Patch Selection Dynamic Program

```
from param import *

######################################################################
# Utility Functions

def listmax(lst):
    """
    Find the largest element in the list lst.  Return a pair containing its
    position and value.
    """
    maxval=lst[0]
    maxpos=0
    for i in range(len(lst)):
        if lst[i]>maxval:
            maxpos=i
            maxval=lst[i]
    return (maxpos,maxval)

######################################################################
# Model functions

def initialize(f):
    """
    Initialize the last column (t=tmax) of f.
    """
    for x in range(xmax+1):
        f[x,tmax] = acap*x/(x + x0)

def V(d,x,t,f):
    """
    Calculate the reproductive success for behaviour d at energy x, time t
    in the fitness array f.
    """
    if d<2:
        # forages
        x1 = chop( x+y[d]-cost, 0, xmax )
        x2 = chop( x-cost, 0, xmax )
        return (1.0 - m[d]) * (p[d] * f[x1,t+1] + (1.0 - p[d]) * f[x2,t+1])
    else:
        # tries to reproduce
        if x < xrep: # can't reproduce
            reproduction = 0
```

```
            x1 = chop(x-cost,0,xmax)
        elif x < xrep+c3: # limited reproduction
            reproduction = x - xrep
            x1 = chop(xrep-cost,0,xmax)
        else: # full reproduction
            reproduction = c3
            x1 = chop(x-cost-c3, 0, xmax)
        return reproduction + (1.0 - m[d]) * f[x1,t+1]

def update(f, dec, t):
    """
    Calculate the values for time t in the arrays f and dec.
    """
    f[0,t] = 0.0
    for x in range(1,xmax+1):
        choices = []
        for d in range(3):
            choices.append( V(d,x,t,f) )

        (dec[x,t], f[x,t]) = listmax(choices)

######################################################################
# Main program

def main():
    # Create and initialize the main data arrays:
    #   f[x,t]: reproductive fitness for energy x at time t
    #   dec[x,t]: the decision made if at energy x, time t
    f = zeros((xmax+1,tmax+1), Float)
    dec = zeros((xmax+1,tmax+1), Int)

    initialize(f)

    # calculate the dynamic arrays and output to a file:
    for t in range(tmax-1,0,-1):
        update(f, dec, t)

    f.tofile(fitfile)
    dec.tofile(decfile)

# don't run main() if we're being imported
if __name__ == '__main__':
    main()
```

Patch Selection Parameters

```
#####################################################################
# Basic model parameters

fitfile="fitness.data"
decfile="decision.data"
montefile="montedata.txt"

xmax = 30
xrep = 4
tmax = 20

m = [.01, .05, .02]  # predation probabilities
p = [.2, .5, 0]      # discovery probabilities
y = [2, 4, 0]        # food sizes
cost = 1             # per period metabolic cost
c3 = 4
acap = 60.0
x0 = 0.25*xmax
xint = 2


# do this stuff here since we need it in several places anyway...

def chop(x, min, max):
    """
    Return x, pushed into the range [min...max]
    """
    if min<=x and x<=max:
        return x
    elif x<min:
        return min
    else:
        return max

# import the numeric array module
from numarray import *
```

Patch Selection Monte Carlo

```
from param import *
from random import random

runs=10

def rv(p):
    """
    A uniform random variable with probability p.

    Return true with probability p
    """
    if random()<p:
        return 1
    else:
        return 0

def next_state(x,t,d):
    """
    Determine what happens during this time unit.

    Returns (x,t,r,e) where:
      x: next energy state
      t: next time unit
      r: number of offspring
      e: 1 if it was eaten, 0 otherwise
    """
    # always move one time step
    next_t = t+1
    # hope we're not eaten
    e = 0

    # look at possible decisions:
    if d<2:
        # forage
        r = 0
        if rv(m[d]):
            # eaten
            next_x = chop( x-cost, 0, xmax )
            e=1
        else:
            # not eaten
            if rv(p[d]):
```

```
                # found food
                next_x = chop( x+y[d]-cost, 0, xmax )
            else:
                # no food found
                next_x = chop( x-cost, 0, xmax )

    else:
        # try to reproduce (no predation here)
        if x < xrep: # can't reproduce
            r = 0
            next_x = chop(x-cost,0,xmax)
        elif x < xrep+c3: # limited reproduction
            r = x - xrep
            next_x = chop(xrep-cost,0,xmax)
        else: # full reproduction
            r = c3
            next_x = chop(x-cost-c3, 0, xmax)

    return (next_x,next_t,r,e)

print "Loading data..."
dec = fromfile(decfile, Int, (xmax+1,tmax+1))

# open and prep the data file
data = open(montefile, 'w')
data.write("sim\tt\tx\td\tr\tterminal\n")

for i in range(runs):
    # set starting state
    x=4 # energy level
    t=0 # time
    r=0 # total reproduction
    terminal=0 # terminal condition

    print "Running simulation %i..." % (i)

    while t<=tmax and not terminal:
        d=dec[x,t]

        (next_x, next_t, new_r, e) = next_state(x,t,d)

        x = next_x # go to the new state
        t = next_t
        r = r + new_r # add in any new reproduction
```

```
        if e==1:
            # we were eaten
            terminal = 1
        if next_x==0:
            # starved to death
            terminal = 2

        # output a line to a file with the current state
        data.write("%i\t%i\t%i\t%i\t%i\t%i\n" % (i,t,x,d,r,terminal))

data.close()
```

Seal Foraging Dynamic Program

```
# parameters in param.py
from param import *

################################
# Modeling functions

def listmax(lst):
    """
    Determine the largest element from a list, returning the position
    and element in a pair.
    """
    maxval=lst[0]
    maxpos=0
    for i in range(len(lst)):
        if lst[i]>maxval:
            maxpos=i
            maxval=lst[i]
    return (maxpos,maxval)

def initialize(f,dec):
    """
    Initialize the last column (t=tmax) of f.

    Energy 0 -> xrep-1 will not reproduce.  Energy xrep -> xmax will have x
    offspring.
    """
    for x in range(xrep,xmax+1):
        for y in range(1,ymax+1):
            f[x,y,0,tmax] = x
            f[x,y,1,tmax] = 0
            f[x,y,2,tmax] = 0
            for h in range(hmax):
                dec[x,y,h,tmax] = h

def interpolate_xy(f,xc,yc,h,t):
    """
    Interpolate to (possibly) fractional xc,yc from neighbouring
    integer values.
    """

    # if they're dead, they aren't going to reproduce
    if xc<0:
```

```
        return 0
    if yc<0:
        return 0

    xint = int(xc) # store the integer parts
    yint = int(yc)

    dx = xc-xint   # store the fraction parts
    dy = yc-yint

    if xc>=xmax and yc>=ymax:
        # it's the bottom corner
        return f[xmax,ymax,h,t]
    elif xc>=xmax:
        # one-way interpolation at the x=xmax side
        return (1-dy)*f[xmax,yint,h,t] + dy*f[xmax,yint+1,h,t]
    elif yc>=ymax:
        # one-way interpolation at the y=ymax side
        return (1-dx)*f[xint,ymax,h,t] + dx*f[xint+1,ymax,h,t]
    else:
        return ( (1-dx)*(1-dy)*f[xint,yint,h,t]
                + (1-dx)*dy*f[xint,yint+1,h,t]
                + dx*(1-dy)*f[xint+1,yint,h,t]
                + dx*dy*f[xint+1,yint+1,h,t] )

def V(d,x,y,h,t,f):
    """
    Expected reproductive success for behaviour d
    """

    if h==0:
        # at haulout
        if d==0:
            # stay at haulout
            return survive_prob[0,0]*interpolate_xy(f, x-basal, yk, 0, t+1)
        elif d==1 and tmax-t>=haul_time:
            # haulout to surface
            return survive_prob[1,0]*interpolate_xy(f, x-haul_time*moving,
                    yk, 1, t+haul_time)
        else:
            # impossible choice
            return impossible
    elif h==1:
        # at surface
```

```
        if d==0 and tmax-t>=haul_time:
            # surface to haulout
            return survive_prob[1,0]*interpolate_xy(f, x-haul_time*moving,
                    yk, 0, t+haul_time)
        elif d==1:
            # stay at surface
            return survive_prob[1,1]*interpolate_xy(f, x-swimming,
                    y+base_o_gain+max_o_gain/(y+1), 1, t+1)
        elif d==2 and tmax-t>=dive_time:
            # surface to depth
            return survive_prob[1,2]*interpolate_xy(f, x-dive_time*moving,
                    y-dive_time*diving_o_cost, 2, t+dive_time)
        else:
            # impossible choice
            return impossible
    elif h==2:
        # at depth
        if d==1 and tmax-t>=dive_time:
            # depth to surface
            return survive_prob[2,1]*interpolate_xy(f, x-dive_time*moving,
                    y-dive_time*diving_o_cost, 1, t+dive_time)
        elif d==2:
            # stay at depth
            return survive_prob[2,2]*(
                    encounter_prob*catch_prob*interpolate_xy(f,
                        x+prey_gain-chasing, y-chasing_o_cost, 2, t+1) +
                    encounter_prob*(1-catch_prob)*interpolate_xy(f,
                        x-chasing, y-chasing_o_cost, 2, t+1) +
                    (1-encounter_prob)*interpolate_xy(f,
                        x-moving, y-diving_o_cost, 2, t+1)
                   )
        else:
            # impossible choice
            return impossible

def update(f, dec, t):
    """
    Calculate the arrays for time t, assuming times >t are already
    filled in correctly.
    """
    for x in range(0,xmax+1):
        for y in range(0,ymax+1):
            for h in range(hmax):
                choices = []
```

```
                # get appropriate V values into choices list.
                for d in range(hmax):
                    choices.append(V(d,x,y,h,t,f))

                # ...and choose the best
                (dec[x,y,h,t], f[x,y,h,t]) = listmax(choices)



##################################
# Main program


def main():
    # create zero-filled arrays for the fitness and decisions
    f = zeros((xmax+1,ymax+1,hmax,tmax+1), Float)
    dec = zeros((xmax+1,ymax+1,hmax,tmax+1), Int)

    initialize(f,dec)

    # backwards iterate from tmax-1 -> 0
    print "Running simulation..."
    for t in xrange(tmax-1, 0, -1):
        update(f, dec, t)
        if t%100==0: # print out every 100th time unit
            print t

    print "Writing data out to file..."
    f.tofile(fitfile)
    dec.tofile(decfile)

# don't run main() if we're being imported
if __name__ == '__main__':
    main()
```

Seal Foraging Parameters

```
##################################
# Model parameters

fitfile = "fitness.data"
decfile = "decision.data"

t_period = 10 # seconds--length of one time unit
e_unit = 1000.0 # kJ per energy unit
o_unit = 400.0 # mL per oxygen unit

#t_total = 1 * 24*60*60 # seconds--total length of the simulation (1 day)
t_total = 1000*t_period # 1000 units for testing

tmax = t_total/t_period # time units--total time units in simulation
haul_time=int(6000/t_period) # time units--time needed to haul out/in
dive_time=int(100/t_period) # time units--time needed to dive up/down

xrep=6 # energy units
xmax=10 # energy units

yk=6 # oxygen units
ymax=10 # oxygen units

hmax=3 # number of habitats

basal=(0.971/e_unit) # energy units
swimming=2*basal # energy units
moving=(1.625/e_unit) # energy units
chasing=2*moving # energy units
max_o_gain=(40.425/o_unit) # oxygen units
base_o_gain=max_o_gain/4 # oxygen units
diving_o_cost=(80.85/o_unit) # oxygen units
chasing_o_cost=2*diving_o_cost # oxygen units

# enc prob: P(find food in bottom time of 2 min)=1, therefore for
# one unit it is 0.08
encounter_prob=0.08 # prob of encountering prey at depth
catch_prob=.40 # prob of catching prey if encountered
prey_gain=2087/e_unit # energy units--gain if prey caught

#june 18 corrected for sleeper shark analysis of iphc-lee data
#then experiments
```

```
#haul is surface/2, dive is depth/4
haul_prob=[0.68e-8, 3.5e-5] # ...in haulout/surface transit
surf_prob=[1.36e-8, 7e-5] # ...at the surface
dive_prob=[0.395e-8, 2.8e-12] # ...in surface/depth transit
depth_prob=[1.58E-08, 2.8e-13] # ...at depth

def survive(p):
    """
    function used to combine two probabilies of depredation into prob.
    of survival
    TODO: check this vs. Clark's formula
    """
    return 1 - p[0] - p[1] + p[0]*p[1]

from numarray import *
from math import pow

# survival probabilities in various situations.
# survive_prob[h,d] is the prob of survival in habitat h, decision d
survive_prob=zeros((hmax,hmax), Float)
survive_prob[0,0] = 1.0
survive_prob[0,1] = pow(survive(haul_prob), haul_time)
survive_prob[1,0] = pow(survive(haul_prob), haul_time)
survive_prob[1,1] = survive(surf_prob)
survive_prob[1,2] = pow(survive(dive_prob), dive_time)
survive_prob[2,1] = pow(survive(dive_prob), dive_time)
survive_prob[2,2] = survive(depth_prob)

impossible=-1 # the fitness if the seal tries to do impossible things
```

Seal Foraging Monte Carlo

```
ssfile="seal-ss.txt"
filename="seal-monte.txt"

################################
# Module import

from math import floor
from random import random
# random number generator is seeded on import

# parameters in param.py
from param import *

def prob_round(x):
    """
    Probabalistically round x to the integer above or below
    For x.y, return x+1 with prob y and x with prob 1-y

    Use of this justified in Clark/Mangel, p. 115
    """
    whole = int(floor(x))
    frac = x-whole
    if random()<frac:
        return whole+1
    else:
        return whole

def rv(p):
    """
    Return true with probability p
    """
    if random()<p:
        return 1
    else:
        return 0

def state(terminal):
    """
    Translate terminal condition code to a word
    """
    if terminal==0:
        return ""
```

```
    elif terminal==1:
        return "starved"
    elif terminal==2:
        return "drowned"
    elif terminal==3:
        return "eaten"
    elif terminal==4:
        return "alive"

def ss_line(i,t,x,y,h,d,terminal):
    line = "%i\t%i\t%i\t%i\t%i\t%i\t%s\n" % (i,t,x,y,h,d,state(terminal))
    return line

def next_state(x,y,h,d,dec):
    if h==0:
        # at haulout
        if d==0:
            # stay at haulout
            x1 = prob_round(x-basal)
            y1 = prob_round(yk)
            t1 = t+1
        elif d==1 and tmax-t>=haul_time:
            # haulout to surface
            x1 = prob_round(x-haul_time*moving)
            y1 = prob_round(yk)
            t1 = t+haul_time
    elif h==1:
        # at surface
        if d==0 and tmax-t>=haul_time:
            # surface to haulout
            x1 = prob_round(x-haul_time*moving)
            y1 = prob_round(yk)
            t1 = t+haul_time
        elif d==1:
            # stay at surface
            x1 = prob_round(x-basal)
            y1 = prob_round(y+base_o_gain+max_o_gain/(y+1))
            t1 = t+1
        elif d==2 and tmax-t>=dive_time:
            # surface to depth
            x1 = prob_round(x-dive_time*moving)
            y1 = prob_round(y-dive_time*diving_o_cost)
            t1 = t+dive_time
    elif h==2:
```

```
        # at depth
        if d==1 and tmax-t>=dive_time:
            # depth to surface
            x1 = prob_round(x-dive_time*moving)
            y1 = prob_round(y-dive_time*diving_o_cost)
            t1 = t+dive_time
        elif d==2:
            # stay at depth
            if rv(encounter_prob):
                # see some prey...
                if rv(catch_prob):
                    # ...caught it
                    x1 = prob_round(x+prey_gain-chasing)
                    y1 = prob_round(y-chasing_o_cost)
                    t1 = t+1
                else:
                    # ...missed it
                    x1 = prob_round(x-chasing)
                    y1 = prob_round(y-chasing_o_cost)
                    t1 = t+1
            else:
                # still looking...
                x1 = prob_round(x-moving)
                y1 = prob_round(y-diving_o_cost)
                t1 = t+1

    x=max(0,min(x,xmax)) # stay in-bounds
    y=max(0,min(y,ymax))
    return (x1,y1,t1)

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

data = open(filename, 'w')
ssdata = open(ssfile, 'w')

ssdata.write("Sim Num\tTime\tEnergy\tOxygen\tHabitat\tDecision\tEndCond\n")

runs=10
for i in range(runs):
    x=2
    y=6
    h=0
```

```
    t=0
    last=()

    print "Running simulation %i..." % (i)
    terminal=0
    while t<=tmax and not terminal:
        d=dec[x,y,h,t]

        # report any changes in its state
        if (x,y,h,d)!=last:
            data.write("Time %i, State (%i,%i), Habitat %i, Decision %i\n"
                    % (t,x,y,h,d) )
            ssdata.write( ss_line(i,t,x,y,h,d,terminal) )
            last=(x,y,h,d)

        # possible terminal conditions
        if x<=0:
            data.write("Starved to death.\n")
            terminal=1
        if y<0:
            data.write("Drowned.\n")
            terminal=2
        if rv(1-survive_prob[h,d]):
            data.write("Eaten.\n")
            terminal=3

        # figure out what happens next
        if not terminal:
            (x,y,t) = next_state(x,y,h,d,dec)

        # prepare for the next cycle
        h=d

    if terminal==0:
        # we got out alive
        terminal=4
    data.write("Simulation ends at time %i\n\n" % (t))
    ssdata.write( ss_line(i,t,x,y,h,d,terminal) )

data.close
ssdata.close
```