sensitivity_calc.d: //if (with_k_omega) nPrimitive += 2;
```

- Recursive Usage:

```
$ grep -r with_k_omega --include="*.d"
    fvcell.d:    bool allow_k_omega_update = true;
    bc/boundary_cell_effect.d:        blk.get_cell(i,j,k)
                   .allow_k_omega_update = false;
```

- Find repeated words with a regular expression:

```
$ grep -E "\b([a-zA-Z]+) \1\b" words.txt
```

Linux Core Dev Toolkit: The Editor Wars

Do you have strong opinions about text editors?

[✓ Yes]

No

- ViM: (Vi Improved)
  - Pros: Powerful, runs on everything, very configurable
  - Cons: Steep learning curve, hard to quit
- EMACS: (Exclusively used by Middle Aged Computer Scientists)
  - Pros: More intuitive than vim, also powerful and configurable
  - Cons: Also steep learning curve, destroyer of alt/ctrl/shift keys
- Sublime Text:
  - Pros: Feature packed, easy to use, looks nice
  - Cons: Closed source, (winzip style?)
- Others: Atom/Nano/etc.

Linux Core Dev Toolkit: History Searching

- Commands typed into the linux commandline are saved in `.bash_history`
- `.inputrc` can be configured to search through these while typing

```
$ make_
$ make INSTALL_DIR=/home/nick/programs/dgd/dmd install_
```

- Specific commands for UP and DOWN to search are:

```
"\e[A": history-search-backward
"\e[B": history-search-forward
```

- This automatically works with other programs that use libreadline
  - ipython3 etc.

Linux Core Dev Toolkit: rsync

- rsync is a powerful file copying utility
- Use 1: Copy things to another computer over ssh
- Use 2: archiving data (with -tarvPp)

```
rsync -tarvPp --delete /home/qungibbo/ /media/qungibbo/Elements/
WintermuteSSD --exclude "*.ssh" --exclude "programs/*"
--exclude "sourcecode/us3d*" --exclude "sourcecode/paraview*"
--exclude "sourcecode/cfcfd3/*" --exclude "sourcecode/hdf5-1.8.12/*"
--exclude "sourcecode/openmpi-1.8.12/*" --exclude "*.vtk"
--exclude "*.o" --exclude "*.mod" --exclude ".cache*"
--exclude ".wine/*" --exclude "*.swp" --exclude "*.pvtu"
--exclude "*.vtu" --exclude "*.npy"

rsync -tarvPp --delete /media/qungibbo/data/ /media/qungibbo/Elements/
WintermuteDATA --exclude "binaries/*" --exclude "*.vtk" --exclude "*.o"
--exclude   "*.mod" --exclude "*.swp" --exclude "*.pvtu"
--exclude "*.vtu" --exclude "*.npy"
```

Advanced Post Processing: Paraview Programmable Filters

- Paraview workflow uses "Filters" to process data
- Common Filters: Slice, Gradients, Calculator
- "Programmable Filter" is a custom filter that uses a python script to do anything
- "Can I do-" "Yes"

Example to compute vorticity vector:

```
du = zeros((neq,neq,nx)) # du[i,j] = du_i / dx_j
du[0] = algs.gradient(u).T
du[1] = algs.gradient(v).T
du[2] = algs.gradient(w).T

vort = zeros((neq,nx))
vort[0] = du[2,1] - du[1,2]
vort[1] = du[0,2] - du[2,0]
vort[2] = du[1,0] - du[0,1]
```

Advanced Post Processing: Paraview Programmable Filters

- More complicated example: Mixing Efficiency

```
splist = "N2,O2,H2,H2O,OH,HO2,H2O2,NO,NO2,HNO,N,H,O".split(',')
gassp = {n:get_species(n) for n in splist}
rhos = [get_array('r{}m'.format(i+1)) for i in range(len(splist))]
rho = sum(rhoi for rhoi in rhos)
cs  = [rhoi/rho for rhoi in rhos]

# Compute the fraction of mass of each atom made up by H or O, excluding an
involving N
atomicmass = lambda s,atom : sum(gassp[k].M*v for k,v in gassp[s].atoms.ite
k==atom)
mH =[0.0 if 'N' in s else atomicmass(s,'H')/gassp[s].M for s in splist]
mO =[0.0 if 'N' in s else atomicmass(s,'O')/gassp[s].M for s in splist]
YO = sum(mOi*csi for mOi,csi in zip(mO, cs))
YH = sum(mHi*csi for mHi,csi in zip(mH, cs))

Ytotal = YO + YH # Mass fraction of the flow that is interesting from a mix
YYH = YH/Ytotal  # Mass fraction fraction of the interesting flow that is H
YYO = YO/Ytotal  # Mass fraction fraction of the interesting flow that is O
massst = 0.5*gassp['O2'].M/(1.0*gassp['H2'].M) # Stoichiometric Oxygen/hydr

# Fraction of interesting flow fraction that is mixed
YYmixed = np.minimum(YYH*(1.0 + massst), YYO*(1.0 + 1.0/massst))
```

Advanced Post Processing: Paraview Scripting

- Paraview has an extremely flexible python scripting interface
- You can automate ANYTHING
- Click "Start Trace" in "Tools", do something, then "Stop Trace"

```
from paraview.simple import *
from glob import glob
from sys import argv

scriptname = argv[1]
pattern = argv[2]
files = glob(pattern)
files.sort()

with open(scriptname) as fp:
     script = fp.read()

for i,filename in enumerate(files):
    print "Computing␣file:␣", filename, scriptname

    soln = OpenDataFile(filename)
    prgfil = ProgrammableFilter(soln)
    my_script = script.replace('REPLACE',str(i).zfill(3))
    prgfil.Script = my_script

    Show(prgfil)
    Delete(prgfil)
    Delete(soln)
```

numpy: Fast numerical computation in python

- numpy is core utility library for numerical computation in python
- ndarray objects are extremely powerful but can be confusing at first
- Their main use is looping without using very slow interpreted "for" loops

```
In[9]: %timeit for i in range(1000): c[i] = a[i]*c[i]
10000 loops, best of 3: 122 us per loop

In[13]: %timeit c = a*b
1000000 loops, best of 3: 1.06 us per loop
```

numpy: Tips and tricks

- Vectorisation Example: Logic
- Copy the smaller element to array c

```
a = random.random(1000)
b = random.random(1000)

# Slow:
c = []
for i,j in a,b:
    if i<j:
        c.append(i)
    else:
        c.append(j)

# Fast
d = a<b             # bool array of true falses
c = d*a + (1-d)*b # acts like 1 or zero in arithmatic
```

numpy: Tips and tricks

- Vectorisation Example: Matrix Multiplication
- Matrix multiply a large number of matching matrices with einsum

```
A = random.random(100*3*3).reshape((100,3,3))
B = random.random(100*3*3).reshape((100,3,3))

# Slow:
C = []
for i in A.shape[0]:
    C.append(A[i].dot(B[i])

# Fast
C = einsum('lij,ljk->lik', A, B)
```

numpy: Tips and tricks

- Vectorisation Example: Linear Algebra
- Solve a large number of algebra problems

```
from numpy.linalg import eig
A = random.random(100*3*3).reshape((100,3,3))

# Slow:
evals = []
evecs = []
for i in A.shape[0]:
    eval,evec  = eig(A[i])
    evals.append(eval)
    evecs.append(evecs)

# Fast
evals, evecs = eig(A)
```

numpy: Tips and tricks

- Always check the library functions!
- If you need it chances are someone else has too
- Vectorisation Example: binning data for a histogram

```
>>> x = np.array([0.2, 6.4, 3.0, 1.6])
>>> bins = np.array([0.0, 1.0, 2.5, 4.0, 10.0])
>>> inds = np.digitize(x, bins)
>>> inds
array([1, 4, 3, 2])
>>> for n in range(x.size):
...   print(bins[inds[n]-1], "<=", x[n], "<", bins[inds[n]])
...
0.0 <= 0.2 < 1.0
4.0 <= 6.4 < 10.0
2.5 <= 3.0 < 4.0
1.0 <= 1.6 < 2.5
```

Coupling C code with Python

- What to do when your application really is too complex for numpy
- Python builtin library ctypes allows you call c code from python
- Simple Example: Integrating Particle Motion
- Complicated Example: The Triple Decomposition Method

Coupling C code with Python: Particle Motion

Pure Python Implementation of a Falling Particle:

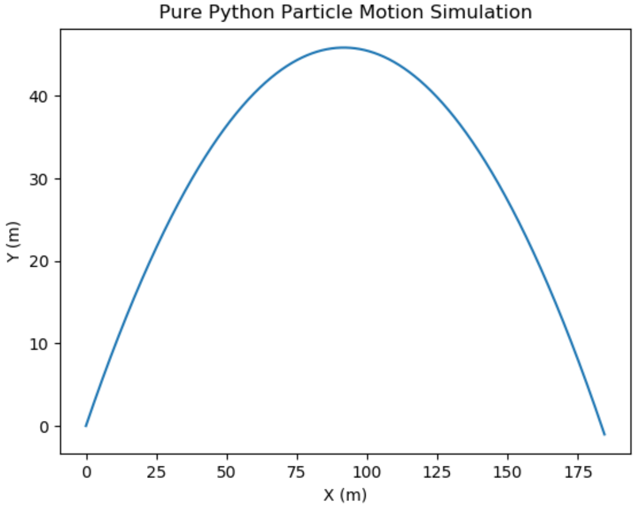
```python
dt = 0.0001
N = 61500
x  = array([0.0, 0.0])
xd = array([30.0, 30.0])
xdd= array([0.0, -9.81])
xs = zeros((N,2))

start = time.clock()
for i in range(N):
    x += xd*dt + 0.5*xdd*dt**2
    xd += xdd*dt
    xs[i] = x.copy()

end = time.clock()
print("Time␣for␣N={}:␣{:4.4f}ms".format(N, (end-start)*1000.0))
```

Coupling C code with Python: Particle Motion

Pure Python Implementation of a Falling Particle:



Coupling C code with Python: Particle Motion

Hybrid Implementation of a Falling Particle: Python Side

```
dt = 0.0001
N = 61500
x  = array([0.0, 0.0])
xd = array([30.0, 30.0])
xdd= array([0.0, -9.81])
xs = zeros((N,2))

c_double_p = POINTER(c_double)
xp = x.ctypes.data_as(c_double_p)
xdp = xd.ctypes.data_as(c_double_p)
xddp = xdd.ctypes.data_as(c_double_p)
xsp = xs.ctypes.data_as(c_double_p)
dt_c = c_double(dt)
lib = cdll.LoadLibrary('libodeint.so')

start = time.clock()
lib.integrate_particle(xp, xdp, xddp, xsp, N, 2, dt_c)
end = time.clock()

print("Time␣for␣N={}:␣{:4.4f}ms".format(N, (end-start)*1000.0))
```

Coupling C code with Python: Particle Motion

Hybrid Implementation of a Falling Particle: C side

```c
void integrate_particle(double* x, double* xd, double* xdd, double* xs,
                        int N, int  dim, double dt){
    // Simple fixed timestep integrator
    for(int i=0; i<N; i++) {
        for(int d=0; d<dim; d++) {
            x[d] += xd[d]*dt + 0.5*xdd[d]*dt*dt;
            xd[d] += xdd[d]*dt;
            xs[i*dim + d] = x[d];
        }
    }
    return;
}
```

Compare the Python Loop:

```python
for i in range(N):
    x += xd*dt + 0.5*xdd*dt**2
    xd += xdd*dt
    xs[i] = x.copy()
```

Coupling C code with Python: Particle Motion

Compiling C code into a dynamic library:

```
$ gcc -c -fPIC odeint.c
$ gcc -shared odeint.o -o libodeint.so
```

Speedup is significant:

```
nick@spark:~/code/randomcrap$ python3 pyshot.py
Time for N=61500: 461.6890ms

nick@spark:~/code/randomcrap$ python3 cshot.py
Time for N=61500: 1.2420ms
```

Coupling C code with Python: Vortex Visualisation

- There are many vortex visualisation techniques that use the decomposed velocity gradients:

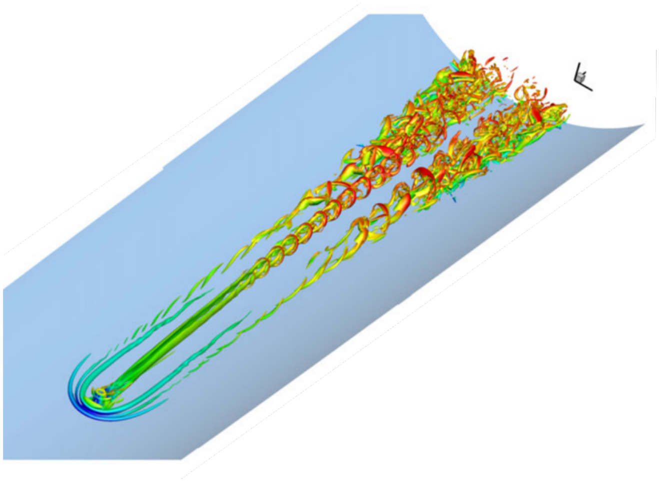


Figure 1: Vortex visualisation using the Q criterion, from Subbareddy, Bartkowicz, and Candler (2015), figure 9

Coupling C code with Python: Vortex Visualisation

- In subsonic flow this is pretty straightforward, first compute S and Ω

$$\frac{\partial u_i}{\partial u_j} = S_{ij} + \Omega_{ij} \tag{1}$$

$$S_{ij} = \frac{1}{2}\left[\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right] \quad \Omega_{ij} = \frac{1}{2}\left[\frac{\partial u_i}{\partial x_j} - \frac{\partial u_j}{\partial x_i}\right] \tag{2}$$

- Then pick a vortex identification method, eg:
  - Q criterion: $Q = \frac{1}{2}(||\Omega||^2 - ||S||^2) > 0$
  - Δ-criterion: $(Q/3)^3 + (det(\nabla u)/2)^2 > 0$
  - λ_2 method: $eigvals(\Omega^2 + S^2) = \lambda_1, \lambda_2, \lambda_3 \quad : \quad \lambda_2 < 0$

Coupling C code with Python: Vortex Visualisation

- In supersonic flow these can be cluttered with shockwaves since the shearing motion is incorrectly included as if it were a vortex
- The Triple Decomposition Method subtracts out a pure shearing term:

$$\frac{\partial u_i}{\partial u_j} - \left(\frac{\partial u_i}{\partial u_j}\right)_{SH} = S_{ij}^{TDM} + \Omega_{ij}^{TDM} \tag{3}$$

- Once $(\nabla u)_{SH}$ is found plug $S_{ij}^{TDM} + \Omega_{ij}^{TDM}$ into your vortex identification method
- How to compute $(\nabla u)_{SH}$?

Coupling C code with Python: Vortex Visualisation

- Computing $(\nabla u)_{SH}$ requires a nasty optimisation problem in every cell
- Rotate the axes $\nabla u \rightarrow \nabla u'$ and find the angles where:

$$MAX(|S'_{12}\Omega'_{12}| + |S'_{23}\Omega'_{23}| + |S'_{31}\Omega'_{31}|) \tag{4}$$

- In this reference frame it is possible to compute $(\nabla u)_{SH}$
- Then subtract out $S_{ij}^{TDM} + \Omega_{ij}^{TDM}$ and convert back into lab frame

Coupling C code with Python: Vortex Visualisation

- Pure python implementation is EXTREMELY SLOW (≈ 30 hours to execute)
- Instead call a C routine from inside Paraview to do the same thing
- This takes ≈ 20 mins or so, and can be done in parallel

Coupling C code with Python: Vortex Visualisation

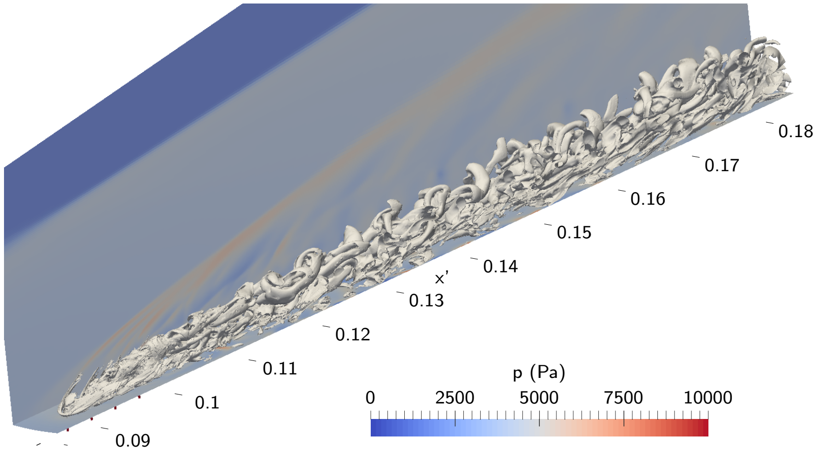


Figure 2: Vortex visualisation using TDM and the L2 method with pressure colour maps.

Conclusions

- Command line tools are very powerful, learn to use them!
- Get comfortable with a good text editor
- Interpreted languages (Python,R) are great for scientific computing
  - As long as you use the datastructures properly!
  - Both python and R come packed with builtin functions too
- If you need to fall back on some low-level programming, (C etc.) tools exist to integrate them into higher level languages.