Classes

- Python is *object-oriented*. That means things called objects contain data and contain methods (i.e. functions) that do things with the data.

- You have seen this in action: e.g. vec=numpy.ones(10);print vec.sum()

- This is very different from e.g. C or (classic) Fortran. In C, you need to know how to sum an array. In Python, the array knows how to sum itself

- In python, objects are called *classes*. You can define them in files with the *class* keyword.

- data/methods of a class are accessed with a period '.'. The first argument to any method is the instance of the class itself. It is customary (and strongly encouraged) to name that variable "self".

Beginnings of a complex variable class

```
#class_example1.py
import numpy
class Complex:
    #__init__ is a special function.  When you create a new
    #instance of a class, if it exists in the class definition,
    #__init__ will get called.  __init__ is assumed to return the first value

    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i


if __name__=='__main__':
    num=Complex()
    print 'real part of num is ' + repr(num.r)
    print 'imaginary part of num is ' + repr(num.i)

    num2=Complex(2,5)
    print 'real part of num2 is ' + repr(num2.r)
    print 'imaginary part of num2 is ' + repr(num2.i)

    #we can assign new data to classes whenever we want.
    #you probably want to be really careful with this however
    num2.len=numpy.sqrt(num2.r**2+num2.i**2)
    print 'length of num2 is ' + repr(num2.len)


-uu-:---F1  class_example1.py    All L1     (Python)--
Loading python...done
```

Left: a bare-bones complex number class.

Below: output

```
Jonathans-MacBook-Pro:lecture5 sievers$ python class_example1.py
real part of num is 0
imaginary part of num is 0
real part of num2 is 2
imaginary part of num2 is 5
length of num2 is 5.3851648071345037
Jonathans-MacBook-Pro:lecture5 sievers$
```

Class Methods

- We’ve made a class that can hold complex numbers.
- Right now the class just holds numbers, it doesn’t do anything.
- We did take an absolute value, but we had to know at the command line how to do that.
- Let’s add a method to the class to take its absolute value

Methods ctd.

```
#class_example2.py
import numpy
class Complex:
    #__init__ is a special function.  When you create a new
    #instance of a class, if it exists in the class definition,
    #__init__ will get called.  __init__ is assumed to return the first value

    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i
    def abs(self):
        return numpy.sqrt(self.r**2+self.i**2)

if __name__=='__main__':

    num=Complex(2,5)
    print 'real part of num is ' + repr(num.r)
    print 'imaginary part of num is ' + repr(num.i)
    myabs=num.abs()
    print 'absolute value is ' + repr(myabs)
```

We have added an abs() method to the complex class. Now you can get the absolute value without having to know anything about complex numbers.

```
Jonathans-MacBook-Pro:lecture5 sievers$ python class_example2.py
real part of num is 2
imaginary part of num is 5
absolute value is 5.385164807134504
Jonathans-MacBook-Pro:lecture5 sievers$
```

What’s the difference?

```
#class_example3.py

import numpy
class Complex:
    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i
    def abs(self):
        return numpy.sqrt(self.r**2+self.i**2)
######################################################
#
# What is the difference between these two classes?
#
######################################################
class Complex2:
    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i
def abs(self):
    return numpy.sqrt(self.r**2+self.i**2)


if __name__=='__main__':

    num=Complex(2,5)
    print num.abs()
    num2=Complex2(2,5)
    print num2.abs()
```

Classes Complex and Complex2 look similar, but they might have different behaviour. Why?

```
Jonathans-MacBook-Pro:lecture5 sievers$ python class_example3.py
5.38516480713
Traceback (most recent call last):
  File "class_example3.py", line 28, in <module>
    print num2.abs()
AttributeError: Complex2 instance has no attribute 'abs'
Jonathans-MacBook-Pro:lecture5 sievers$
```

What's the difference?

```
#class_example3.py

import numpy
class Complex:
    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i
    def abs(self):
        return numpy.sqrt(self.r**2+self.i**2)
#####################################################
#
# What is the difference between these two classes?
#
#####################################################
class Complex2:
    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i
def abs(self):
    return numpy.sqrt(self.r**2+self.i**2)


if __name__=='__main__':

    num=Complex(2,5)
    print num.abs()
    num2=Complex2(2,5)
    print num2.abs()
```

Always remember your indenting! By not indenting we closed the Complex2 definition *and* defined a global function abs that replaced the built-in function.

```
>>> abs(-3)
3
>>> execfile('class_example3.py')
5.38516480713
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "class_example3.py", line 28, in <module>
    print num2.abs()
AttributeError: Complex2 instance has no attribute 'abs'
>>> abs(num2)
5.3851648071345037
>>> abs(num)
5.3851648071345037
>>> abs(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "class_example3.py", line 20, in abs
    return numpy.sqrt(self.r**2+self.i**2)
AttributeError: 'int' object has no attribute 'r'
>>>
```

Python Uses References

- Python uses *references*. If *a* is an instance of a class, and you say *b*=*a*, then the contents of *b* will point to the same memory as the contents of *a*.
- This means that if I then change *b*, *a* will also change.
- General rule is if you change/assign a piece of *b*, same piece of *a* will change.
- Be very careful - don't change values inside of functions unless you meant to.

```
>>> a=Complex(3,5)
>>> b=a
>>> print a.r
3
>>> b.r=5
>>> print a.r
5
>>>
```

Copy

- Because of this, it is often customary to have a copy() function.
- Copy should make a fully distinct version of the instance.
  NB - you might want to have a look at the copy module (import copy)

```
#class_example4.py

import numpy
class Complex:
    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i
    def copy(self):
        return Complex(self.r,self.i)
    def abs(self):
        return numpy.sqrt(self.r**2+self.i**2)


if __name__=='__main__':

    num=Complex(2,5)
    num2=num.copy()
    num2.r=10
    print 'real part of num is ' + repr(num.r)
    print 'real part of num2 is ' + repr(num2.r)
```

```
Jonathans-MacBook-Pro:lecture5 sievers$ python class_example4.py
real part of num is 2
real part of num2 is 10
Jonathans-MacBook-Pro:lecture5 sievers$
```

Overloading

- The operators in python (e.g. +,-,*…) just map to a set of special functions. You can use them on your classes if you include methods with those names.
- Extending the behaviour of the default operators is called overloading.
- __add__ is the keyword for '+'. __repr__ is the keyword for printing things.
- If you want to do this, you can google to get the rest of the special function names.
- Note that a+b is shorthand for a.__add__(b) so as written a+2 will work, but 2+a won't. Why?

```
#overload.py

import numpy
class Complex:
    def __init__(self,r=0,i=0):
        self.r=r
        self.i=i
    def copy(self):
        return Complex(self.r,self.i)
    def __add__(self,val):
        ans=self.copy()
        if isinstance(val,Complex):
            ans.r=ans.r+val.r
            ans.i=ans.i+val.i
        else:
            ans.r=ans.r+val
        return ans
    def __repr__(self):
        if (self.i<0):
            return repr(self.r)+' - '+repr(-1*self.i) +'i'
        else:
            return repr(self.r)+' + '+repr(self.i) +'i'
```

```
>>> from overload import Complex
>>> a=Complex(2,5)
>>> b=Complex(4,-3)
>>> c=a+b
>>> print c
6 + 2i
>>> d=a+b+2
>>> print d
8 + 2i
>>>
```

Try/Except

- Sometimes things go wrong. Say a method is given invalid input
- Python has *try/except*. The code will execute the *try* block. As soon as that hits an error it jumps to the *except* block.
- If there is no error, *except* is skipped.
- Optionally, you can include a *finally* clause that always gets executed after the *try/except*. Useful for e.g. freeing memory/closing files etc.

```
def __add__(self,val):
    ans=self.copy()
    if isinstance(val,Complex):
        ans.r=ans.r+val.r
        ans.i=ans.i+val.i
    else:
        try:
            ans.r=ans.r+val
        except:
            print 'Invalid type in Complex.__add__'
            ans=None
    return ans
```

```
>>> a=Complex(2,5)
>>> b=3
>>> c=a+b
>>> print c
5 + 5i
>>> b='abc'
>>> print a+b
Invalid type in Complex.__add__
None
>>>
```

Reminder: Advection

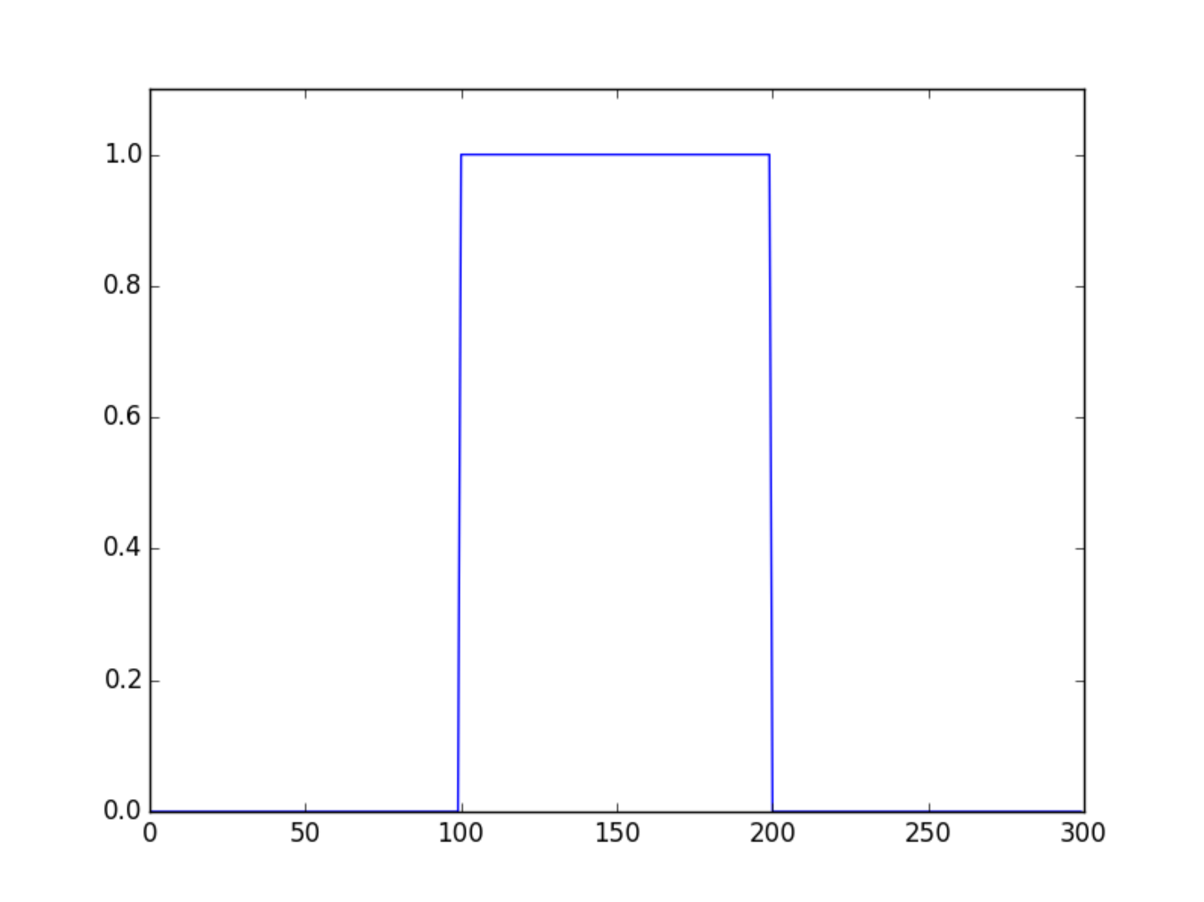
- Advection equation: $\partial f/\partial t+u\partial f/\partial x=0$
- Trial: $f=f(ut-x)$: then $uf' +u(-f')=0$. check
- So, any function of $(ut-x)$ will solve this equation.
- So, if we watch the spot in the function at x_0 when $t=0$, then at time$=t$, the position will be: $ut-x=0-x_0$, or $x=x_0+ut$. Information moves with velocity u.

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()
```

Reminder: 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.

```
#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()
```

Reminder: Stability

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

- You can learn a lot by plugging in sine waves.
- If ρ_j=exp(ikj), ρ_j^{new}=what? define a=vdt/dx
- ρ_j^{new}=exp(ikj)-a(exp(ikj)-exp(ik(j-1)) = exp(ikj)-a(exp(ikj)-exp(-ik)exp(ikj))
- ρ_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!

Reminder: 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 []=$1-2a$ can have magnitude >1 for sufficiently large a.
- By construction, a is positive, so can’t get []>1. But can get []<-1: $1-2a<-1$, $2<2a$, 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.

Numerical Viscosity/Lax

- We saw setting df/dx with $(f_{x+1}-f_{x-1})/2dx$ led to unconditional instability in advection.

- However - take (f(x,t+dt)-f(x,t))/dt for time derivative to (f(x,t+dt)-(f(x+dx,t)+f(x-dx,t))/2) leads to stability. Can you guess criterion for stability?

- Rewrite: (f(x,t+dt)-f(x,t))/dt=-v(f(x+dx,t)-f(x-dx,t))/2dx + (f(x+dx,t)-2f(x,t)+f(x-dx,t))/2dt.

- This is solving df/dt=-vdf/dx + $(dx)^2/2dt\ \nabla^2 f$. New term looks like diffusion/viscosity equations - we're adding numerical viscosity to induce stability.

Conservation Equation

- If a quantity is conserved, time rate of change in a volume is equal to net flow into/out volume.
- If conserved quantity is ρ and velocity is u then flow out of region is $\rho_{+}u_{+}$ and flow in is $\rho_{-}u_{-}$. Net flux is then $-\partial(\rho u)/\partial x$.
- Equation then become $\partial\rho/\partial t = -\partial(\rho u)/\partial x$, or $\partial\rho/\partial t+\partial(\rho u)/\partial x=0$
- If a quantity is created, then we pick up extra term for rate of creation:
- now $\partial\rho/\partial t=-\partial\rho/\partial x + q$, where q is the creation rate.

Euler Equations

- Now we’re set to derive equations of fluid mechanics.
- The full fluid equations (Navier-Stokes) include forces from viscosity
- We will make approximation that viscosity is negligible
- Further, we will assume no energy flows between pieces of fluid (this is usually quite a good approximation)
- Leaves us with Euler equations. What equations should we have?

Mass Conservation

- Generally, no matter is created/destroyed, so mass is strictly conserved.
- Mass conservation becomes $\partial \rho / \partial t + \partial (u\rho) \partial x = 0$
- Note that if you had source/sink of matter, it would appear as an extra term

Momentum

- Momentum is ρu. So conservation equation is $\partial(\rho u)/\partial t+\partial(\rho u^2)/\partial x=0$
- Velocity appears squared, so equation is nonlinear
- Fluid pressure will exert a force, so force term must be added.
- Force on right side of a packet is $-P_+$, force on left is $+P_-$, so total net force is difference, limit is $-\partial P/\partial x$. This force has to go into momentum equation.
- Momentum equation: $\partial(\rho u)/\partial t+\partial(\rho u^2)/\partial x=-\partial P/\partial x$
- Conservation form: rewrite as $p=\rho u$, get $\partial p/\partial t+\partial(pu+P)/\partial x=0$

Energy

- Two pieces of energy - internal thermal energy and bulk kinetic.
- Call total energy (thermal+kinetic) per unit mass E.
- Energy creation rate from pressure is power, or force * velocity
- Gives $\partial(\rho E)/\partial t+\partial(u\rho E)/\partial x=-\partial(uP)/\partial x$
- Rewrite into conservation form: $\partial(\rho E)/\partial t+\partial(u\rho E+uP)/\partial x=0$

Euler So Far

- $\partial\rho/\partial t+\partial(u\rho)\partial x=0 \quad \partial p/\partial t+\partial(pu+P)/\partial x=0 \quad \partial(\rho E)/\partial t+\partial(u\rho E+uP)/\partial x=0$
- Three equations, how many unknowns? Solution needs velocity, density, energy, and pressure.
- So, need one more equation. Normally done by specifying a relation between pressure and energy. This is called an *equation of state*.
- Classic EoS is gamma law, $P\sim\rho^{\gamma}$. For ideal gas, $e=3/2\ nkT$, pressure is nKT, so $P=2/3\rho e$ (where $e=E-1/2\rho u^2$ is the thermal energy).

Derivation of γ

- Let's compress a volume of gas and see how energy changes.
- $dE=-PdV$. $E=aPV$ (where $a=3/2$ for ideal gas)
- $ad(PV)=-PdV$. $aVdP+aPdV=-PdV$
- $dP(aV)=-dV(P(1+a))$, $adP/P=-(1+a)dV/V$.
- $\log(P)\sim-(1+a)/a\log(V)$. $P\sim V^{-(1+a)/a}$. Density $\sim 1/V$, so $P\sim\rho^{1+1/a}$. The index is usually called γ (gamma). For ideal gas, a is 3/2, so $\gamma=1+2/3=5/3$.

Euler Equations with EoS

- We can now write down Euler equations in conservation form with EoS
- $E=1/2u^2+e$, $\rho e=P/(\gamma-1)$. So $P=\rho(\gamma-1)(E-1/2u^2)$
- $\partial Q/\partial t+\partial(f(Q))/\partial x=0$
- $Q=[\rho,\rho u,\rho E]$, $f(Q)=[\rho u, \rho u^2+P, \rho uE+uP]$
- using momentum $p=\rho u$: $Q=[\rho,p,\rho E]$, $f(Q)=[p,pu+P,pE+uP]$

PDE Systems, Ctd:

- System uv=Cv is just eigenvalue problem. Get a solution for each eigenvector/eigenvalue pair, where propagation speed is eigenvalue.
- When eigenvalues are real, system is called *hyperbolic*, solutions of form h(x-ut). Information propagates at finite speed.
- When eigenvalues are imaginary, system is *elliptical,* solutions of form h(x-iut). You might expect treatment in numerical solvers to be different.
- Do you think fluid equations should be elliptical or hyperbolic?

CFL Condition Revisited

- Euler equations give us a system of 3 coupled equations.
- This means 3 eigenvalues. For CFL condition, want time step stable for largest velocity eigenvalue.
- What do you think the three eigenvalues are? You should be able to guess from physical intuition. (recall that the speed of sound $c_s^2=\gamma P/\rho$)

Aside: Stiff Equations

- We get one eigenvalue for fluid velocity, and 2 for velocity ±speed of sound.

- If $c_s >> u$, then CFL means timestep has to be tiny compared to natural one from fluid velocity. When eigenvalues diverge like this, equations are called *stiff*. Different computational techniques required.

- Incompressible fluid mechanics - limit where $c_s >> u$. Fluid has time to move out of the way. Otherwise it would compress.

- Techniques to solve stiff equations are different. If you hit a stiff set, look them up. Always check if your system is stiff!

Structure of a Simple1-D Fluid Code

- First, do boundary conditions. For 1-D fluid, we might want smooth conditions - that gradients go to zero on boundary (period often unnatural).
- If we use density, momentum, total energy as variables (the conservation quantities) then need to calculate velocity
- Now need to calculate pressure
- Next calculate gradients - we use upwind 1st order scheme, where I flow with my velocity
- Calculate CFL timestep
- Finally, update density, momentum, Energy

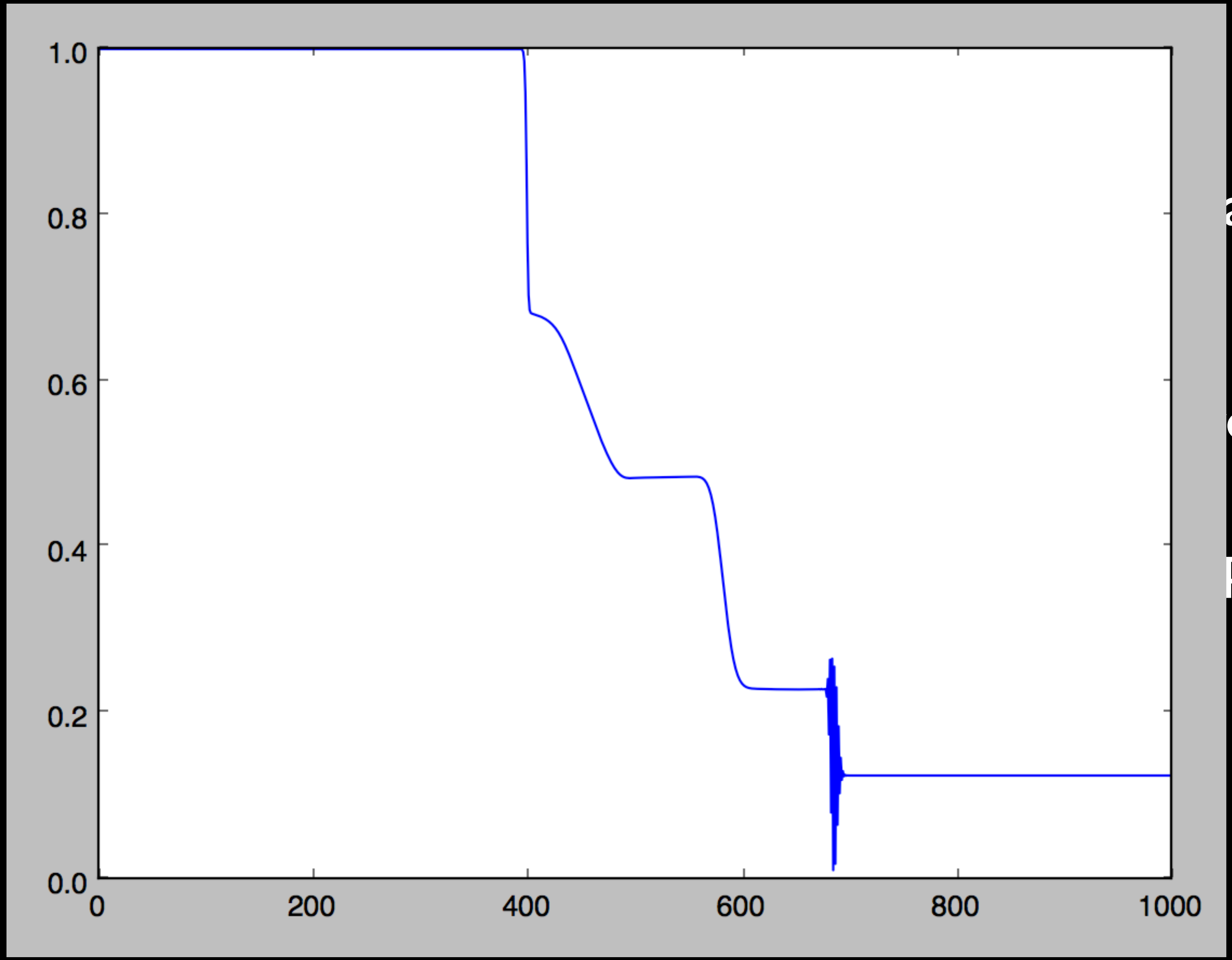
Primitive Equations

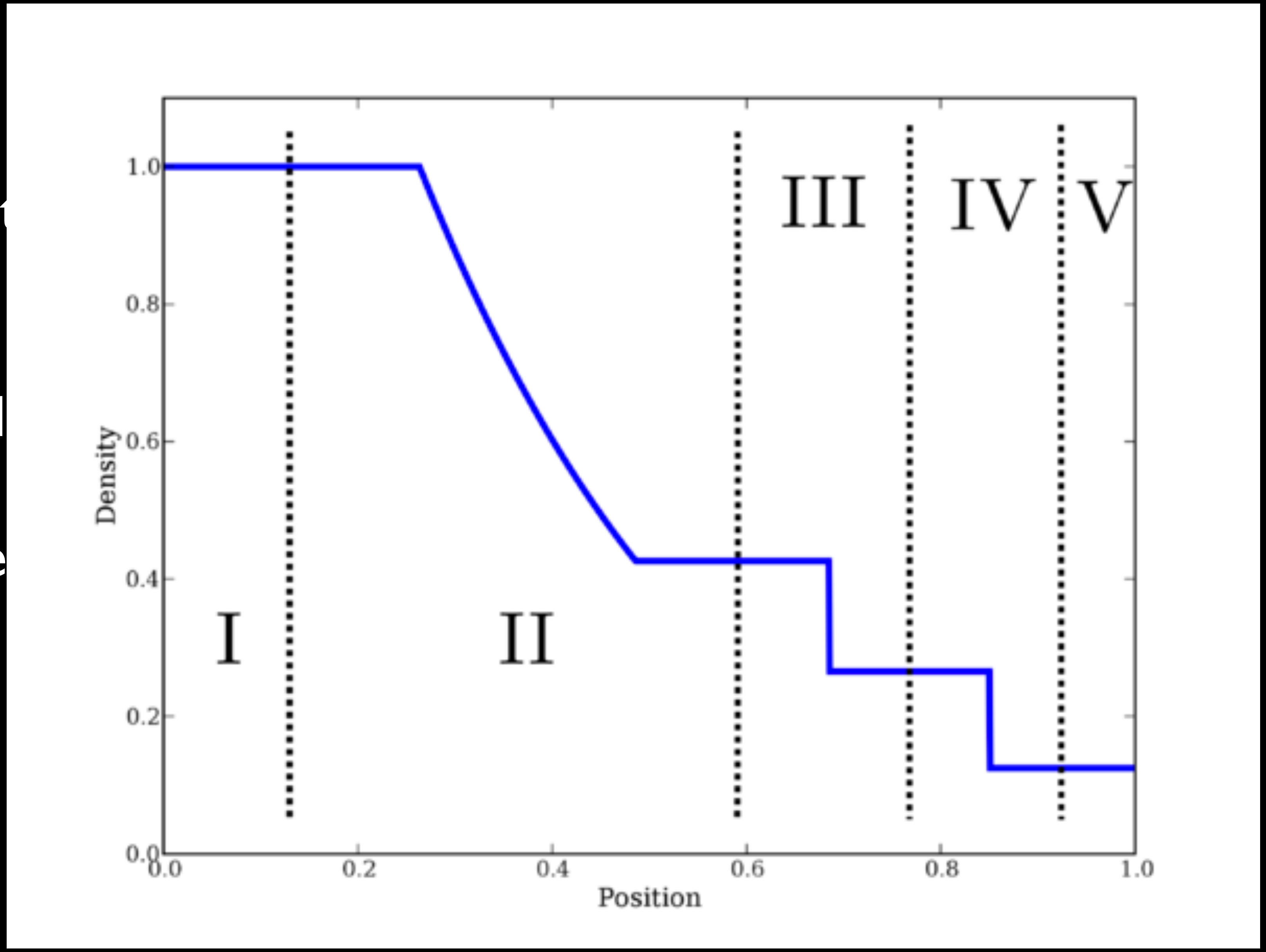
- Conservation has derivatives of joint quantities. Hard to work with numerically, so expand derivatives.
- Assuming $P \propto \rho^{\gamma}$, we can rewrite the Euler equations in *primitive* form. After math, we get:
- $\rho_t + u\rho_x + \rho u_x = 0$ where e.g. $\rho_t = \partial\rho/\partial t$
- $u_t + uu_x + 1/\rho \, P_x = 0$
- $P_t + uP_x + \gamma P u_x = 0$

Shock Tube

- Classic testing problem is a shock tube: start with a density/pressure jump in the middle, with velocity=0.
- What should this look like? let's run hydro1d.py
- What answer *should* look like from wikipedia:

Shock Tube


1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000


I
II
III
IV
V
Density
Position
0.0
0.2
0.4
0.6
0.8
1.0

Riemann problem/Godunov Solver

- If we're facing solving $u_t + Au_x = 0$, we rotate into the eigenspace of A. This gives us uncoupled equations that look like advection (when looking at short enough time).
- Finite volume can be mapped into Riemann problem - you have a discontinuity between cells. Know how to propagate eigenmodes
- Godunov solvers do this - evolve solution by solving Riemann problem.
- First order accurate, but can be built into more accurate solution.