Dynamic Modeling, Week 4

Greg Baker

December 3, 2003

All notes and code available at:

`http://www.cs.sfu.ca/~ggbaker/reference/modeling/`

1 Debugging and Testing

- You need to be sure your program is working correctly.
- It's impossible to check a whole program at once. You have to test piece-by-piece.
- You can test functions by importing your program into the Python interpreter:

```
>>> from dynamic import * (suck in all of the functions, etc from dynamic.py)
>>> f = zeros((xmax+1,tmax+1), Float) (create the arrays)
>>> dec = zeros((xmax+1,tmax+1), Int)
>>> print f[:,tmax] (output all values of x with t=tmax)
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  ...]
>>> initialize(f) (run initialize so we can see what it does)
>>> print f[:,tmax] (check f after)
[ 0. 7.05882353 12.63157895 17.14285714 20.86956522  ... ]
>>> update(f, dec, tmax-1) (fill in the tmax-1 column)
>>> print f[:,tmax-1] (check f after)
[ 0. 9.91304348 14.75294118 18.66666667 21.9014778  ... ]
>>> print dec[:,tmax-1] (check dec after)
[ 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 2 2 2 2 2 2 2  ... ]
>>> listmax( [17.08, 19.10, 15.85] ) (test listmax)
(1, 19.10)
>>> chop(13, 0, 10) (test chop)
10
```

- If you're suspicious of particular values (possibly because they're causing an error), print them out and make sure they're right. Add a line like:

```
print t,x,h
```

This will let you check the values (before the error) and see what's going on.

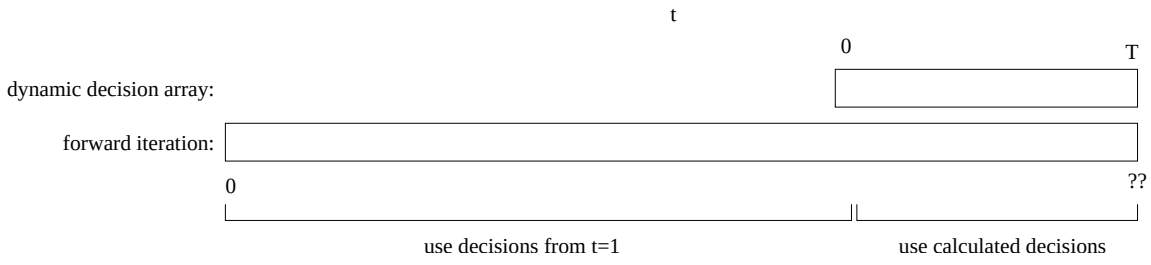
- As you make changes, try running your program and make sure the output changes appropriately. If not, you probably didn't make the change you thought.

2 Getting to your data

- Both a backward and forward iteration generate a lot of data—more than you could look through manually.
- It's easy to write programs to go through the fitness and decision arrays and do a particular operation—just write some `for` loops to skim through the arrays.
- examples:
  - output fitness values for a particular state for all times
  - list all conditions where the decision is to return to the haulout
  - check to see at what time the decisions change
- Any output can be formated for a spreadsheet/stats program and imported for further analysis or graphing.

3 Stationarity

- Running the forward iteration (Monte Carlo) is much faster than the backward (dynamic program).
- Do the individuals always behave the same way if they're far enough from the end of time?
- If so, there's no point in calculating the decisions for long periods of time—they're all the same anyway. The model is "stationary."
- We can use stationarity to do long forward simulations.
- For times before the start of the dynamic arrays, just use the decisions from $t = 1$:



- You need to first convince yourself that the model is stationary—use a checking program like the ones described above.

- Then, modify the Monte Carlo program to look at the right place in the decision array.
  - Maybe, define a function like:

```
def array_time(t):
    if t < monte_steps - tmax + stationary_point:
        # We're back before the stationary point.
        # Use the stationary point.
        return stationary_point
    else:
        # We have a real calculated decision value.
        # Use it.
        return tmax-(monte_steps-t)
```

  - Then, references to the decision array become:

```
dec[x,y,h,array_time(t)]
```

  - ...and set `monte_steps` to whatever you like.

4 How many state values?

- The total calculation time and memory for your dynamic program will be proportional to the number of entries in the array.
- This will be the product of the array's indices, eg. $x_{max} \times y_{max} \times h_{max} \times T$.
- Decreasing the number of possible values for any one state will proportionally decrease calculation time and memory.
- So, how few is too few?
  - Less state values means more linear interpolation
  - ...and more chance in the Monte Carlo simulation.
  - Linear interpolation won't be a problem if: (i) you keep enough states to capture the features of the fitness landscape and (ii) the fitness landscape is generally smooth.
  - In the Monte Carlo, the values being probabilistically rounded will be smaller, so there would be more variation in the times taken to lose/gain value. But, the *expected* behaviour should be the same. You may have to run more simulations to get good averages.
- How many is that again?
  - At least 10. Probably less than 100.
  - Try it and see if less states changes the behaviour (holding everything else constant).

Slice of Arrays

```
# Output fitness and decision values for a particular
# x,y,h through the whole time window.

# parameters in param.py
from dynamic import *

filename="timeslice%i.txt" % (xmax)

print "Loading data..."
f = fromfile(fitfile, Float, (xmax+1,ymax+1,hmax,tmax+1))
dec = fromfile(decfile, Int8, (xmax+1,ymax+1,hmax,tmax+1))

data = open(filename, 'w')
x=5
y=10
h=1

data.write("x=%i\ny=%i\nh=%i\n\n" % (x,y,h))

data.write("t\tdecision\tfitness\n")

print "Writing output file..."
for t in range(tmax+1):
    if t%10==0:
        data.write("%i\t%i\t%g\n" % (t,dec[x,y,h,t],f[x,y,h,t]) )

data.close()
```

Check for Stationarity

```
# Check to see when decisions destabilize--if it's late, our model
# looks stationary.

# parameters in param.py
from param import *

print "Loading data..."
#f = fromfile(fitfile, Float, (xmax+1,ymax+1,hmax,tmax+1))
dec = fromfile(decfile, Int8, (xmax+1,ymax+1,hmax,tmax+1))

# the starting time for the test
t0 = 1

print "Checking for differences..."
found=0
for t in xrange(t0,tmax):
    if t%100==0:
        print t
    for x in range(xmax+1):
        for y in range(ymax+1):
            for h in range(hmax):
                if dec[x,y,h,t0] != dec[x,y,h,t]:
                    print ("Difference at x=%i, y=%i, h=%i, t=%i: "
                           + " init=%i, now=%i" )\
                           % (x,y,h,t,dec[x,y,h,t0],dec[x,y,h,t])
                    found=1

    if found:
        print "All decisions identical for t=%i to %i." % (t0,t-1)
        break;

raw_input("Press return to exit");
```