# Dynamic Modeling, Week 2

Greg Baker

November 19, 2003

All notes and code available at:

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

# 1 Patch Selection Dynamic Program

## 1.1 Main program

- This is a revised version of patch selection that moves the parameters and common definitions into a separate file. It will be shared by the dynamic program and Monte Carlo program. This way, there are no duplicate numbers to keep in sync.
- Starting with "`from param import *`" runs the code in `param.py`. Now we have the parameter definitions from that file.
- `param.py` defines some variables and a function
  - "`from numarray import *`" gets everything from the `numarray` module.
  - `numarray` is needed to create the big arrays that we need (`f`, `dec`).
- In `dynamic.py`, the `main()` function is now the starting point (more on that later)
- The `zeros` function comes from `numarray`. It creates an array with the given dimensions, filled with 0s.
- Types for numarray arrays:
  - `Int` for integers: -10, 2291, 0, 4294967295
  - `Float` for real numbers: 36.923076, -229.00219, 2.34e12 ($= 2.34 \times 10^{12}$)
- The rest of `main()` is pretty straightforward. It calls some other functions to do the backwards iteration.
- The last two lines dump the two big arrays to files, so they can be used in other programs. This lets us do a bunch of analysis with only one dynamic run.

## 1.2 update(f, dec, t)

- Fill in all values in `f` and `dec` for time `t`.
- `x` = 0 is easy—fitness is 0.0, there is no decision (it's already 0, so we might as well leave it)
- For every other `x` value, check all possible decisions. Select the one with max fitness.
- `choices` is a Python "list". The first line in the `for x` loop sets it to an empty list.
- The `for d` loop calculates the fitness for each possible decision.
- `choices.append()` adds another item to the end of the list.
- After the `for d` loop, `choices` contains the fitness values for each of the three decisions.
  - eg. `[17.08, 19.10, 15.85]`
  - the fitness value for decision 0 (first foraging patch) is 17.08.
- The `listmax` function returns the position of the largest value in the list, and the value itself.

  ```
  >>> listmax( [17.08, 19.10, 15.85] )
  (1, 19.10)
  ```

  In this case, we make decision 1 (foraging patch 2) and have fitness 19.10.
- Note that the decisions are 0, 1, 2 (not 1, 2, 3). This is easier with Python's zero-based arrays.
- These values are entered into the `dec` and `f` arrays, so we can use them later.

## 1.3 V(d, x, t, f)

- Calculate reproductive success, assuming decision `d` from energy `x` at time `d`.
- The `if/elif/else` block in Python selects one of the chunks of code to execute.
  - Conditions are checked in order. When the *first* true one is found, execute that code.
  - If none are true, execute the `else` code.
- Decisions 0 and 1 (`d<2`) are foraging patches. Calculate fitness from formulas in the model.
- Decision 2 is the reproduction patch. Fitness it reproduction plus new energy state.
- Restrictions on reproduction: can't reproduce if $x < x_{rep}$. Can have at most $c_3$ offspring.
- Each offspring costs one energy unit. Can't go below $x = x_{rep}$.

# 2 Seal Foraging Dynamic Program

Mostly the same as the patch selection model. Some key differences:

- Linear interpolation needed
  - Handled by the `interpolate_xy` function
  - Instead of using `f[new_x, new_y, new_h, new_t]` elsewhere, call `interpolate_xy (f, new_x, new_y, new_h, new_t)` to take care of any fractional $x$ and $y$ values and return the correct fitness value.
  - Uses the formula from Clark-Mangel.
  - Also takes care of bounds checking—won't let $x$ go past 0 or $x_{max}$.
- More complicated fitness function makes `V` more complicated.
  - Every fitness value is multiplied by the probability of survival for that habitat/decision. These are pre-calculated in `param.py`.
  - Some decisions take more than one time unit, so we have to make sure there is time before $T$ to complete the decision.
  - Some decisions are impossible. Return $-1$ fitness if it is—that decision won't be taken.

# 3 Forward Iteration

- After the backwards iteration, we have an array, `dec`, full of decisions. Let's use that.
  1. Choose some starting values for the state variables and set $t = 0$.
  2. Have a look at the decision for that $t$ and state variables. Output the decision.
  3. Determine how the state variables will change after that decision.
  4. Move the individual to that position in the matrix—change $t$ and state variables as appropriate.
  5. Repeat until $t = T$ or it dies.
- You can then look at the decisions and overall behaviour. Is it realistic?
- If not, you can (a) accept it or (b) change your model.
- Some state transitions may depend on chance—it might find food, get eaten, etc.
  - If so, you'll have to randomly decide whether the chance comes up or not.
  - So, not every forward iteration will be the same. Run several to see what happens on average.
- If you are doing linear interpolation, it comes up here too.

  - eg. If we get the new value $x = 2.9$, what do we do?
  - Probabilistically round to an integer. Eg. round to three 90% of the time and to two 10% of the time.

# 4 Patch Selection Monte Carlo

## 4.1 Main Program

- Sends output to a file that can be read into a spreadsheet/stats program. (How that's done is explained below.)
- Does a fixed number of simulations, `runs`.
- At each simulation step (decision by the individual), get the decision from `dec` and determine what happens next.
- Move to the next state and time and count any reproduction before we move on.
- If we're eaten or starved, the simulation is over.

## 4.2 `next_state(x, t, d)`

- Take a given energy, time and decision. Figure out what happens next.
- Mirrors the calculations done in `V` in the dynamic program.
- For anything that *might* happen, use the `rv` function to generate the randomness. Return the state for the possibility that came true.
- Returns $(x, t, r, e)$: $x$ is the energy state, $t$ is the time, $r$ is the reproduction in this time step, $e$ is one if we were eaten.

# 5 Seal Foraging Monte Carlo

Mostly the same as the patch selection model. Some key differences:

- Since linear interpolation was used in the dynamic program, we need to deal with it here as well.
  - The `prob_round` function does the probabilistic rounding. The same as `rv`, but with an integer part.
- Different file output:
  - Also outputs a human-readable file.
  - The `state` function coverts the terminal condition code to a word for easy reading.

  - Formatting a line for the spreadsheet file is handed off to the `ss_line` function. It used in two places; this keeps them the same.

- Only lines with changed states are outputted
  - Nothing printed unless one of $x, y, h, d$ change.
  - Too much output otherwise.
  - The variable `last` holds the list `(x,y,h,d)`. If that changes, we need to do the output.

- The possibility of getting eaten is handled in the main program, not `next_state`. (No reason, just a style difference.)

- No reproduction is possible during the simulation, so we don't have to keep track of it.

# 6 Formatting Output

- Create data for a spreadsheet/stats program is easy, as long as its formatted correctly.

- To output to a file:
  1. Open the file for writing: `data = open(montefile, 'w')`
  2. Write to the `data` file handle: `data.write("to the file")`
  3. Write as much as you need.
  4. Close the file: `data.close()`

- `data.write` doesn't automatically start a new line with each statement (`print` does). To go to a new line, type `\n`.

- Data that you want to import into a spreadsheet should be formatted in tab-separated columns. To print a tab, type `\t`.

- eg. output a row with values A, 32.2, 10 in the first three columns:

```
data.write("A\t32.2\t10\n")
```

- The best way to output variable values is the Python formatting operator, `%`. The `%` operator is a way to create formatted output from a template:

```
"template string" % (replacements)
```

- Any percent signs in the template string will be replaces with the values after the `%`.

```
>>> print "((%i))[[%g]]" % (10, 12.345)
((10))[[12.345]]
>>> a=10-2
>>> print "%i\t%i\t%i\n" % (a, a/2, 15)
8       4       15
```

- For an integer, use `%i` and `%g` for a float. The replacements are taken in order.

# 7 Testing

- You can put the main code for your program in a function called `main` and put this at the bottom of the file:

```
if __name__ == '__main__':
    main()
```

- This lets you `import` your code for testing or run it by itself.
- eg. in the Python interpreter (with the patch selection model):

```
>>> 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
```

  This lets us check various parts of the program by hand. Without the `__main__` trick, the `import` would have made the whole program run—not what we wanted.

- Also, adding a few `print` statements to output suspicious values at key times can be very useful.