Code Documentation

Daniel W. Zaide

February 11, 2015

1 Overview

This code was developed to model facade envelopes with several main objectives:

- Versatility - The code should be versatile, flexible, and handle a wide range of possible physics
- Simplicity - The code should be simple, and defining new components and physics should be easy and clean
- Readable - Outside of the infrastructure, the physics components and problem setup should be readable

With that in mind, we leverage python using `list` and `dict`, rather than numerical arrays to allow for flexibility. We also use function handles to allow for easy definition of functions and allow for runtime adding variables classes.

2 Infrastructure

The goal is ultimately to solve the global system

$$\mathbf{R}(\mathbf{U}) = \sum_F \mathbf{F}(\mathbf{U}) + \sum_S \mathbf{S}(\mathbf{U}) = \mathbf{0} \tag{1}$$

for a set of states $\mathbf{U}$, flux functions $\mathbf{F}$, and sources $\mathbf{S}$. Rather than a typical control volume approach, a simpler abstraction is used: `blocks`. Each block has its own state variables, and the equations for these state variables are defined by fluxes (information from neighboring blocks) and sources (external information), which are defined within `flux.py` and `source.py`. Boundary conditions are done implicitly with a ghost cell type approach, defining blocks with states that remain constant. Wrapping the whole thing together is a `problem` object, which is initialized with the blocks we are solving for. The boundary blocks are created, but left explicitly outside the problem object.

2.1 Blocks

Each typical control volume can be considered as a block. Blocks are connected to other blocks through fluxes, which act as boundary conditions. Blocks can have arbitrary state variables, and not all blocks have to have the same set of states, provided flux functions are defined that connect them together. Every block has a set of states, U, stored in the `.state`. Consider a block with temperature, density, and velocity defined. The state for example, may look like

```
>>> block.state
OrderedDict([('T',20), ('rho',1.05), ('u',0.01)])
```

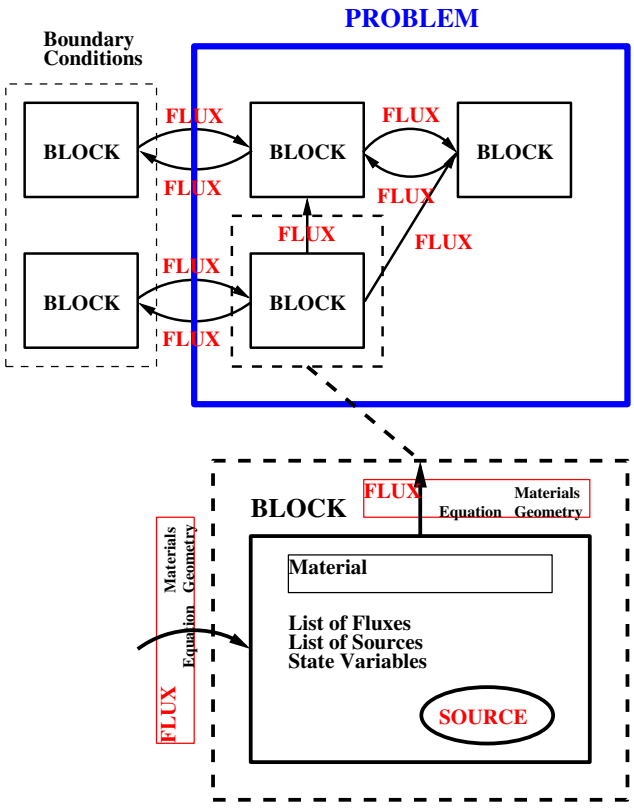

Figure 1: Overview of basic framework defining a problem.

where the ordered dictionary is used to preserve the mapping used in construction of the global problem. We can access these variables using the dict, such as `block.state['T']`. Every block defines its own equation

$$.R(U) = \sum_F F(U) + \sum_S S(U) = 0 \tag{2}$$

for block B and states $U \in B$. In this framework the `OrderedDict` structure is used to keep track of state variables. Each block can have its own set of states, provided there is an equation (flux or source) to be solved (R cannot be empty). The form of this will be a dictionary, and the sum of the fluxes and sources will be a sum over similar dictionaries, using some pythonization.

2.2 Fluxes

Fluxes are defined through blocks which remain constant, such as external temperatures or inflow conditions. Each flux is a function of two blocks. Boundary conditions are currently defined by fluxes with one block not part of the problem. The current definition is a bit strange, and its not clear where the flux should be currently implemented. Each flux function is defined in `flux.py`, and is initialized by passing in the flux function name. Fluxes are one directional (from one block to another) and are assigned to a block.

2.3 Sources

Sources are defined similar to fluxes, but are functions of a block. For now, only constant sources are defined.

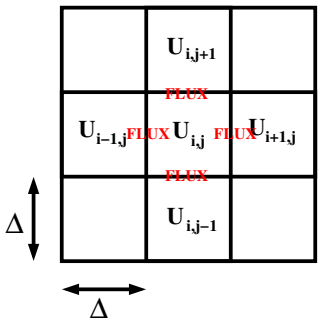

Figure 2: Set up for Poisson's equation

2.4 Problem

This is a class that serves as a wrapper for everything, converting from the local block regions into a global matrix and back to do the solve. The problem is initialized with a list of blocks.

```
def __init__(self,blocks):
                self.b = blocks
                self.mapping = [(i, k) for i, b in enumerate(blocks) for k in b.state.keys()]
```

where the mapping between blocks into a global list is done at initialization, and stored for reuse.

```
def solve(self):
                solution = [None]*len(self.mapping)
                for ix, (i,k) in enumerate(self.mapping):
                        solution[ix] = self.b[i].state[k]
                solution = fsolve(self.r, solution)
                self.update(solution)
```

This class is the one that needs the least work, the only places that should be modified are in `solution = fsolve(self.r, solution)`, where `fsolve` is a built-in function from `scipy.optimize`.

3 Materials

4 Examples

4.1 Poisson's Equation

This is solved in the file `poisson2D.py`. Consider the decoupled pair of Poisson's equations, $\nabla^2 \mathbf{U} = \mathbf{S}$, $\mathbf{U} = [u, v]^T$

$$\frac{\partial^2}{\partial x^2}\begin{bmatrix} u \\ v \end{bmatrix} + \frac{\partial^2}{\partial y^2}\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} (x^2+y^2)e^{xy} \\ 4(x^2+y^2+1)e^{x^2+y^2} \end{bmatrix} \tag{3}$$

which has the exact solution $u = e^{xy}, v = e^{x^2+y^2}$. We will solve it using a finite volume method on a uniform grid of $[-1, 1] \times [-1, 1]$ with $\Delta x = \Delta y = \Delta$, with variables stored on the center of the grid. For a grid with N interior cells, we have the cell center as $(x, y)_{i,j} = (i\Delta - \Delta/2 - 1, j\Delta - \Delta/2 - 1)$ The discretization is[1]

$$\frac{1}{\Delta^2}\sum_{neighbors}(\mathbf{U}_{neighbor}-\mathbf{U}_{i,j})-\mathbf{S}_{i,j} \tag{4}$$

The flux can be written as a difference for a given block,

$$\mathbf{F}(\mathbf{U},\mathbf{U}_N)=\frac{1}{\Delta^2}(\mathbf{U}_N-\mathbf{U}) \tag{5}$$

The implementation is as follows. First, import the modules as

```
import src.blocks as b
import src.flux as f
import src.problem as p
import src.source as s
```

First, define an extra layer of cells outside of the domain, and initialize to the exact solution. This is done at the start,

```
d = 2./float(N)
B = [b.Block('('+str(i*d-d/2-1)+','+str(j*d-d/2-1)+')',None,{'u':math.exp((i*d-d/2-1)*(j*d-d/2-1)),
        'v':math.exp((i*d-d/2-1)**2+(j*d-d/2-1)**2)}) for i in range(0,N+2) for j in range(0,N+2)]
```

using a for loop, wher we have each cell named after its coordinates,

```
'('+str(i*d-d/2-1)+','+str(j*d-d/2-1)+')'
```

we do not define a material (theres no need), so we have `None`. Finally, we initialize with a dictionary evaluated to the exact solution,

```
{'u':math.exp((i*d-d/2-1)*(j*d-d/2-1)),'v':math.exp((i*d-d/2-1)**2+(j*d-d/2-1)**2)}}
```

With the blocks initialized, the source functions can be initialized. In this case, the coordinates of the block are lazily stored in the block name. For further development of the code, coordinates may be needed, but in this case, the name is a sufficient place to hide them. Here for each block, we add a constant source, evaluated at the center. Each block has its own source added.

```
for block in B:
        (x,y) = eval(block.name)
        block.addSource(s.Source('const',u = -(x*x+y*y)*math.exp(x*y),v = -4.0*(x*x+y*y+1.0)*math.exp(x*x+y*y)))
```

The source, defined in `source.py`, is

```
def const(self,b):
        return dict([(state,self.kwargs[state]) for state in b.state])
```

which takes the option key-value pairs passed into `kwargs` as constant, and returns a dictionary with the constant corresponding to each state.

Next, the fluxes for each block can be added. Define geometry consisting of Δ, with no materials inside the edge

```
G = {'type':'edge','d':d*d,'m':[]}
```

Now we can initialize the fluxes. We define a flux function called `difference`, in `flux.py` as

[1] Technically this is not correct, as the source should be averaged over a cell, not the point value evaluated at the cell center. As the cell average is second-order accurate, there is no change in the overall accuracy of the scheme, not to mention its not really the point of this exercise.

```
def difference(self,b):
        return dict((s,(self.N.state[s]—b.state[s])/self.G['d']) for s in b.state)
```

This function returns a dictionary, by summing over each state, subtracting, and doing the difference. If this were a real CFD code, for each edge, we would only define a flux once. In this example, each flux is defined twice, once for each direction. We create a list of interior blocks which we will actually solve on, ignoring the boundary blocks. For each neighboring block, we create a flux based on that block, connecting the two blocks. We then initialize the interior block's solution to zero.

```
n = N+2
for i in range(1,n—1):
        for j in range(1,n—1):
                for k in [(i—1)*n+j, i*n+j—1,(i+1)*n+j, i*n+j+1]:
                        B[i*n+j].addFlux(f.Flux(B[k],'difference',G))
                B[i*n+j].state['u'] = 0.
                B[i*n+j].state['v'] = 0.
interiorBlocks = [B[i*n+j] for i in range(1,n—1) for j in range(1,n—1)]
```

Finally, with the blocks created, the problem is easily created and solved with

```
P = p.Problem(interiorBlocks)
P.solve()
```

The rest of the file does error analysis, and computes the accuracy of the scheme.

5 Concerns

1. Blocks have no geometry associated with them.
2. Runtime speed is not considered
3. Conditioning of the system is not considered, with respect to block ordering in the global system.
4. Fluxes are computed twice, once for each block.
5. Solving is done without explicitly computing the $\frac{\partial \mathbf{R}}{\partial \mathbf{U}}$.