Computational Physics
Lecture 9

sieversj@ukzn.ac.za

git clone https://github.com/ukzncompphys/lecture7.git
wget www.cita.utoronto.ca/~sievers/compphys/lecture7.tar.gz
(followed by tar -xzf lecture7.tar.gz)

Time Steps in Advection

- Smaller time step normally more accurate.
- Let's look at solution for some different time steps.
- What happened?
- Behaviour of sharp features often very important - in practice, run test problems with known solutions to verify behaviour.

```
#advect_finite_volume_timestep.py
dt=1.0
big_rho=numpy.zeros(n+1)
big_rho[1:]=rho
del rho  #we can delete the to save space
oversamp=10 #let's do finer timestamps
dt_use=dt/oversamp
for step in range(0,150):

    big_rho[0]=0
    for substep in range(0,oversamp):
        drho=big_rho[1:]-big_rho[0:-1]
        big_rho[1:]=big_rho[1:]-v*dt_use/dx*drho

    plt.clf()
    plt.axis([0,n,0,1.1])
    plt.plot(x,big_rho[1:])
    plt.draw()
```

2nd Order Advection

- Solution is not very accurate. Where did errors come from?
- Density has changed during a timestep. We ignored that.
- Density within a cell can be non-constant. We ignored that.
- For a *smooth* flow, we can Taylor expand solution, use higher-order derivative approximation, then get more accurate solution.
- Simplest extension is to use the half-way point in time for the time step, and 1st order slopes within cell.

2nd order ctd.

- To be more accurate, we want to know flow at left and right edges of cell at the half-way point in time.
- $F(x+dx/2, t+dt/2)=F(x,t)+\partial F/\partial x*dx/2+\partial F/\partial T*dt/2$
- From advection equation, $\partial F/\partial t+u\partial F/\partial x=0$, so remove $\partial F/\partial t$
- $=F(x,t)+\partial F/\partial x*dx/2-u\partial F/\partial x*dt/2$
- $=F(x,t)+dx/2*\partial F/\partial x*(1-udt/dx)$
- similarly, left edge $F(x-dx/2,t+dt/2)=F(x,t)-dx/2*\partial F/\partial x*(1+udt/dx)$

How Many Ghost Cells?

- To know flow rate, I need to use a derivative. Simplest good estimate is $\partial F/\partial x=[F(x+dx)-F(x-dx)]/2$
- So, I need to know my neighbors to know my derivative.
- My neighbor flows into me. So I need to know its flow rate at boundary.
- It needed to know its neighbors for derivative, so I feel 2 cells away.
- This means at edge of domain, I need 2 ghost cells.
- Generally, the more terms in a Taylor series we solve exactly, the more ghost cells we'll need.

Riemann Problem

- In general, cells do not need to agree at the boundary.
- Figuring out what happens at interface can be complicated. This is called the Riemann problem, and any fluid-like code needs a Riemann solver.
- For advection, flow is in one direction, so we can just take conditions from upstream cell.
- For Euler equations, recall we have 3 eigenvalues - u and $u \pm c_s$. So there are 3 waves.
- To solve Riemann problem, need to solve for *each* wave. Stuff will possibly flow in both directions across boundary.

Final 2nd Order Advection

- Now we have pieces to do 2nd order advection.
- Need to get boundary conditions - 2 cells now instead of 1
- Need to calculate timestep - what fraction of Courant # do we use?
- Need to calculate interface properties
- Need to solve Riemann problem (in this case take upstream interface)
- Need to evolve our solution. Let's watch…

BC/Interface - advect2.py

```
def get_bc(self):
    #we now need to put in 2 ghost cells.  These are periodic
    self.rho[0:2]=self.rho[-4:-2]
    self.rho[-2:]=self.rho[2:4]
def get_interfaces(self):
    #calculate the derivatives for the cell centers
    self.myderiv=0.5*(self.rho[2:]-self.rho[0:-2])/self.dx
    #now calculate the flux at the right and left edges of the cell
    #after taking half a step forward in time.
    self.right=self.rho[1:-1]+0.5*self.dx*(1.0-self.C)*self.myderiv
    self.left=self.rho[1:-1]-0.5*self.dx*(1.0+self.C)*self.myderiv
```

- Boundary conditions are 2 cells wide now.
- Interface conditions: calculate derivative using left and right neighbours.
- Then use 1st order estimates of flux at cell boundary.

Riemann/Update for 2nd order.

```
    def update(self):
        dt=self.get_ts()
        self.get_bc()
        self.get_interfaces()
        #Solve Riemann problem, and update.  For advection, if velocity is positive,
        #cell to my left flows into me with the flux at its right edge, and I flow into
        #the cell to my right with my flux at my right edge.
        if self.u>0:
            self.rho[2:-2]=self.rho[2:-2]+(self.right[0:-2]-self.right[1:-1])*self.C
        else:
            self.rho[2:-2]=self.rho[2:-2]+(self.left[1:]-self.left[0:-2])*self.C
```

- to take a timestep, need to get the boundary and interface conditions.
- For advection, Riemann problem is just deciding to take left or right flux. Fold this into time update.
- After checking sign of velocity, update the density.
- Indexing is funny because derivative (hence interface) vector chops off one element from either end.

Finally Have all the Pieces

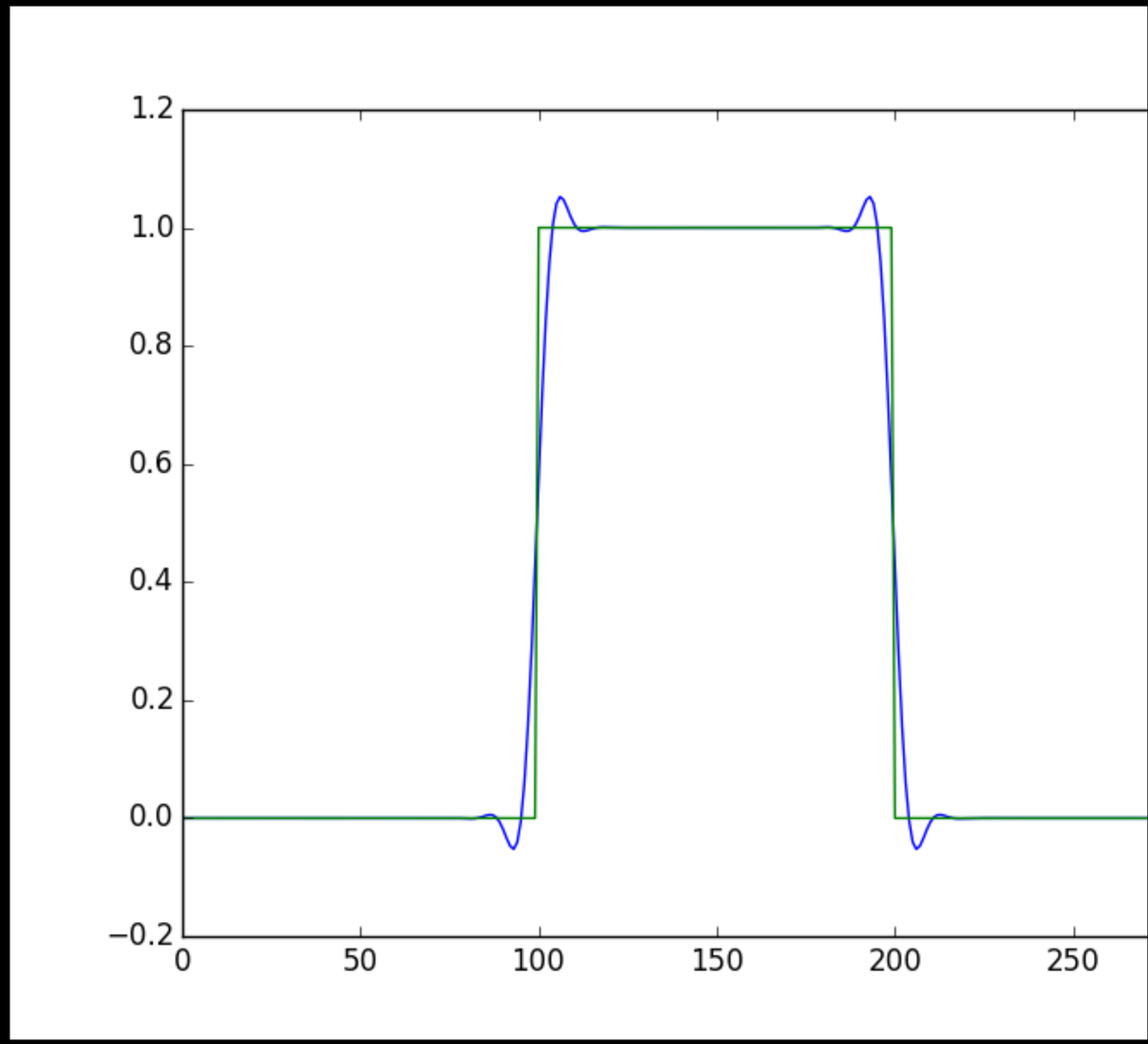
```
fac=2;
npt=300
C=1.0/fac
fwee=advect(n=npt,C=C)
fwee_org=advect(n=npt,C=C)
plt.ion()

#we have 2 ghost cells on either side, so only have npt-4 true cells
#if we want to get back to the initial state for periodic BC's.
for i in range(0,npt-4):
    for ii in range(0,fac):
        fwee.update()
    if i%10==0:  #to plot very often, makes code much faster.
        plt.clf()
        plt.plot(fwee.rho)
        plt.draw()
print numpy.mean(numpy.abs(fwee.rho-fwee_org.rho))
```

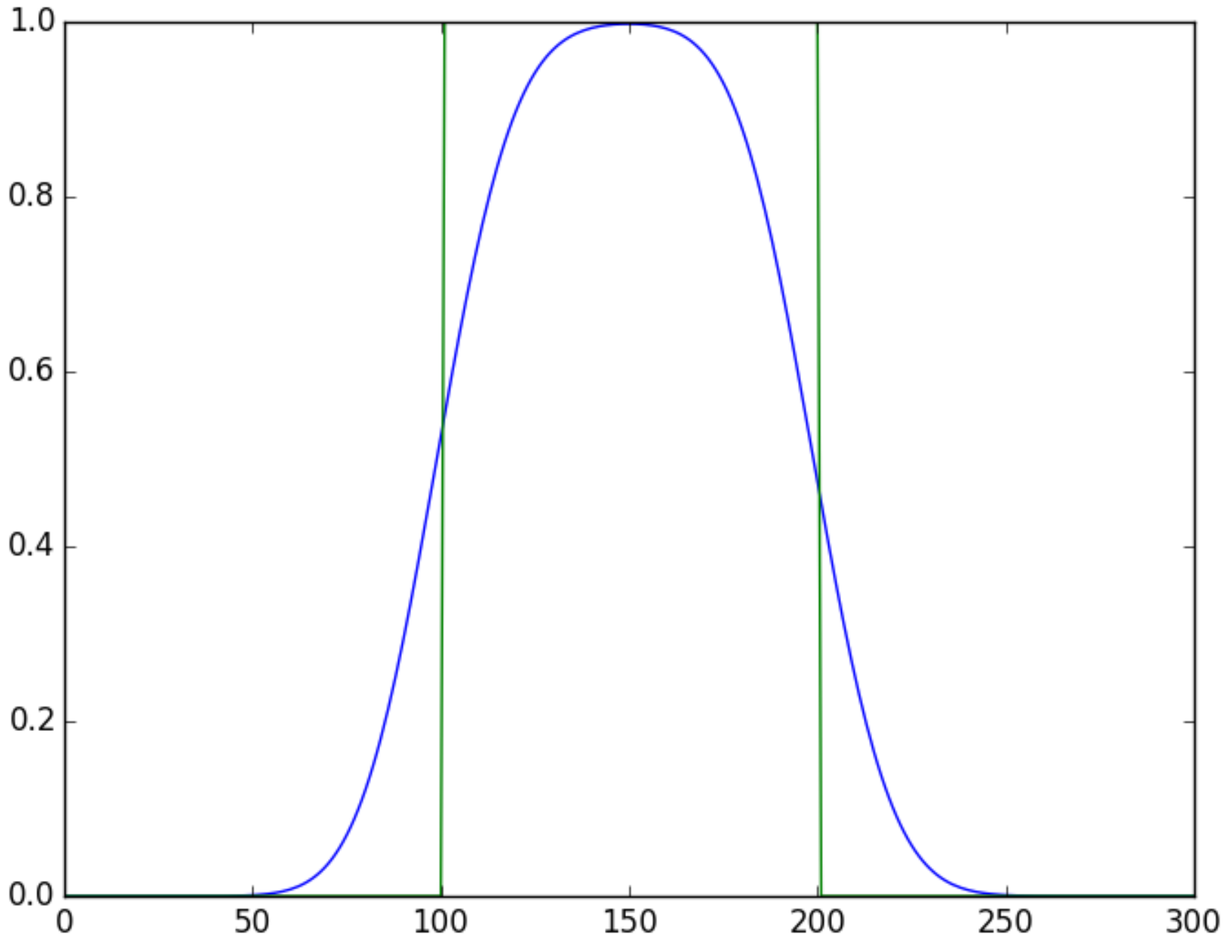
- With periodic boundary conditions, we can run code for 1 full loop.
- Ideally, should get back what we put in. How well does this work?

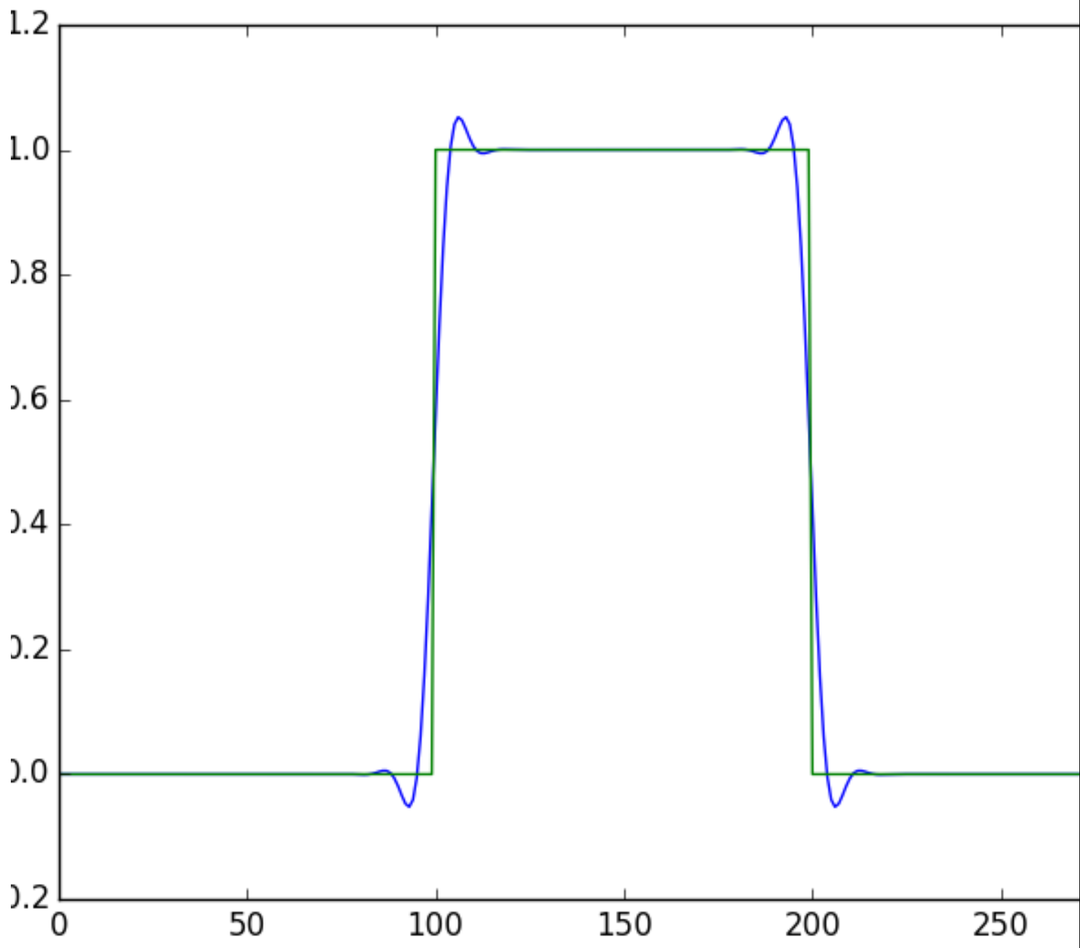
Overshooting

- After 1 full cycle, we have made some bonus dimples. Not ideal.
- Have we done better than 1-d?



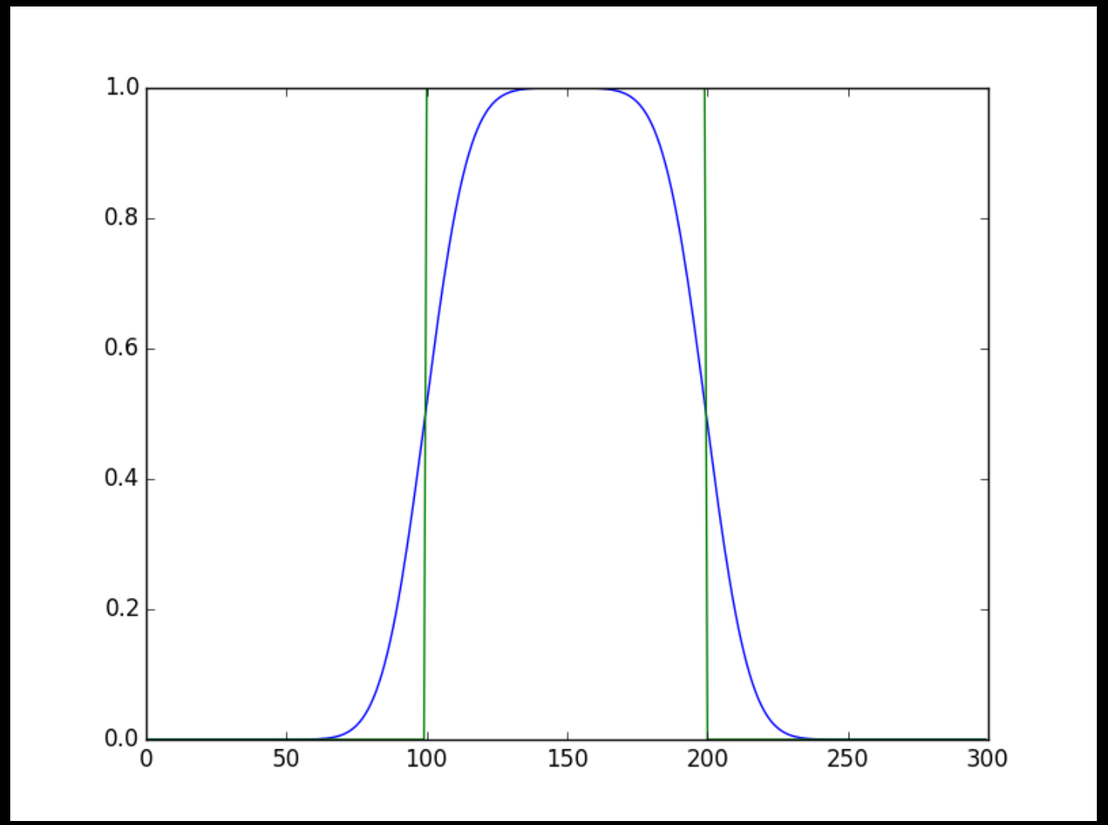
Overshooting





Limiting

- Godunov's theorem: *any* PDE solver which does not introduce new minima/maxima can be at most 1st order accurate.
- Can't have second order scheme that doesn't introduce spurious maxima
- If you don't like them, can do *limiting*. In this case, when calculating $\partial F/\partial x$, get derivative to both left neighbour and right neighbour. If same sign, take smaller of the two (in magnitude). If opposite sign, set derivative to zero
- This is called minimod limiting.



Unit Testing

- As codes get more complicated, making sure they do the right thing gets tricky.
- One approach: break down problem. Write tests that call a pieces of the code (routine, class method, etc.), check output against known behaviour.
- This is called unit testing, and is a good thing.

Vectorizing

- Python is interpreted. Each line has overhead. So, simple *for* loops are slow.
- Numpy is vectorized. Those calls go straight to compiled (i.e. much faster) code.
- Code runs much faster when vectorized.

Vectorized vs. Non-Vectorized

```
    def get_interfaces_slow(self):
        #calculate the derivatives for the cell centers
        self.myderiv=numpy.zeros(self.n-2)
        for i in range(0,self.n-2):
            self.myderiv[i]=0.5*(self.rho[i+2]-self.rho[i])/self.dx

        #now calculate the flux at the right and left edges of the cell
        #after taking half a step forward in time.
        self.right=numpy.zeros(self.n-2)
        self.left=numpy.zeros(self.n-2)
        for i in range(0,self.n-2):
            self.right[i]=self.rho[i+1]+0.5*self.dx*(1.0-self.C)*self.myderiv[i]
            self.left[i]=self.rho[i+1]-0.5*self.dx*(1.0+self.C)*self.myderiv[i]
```

```
    def get_interfaces(self):
        #calculate the derivatives for the cell centers
        self.myderiv=0.5*(self.rho[2:]-self.rho[0:-2])/self.dx

        #now calculate the flux at the right and left edges of the cell
        #after taking half a step forward in time.
        self.right=self.rho[1:-1]+0.5*self.dx*(1.0-self.C)*self.myderiv
        self.left=self.rho[1:-1]-0.5*self.dx*(1.0+self.C)*self.myderiv
```

- top is non-vectorized code. we explicitly loop through each element
- bottom is vectorized.

Timing Results

- You can get times from the *time* module. time.clock() will give you elapsed time since some arbitrary time. only differences are valid.
- Vectorized code ran nearly 50x faster!

```
t1=time.clock()
for i in range(0,100):
    fwee.update()
t2=time.clock()
t1_slow=time.clock()
for i in range(0,100):
    fwee_slow.update_slow()
t2_slow=time.clock()
print "fast time is " + repr(t2-t1)
print "slow time is " + repr(t2_slow-t1_slow)
print "error is " + repr(numpy.mean(numpy.abs(fwee.rho-fwee_slow.rho)))
```

```
>>> execfile('advect2_nonvector.py')
fast time is 0.004623000000000044
slow time is 0.2147460000000001
error is 0.0
```

Profiling

- Good to know where you code is spending its time.
- A simple profiling tool is cProfile. You can call from the command line
- Try: python -m cProfile -s tottime advect2.py > prof.txt
- now look at prof.txt. Where are we spending our time?

Tutorial Problems

- Add Gaussian initial conditions to both 1st and 2nd advection. For 1st order, write unit test to check single timestep with Fourier mode. Write test to check error. How does error scale? (10)

- Introduce a minimod limiter into the second order solver. Did it get rid of the overshooting? How does accuracy scale with grid resolution now? (10)