Computational Physics
Lecture 7

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)

Tutorial

- Let's add some methods to the class from the last tutorial so we can use it in an n-body simulation. First, add a method to initialize # of particles witih random positions in 2-D (numpy.random.randn() will get gaussian random numbers). (5)

- Next, write a method that calculates the forces on the particles using a softened potential (10)

- Now write a method that will update the particle positions and velocities using a timestep. (5)

- Finally, plot the total kinetic and potential energies as a function of time, and show that the total energy is approximately conserved (5)

PDE’s

- Partial differential equations are ubiquitous in nature
- Solving PDE’s on computers is a huge industry
- Several different techniques are used, each with advantages/disadvantages
- Diffusion, fluid flow, wave propagation, many many others examples of PDE’s solved on computers.

Advection Problem

- Next week we will do fluid mechanics. Many of the computational issues can be seen more simply through *advection*, which we will look at today.

- Imagine we have a velocity field v and a density field ρ (could be matter, could be something else).

- In advection, there are no internal forces/viscosities etc. The material just goes with the flow. Velocity is constant and field is conserved.

- Good source is tutorial from Mike Zingale, online at http://bender.astro.sunysb.edu/hydro_by_example/CompHydroTutorial.pdf

Some Techniques

- What should code even look like? Two broad classes:
- Eulerian: decompose space into domains (e.g. on a grid). Solve for $\rho(r,t)$, $v(r,t)$, etc.
  - Finite difference - function defined on grid cells
  - Finite volume - each cell covers a finite volume, value in cell is "average" of quantity across volume.
- Lagrangian: follow discrete packets of mass ("particles") through flow
  - Smoothed particle hydrodynamics (SPH)

Eulerian Visualizations

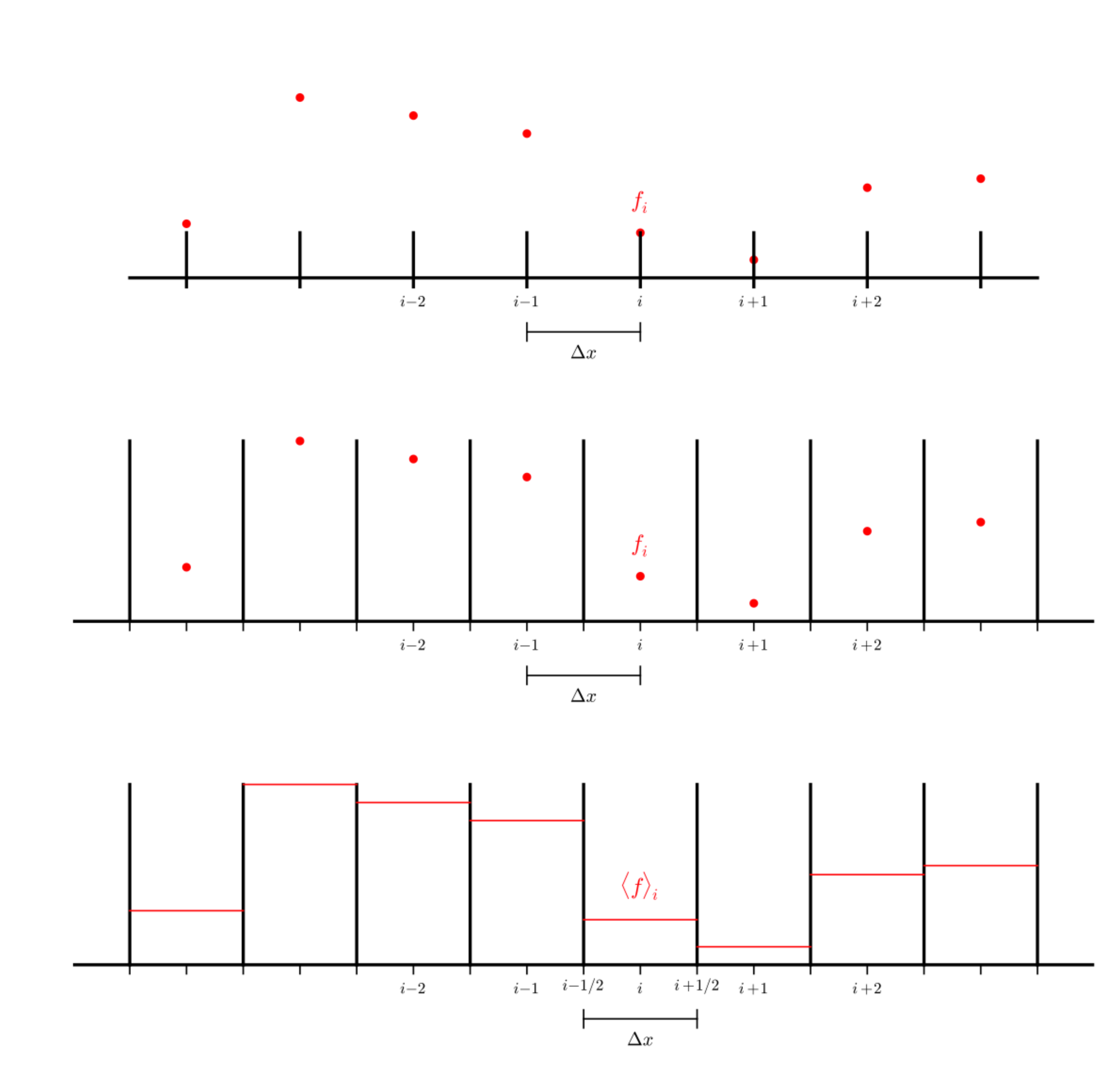


Figure from Zingale

- Top - finite difference. Function defined at grid points.
- Middle - finite difference, but with function defined at grid centers.
- Bottom - finite volume - function value is average across cell.

Finite Volume Advection

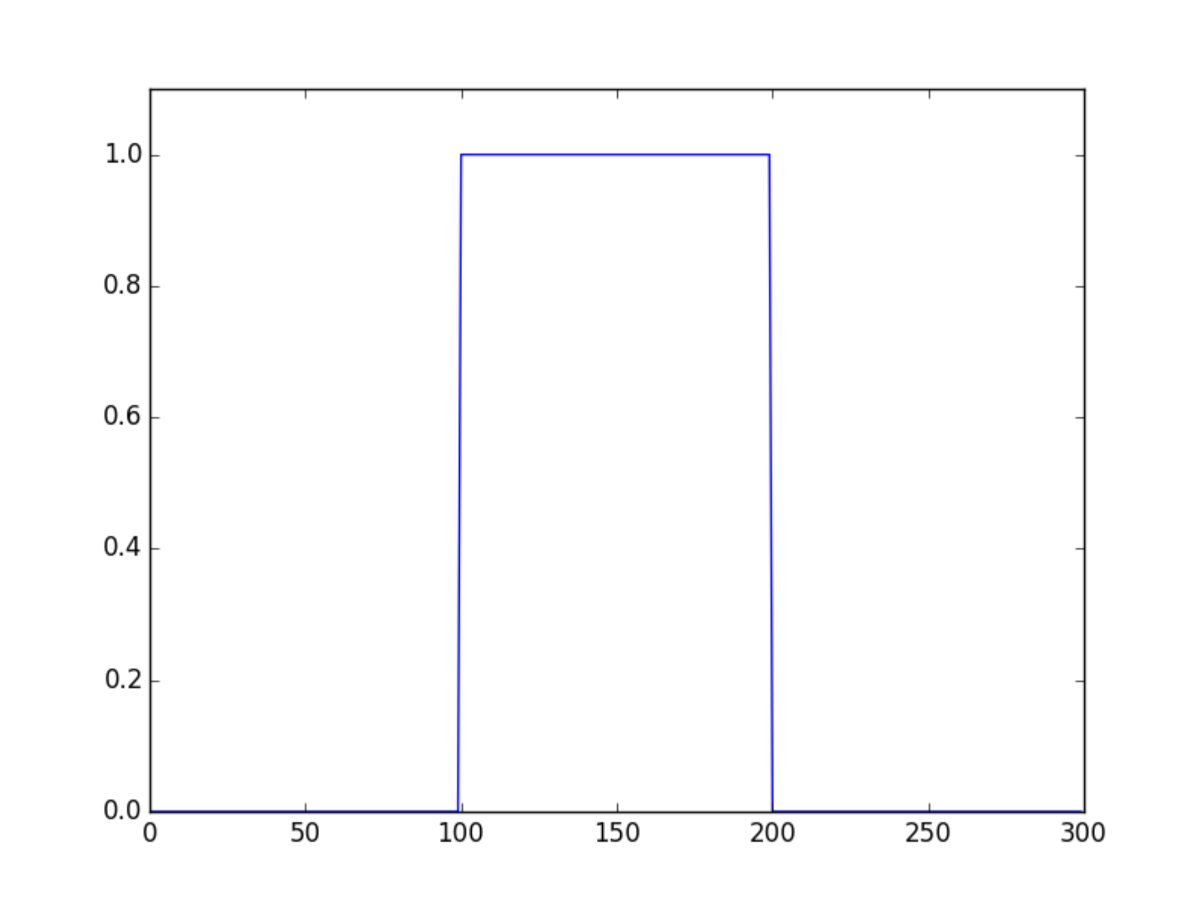
- Have density ρ_i and velocity v, with velocity taken to be uniform & constant for all grid cells.
- How does density change with time?
- Assume velocity is to the right. I flow into cell to my right, cell to my left flows into me.
- In (short time) dt flow moves *v*dt to the right. Cell is dx wide, so fraction of material that leaves cell is vdt/dx, total amount is $\rho_i v\, dt/dx$.
- Material flowing in is similarly $\rho_{i-1} v dt/dx$.
- New value is $\rho_i^{new} = \rho_i - \rho_i v dt/dx + \rho_{i-1} v dt/dx$

Finite Volume Advection

```
#simple_advect_finite_volume.py
import numpy
from matplotlib import pyplot as plt
n=300
rho=numpy.zeros(n)
rho[n/3:(2*n/3)]=1
v=1.0
dx=1.0
x=numpy.arange(n)*dx

plt.ion()
plt.clf()
plt.plot(x,rho)
```

Left: set up initial conditions. Density is 1 in the middle third of region, zero otherwise. Below left: initial density plotted. Bottom: advection code.




```
dt=1.0
for step in range(0,50):
    #take the difference in densities
    drho=rho[1:]-rho[0:-1]
    #update density.  We haven't said what happens at
    #cell 0 (since cell -1 doesn't exist), ignore for now
    rho[1:]=rho[1:]-v*dt/dx*drho
    plt.clf()
    plt.plot(x,rho)
    plt.draw()
```

Conservation

- New value is $\rho_i^{new}=\rho_i$ - $\rho_i$vdt/dx + ρ_{i-1}vdt/dx
- But cell i+1 looks the same, with i—> i+1: $\rho_{i+1}^{new}=\rho_{i+1}$ - ρ_{i+1}vdt/dx + $\rho_i$vdt/dx
- if I sum - $\rho_i^{new}+\rho_i^{new} =$ ρ_i - $\rho_i$vdt/dx + ρ_{i-1}vdt/dt+ρ_{i+1} - ρ_{i+1}vdt/dx + $\rho_i$vdt/dx
- Amount leaving me matches amount flowing into neighbour: $\rho_i^{new}+\rho_i^{new} =$ ρ_i +ρ_{i+1} - (ρ_{i+1}- $\rho_{i-1)}$vdt/dx
- If I sum over all cells, cancellation continues: $\sum\rho^{new}=\sum\rho-(\rho_{end}-\rho_{begin})$vdt/dx
- Modulo funny things at edges, stuff is conserved. This is a good thing.

Boundary Conditions

- For a finite-sized region, we have no way of solving for what happens at domain boundary.
- We need to specify this behaviour as part of the problem.
- One common case is all gradients equal zero on boundary
- Another common case is *periodic*: $\rho_{-1}=\rho_{end}$.
- What would our advection example look like with periodic boundary conditions?
- You should *always* think carefully about your boundary conditions.

Guard Cells

- The way BC's are implemented in practice is through *guard* or *ghost* cells.
- Pad your domain with extra cells. Fill them in as per BC's. Take time step. Extract original domain.
- # of guard cells may depend on details of your algorithm, but you will almost certainly need them.

In Practice

```python
#advect_finite_volume_guard.py
dt=1.0
for step in range(0,150):
    #we need one guard cell - make a 1-larger temp array
    big_rho=numpy.zeros(n+1)
    big_rho[1:]=rho
    #explicitly set the density of the guard cell
    big_rho[0]=0
    #take the difference in densities
    drho=big_rho[1:]-big_rho[0:-1]
    big_rho[1:]=big_rho[1:]-v*dt/dx*drho
    rho=big_rho[1:]
    plt.clf()
    plt.axis([0,n,0,1.1])
    plt.plot(x,rho)
    plt.draw()
```

```python
#advect_finite_volume_guard_compact.py
dt=1.0
#set up padded array outside loop
big_rho=numpy.zeros(n+1)
big_rho[1:]=rho
del rho  #we can delete the to save space
for step in range(0,150):
    #still need to explicitly set the density of the guard cell
    big_rho[0]=0
    #take the difference in densities
    drho=big_rho[1:]-big_rho[0:-1]
    big_rho[1:]=big_rho[1:]-v*dt/dx*drho

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

- Initialization is identical.
- For simple advection need one extra cell.
- Can even do in-place, saving memory, probably faster, too (see bottom)

Time Steps

- 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.

```python
#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()
```

Stability

$$\rho_j^{new}=\rho_j-(\rho_j-\rho_{j-1})vdt/dx$$

- You can learn a lot by plugging in sine waves.
- If $\rho_j=\exp(ikj)$, ρ_j^{new}=what? define a=vdt/dx
- $\rho_j^{new}=\exp(ikj)-a(\exp(ikj)-\exp(ik(j-1)) = \exp(ikj)-a(\exp(ikj)-\exp(-ik)\exp(ikj))$
- $\rho_j^{new}=\exp(ikj)*[1-a(1-\exp(-ik))]$
- If quantity in [] gets bigger than unity, solution will grow with time. Our code would be *unstable* - this is bad!

CFL Condition (*a*=vdt/dx)

- Look at 1-*a*(1-exp(-ik)). 1-exp(-ik) is bounded by (0,2)
- if 0, []=1, solution always stable.
- if 2, then []=1-2*a* can have magnitude >1 for sufficiently large *a*.
- By construction, *a* is positive, so can't get []>1. But can get []<-1: 1-2*a*<-1, 2<2*a*, or *a*>1.
- For stability, a<=1, or dt<dx/v. In words, dt has to be shorter than crossing time for cell.
- This is called the Courant–Friedrichs–Lewy (CFL) condition. vdt/dx is the Courant number.

Lagrangian

- An alternative way of solving is to label fluid packets, then follow them with time.
- Labelling usually refers to position at time $t=0$.
- Particularly simple for advection: $x^{new}=x+vdt$, or $x_j(t)=j+vt$

In Practice

```
#advect_lagrangian.py
import numpy
from matplotlib import pyplot as plt

n=300
#set up density the usual way
rho=numpy.zeros(n)
rho[n/3:(2*n/3)]=1

v=1.0
dx=1.0
x=numpy.arange(n)*dx

plt.ion()
plt.clf()
plt.axis([0,n,0,1.1])
plt.plot(x,rho)
plt.draw()

dt=1.0
#now take time steps
for step in range(0,150):
    #new particle position is just old position plus velocity
    x=x+v*dt
    plt.clf()
    plt.axis([0,1.5*n,0,1.1])
    plt.plot(x,rho,'*')
    plt.draw()
```

- Note differences in code - we just find new x position.
- Since we only follow particles that existed at beginning, we can ignore boundary conditions.

Eulerian vs. Lagrangian

- Eulerian vs. Lagrangian choice can depend on problem
- Mass conservation trivial with Lagrangian codes.
- More work to calculate density in Lagrangian code
- Lagrangian codes can have multiple velocities at same position. Unnatural with Eulerian code.
- In astrophysics, streams of dark matter can cross - Lagrangian might work better. Streams of gas can't (the wind only blows in one direction) so Eulerian might be simpler there.

Tutorial Problems

- Write a finite-volume advection solver similar to the one we saw in class. Make this one have a negative velocity, and give it periodic boundary conditions. Plot the solution as a function of time - how does it behave? How does the *total* mass behave with time? (10)

- For an Eulerian advection solver, if we increase the grid resolution by a factor of 10, how does the timestep change to maintain stability? To reach a solution at time t, how does the total amount of work scale with grid resolution dx? (10)

- Show that sufficiently large scales (small k) can still be stable even when CFL condition is violated. (5) Say I forced you to use a large dt (a>1) - derive the expression for the largest k you can use without instability kicking in (5)

- Write a particle-based advection solver. Start with a uniform density for $0<x<x_0$. Set the velocity to be equal to v_0 at $x=0$ and 0 at $x=x_0$. Plot the density as a function of time. Note that the density will be the number of particles per unit length so you will have add the particle positions into a grid. (10)

Tutorial Bonus

- We saw in class how to analytically evolve a sine wave. You can couple this with Fourier transforms to write down the solution to the Eulerian advection problem at any given time for any given dt. Write a code to do this and verify it gives the same solution as your code from problem 1). (5) You can also now analytically write down when a Fourier mode will be suppressed by half its initial amplitude. For timesteps of 0.1 and 0.5 the CFL limit, plot the 50% suppression time vs. k (5).