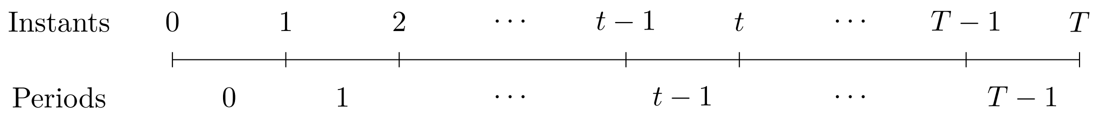
Instituto Tecnológico Autónomo de México
Maestría en Teoría Económica / Licenciatura en Economía

Dynamic Programming

Computing the "Cake-Eating" Problem, Fall 2014

Ricard Torres

Given an integer $T \geq 1$, consider a discrete-time problem with time horizon T that has the following structure:



Given $A > 0$, we are going to consider the following "cake-eating" problem:

$$\begin{aligned} \max_{(x_t)_{t=0}^{T-1}} \quad & \sum_{t=0}^{T-1} \beta^t \log(x_t) \\ \text{s.t.} \quad & y_{t+1} = y_t - x_t, \qquad 0 \leq t \leq T-1 \\ & x_t \geq 0,\ y_{t+1} \geq 0, \quad 0 \leq t \leq T-1 \\ & y_0 = A \end{aligned}$$

It can be interpreted, for example, as a growth problem with a linear (net) production function, or as a resource-extraction problem. We will also consider the infinite-horizon version:

$$\begin{aligned} \max_{(x_t)_{t=0}^{\infty}} \quad & \sum_{t=0}^{\infty} \beta^t \log(x_t) \\ \text{s.t.} \quad & y_{t+1} = y_t - x_t, \qquad t \geq 0 \\ & x_t \geq 0,\ y_{t+1} \geq 0, \quad t \geq 0 \\ & y_0 = A \end{aligned}$$

In the case of a finite horizon T, the "Bellman equation" of the problem consists of an inductive definition of the current value functions, given by $v(y, 0) \equiv 0$, and, for $n \geq 1$,

$$\begin{aligned} v(y, n) = \max_{x} \ & \log(x) + \beta\, v(y - x, n-1) \\ \text{s.t.} \quad & 0 \leq x \leq y \end{aligned}$$

Where n represents the number of periods remaining until the last instant T.

When the horizon is infinite, the Bellman equation becomes a fixed-point problem for the current value function:

$$\begin{aligned} v(y) = \max_{x} \ & \log(x) + \beta\, v(y - x) \\ \text{s.t.} \quad & 0 \leq x \leq y \end{aligned}$$

We have found in class that the solution for those problems can be easily found analytically by different methods: discrete optimal control, calculus of variations, value or policy recursion for dynamic programming, or even the method of indeterminate coefficients for the latter formulation. In the case of a finite horizon, the solutions are:

$$
\begin{aligned}
x_t(A) &= \frac{(1-\beta)\beta^t}{1-\beta^T}\,A & (0 \le t \le T-1)\\
y_t(A) &= \frac{\beta^t-\beta^T}{1-\beta^T}\,A & (0 \le t \le T)\\
v(y,n) &= \frac{1-\beta^n}{1-\beta}\,\log\left(\frac{1-\beta}{1-\beta^n}\,y\right) + \left[\frac{\beta\,(1-\beta^{n-1})}{(1-\beta)^2} - \frac{(n-1)\beta^n}{1-\beta}\right]\log(\beta) & (0 \le n \le T)
\end{aligned}
$$

Here t denotes, at a certain instant in time, the number of periods elapsed from the initial instant 0, and n the number of periods remaining till the time horizon T, so that $t+n=T$.

Moreover, the (bounded) solution for the infinite-horizon case is well approximated by the limits of the solutions for the finite-horizon case when $T\to\infty$ (so that $n\to\infty$ also):

$$
\begin{aligned}
x_t(A) &= (1-\beta)\,\beta^t\,A & (t\ge 0)\\
y_t(A) &= \beta^t\,A & (t\ge 0)\\
v(y) &= \frac{1}{1-\beta}\,\log\left[(1-\beta)\,y\right] + \frac{\beta}{(1-\beta)^2}\,\log(\beta)
\end{aligned}
$$

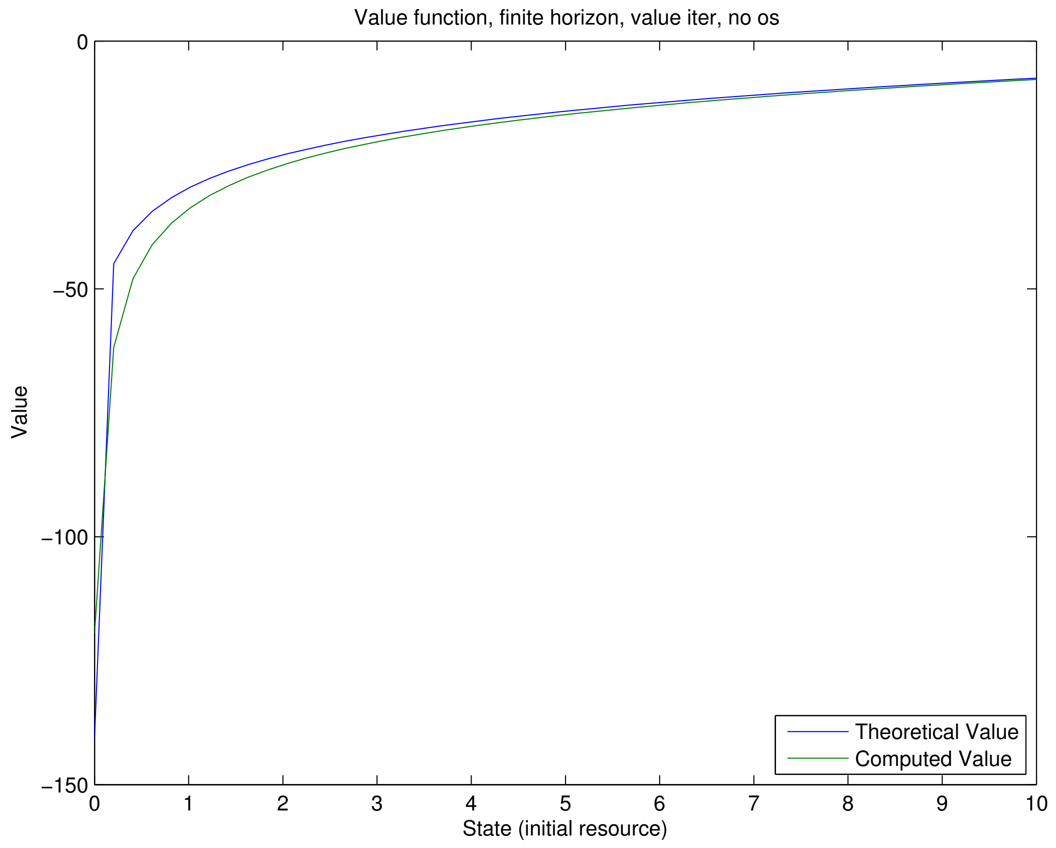
We are interested in programming the computation of these results in `Matlab`. This is more complicated than previous computations we have seen because the choice set is now infinite. In order to tackle this problem, we are going to use "fitted value iteration" (see section 6.2.2 in John Stachursky's book). In principle the idea is simple: we define a finite grid of points for state and control variables, and evaluate the functions at the grid points, using linear interpolation when evaluation at other points is needed. For our "cake-eating" problem, let us set up the initial values. Note that, the way things are defined, there is a relationship between the initial amount of resource A and the number of grid points necessary in order to approximate the state (we could instead define a density of grid points for a given interval length).

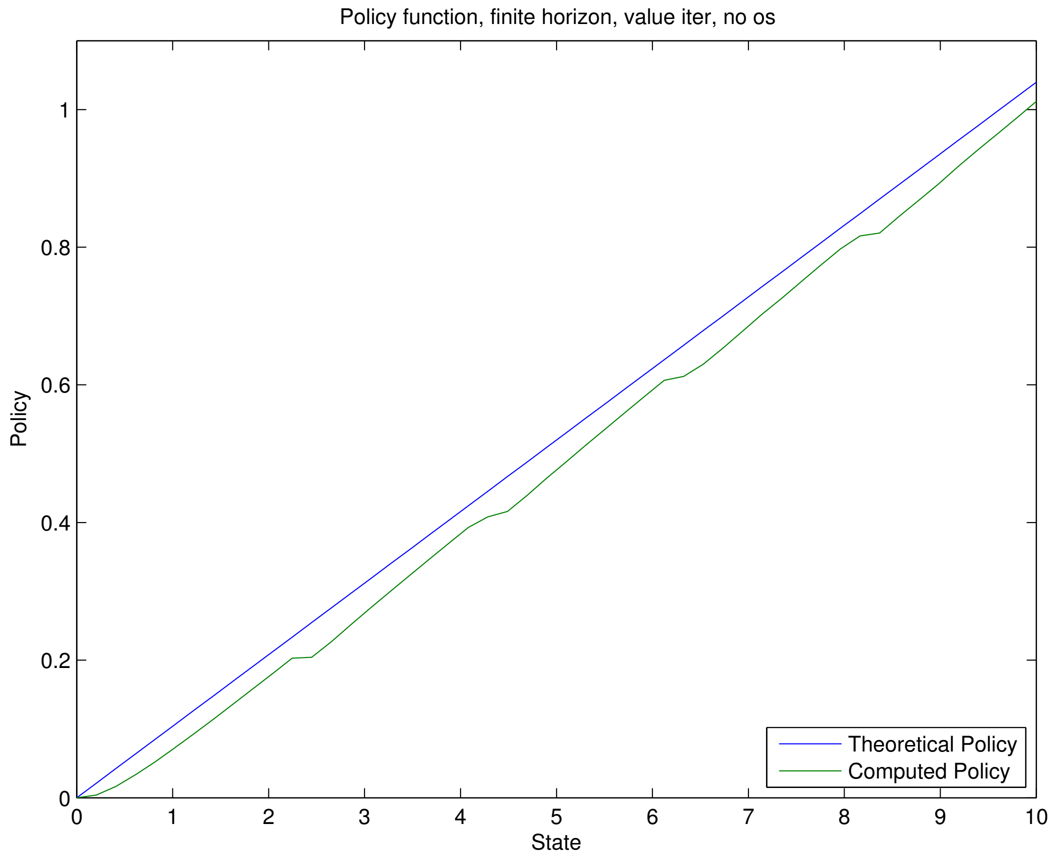
```
%% Initial amount of resource
initial_resource = 10 ;
%% Time horizon
tmax = 30 ;
%% Discount factor
beta = 0.9 ;
%% Instantaneous utility
utility = @(x) log(x) ;
%% Grid size
grid = 50 ;
```

We want to avoid the initial state in any period having the value 0, because this would result in a negative infinite value for the objective function, so we set the minimum allowed state to a positive, but small value. Let us act naively and set the grid to an equally spaced set of points:

```
min_state = 1e-5;
state = linspace(min_state,initial_resource,grid);
```

Let us now advance the result, given this definition of the state, of the computations based on the Bellman equation, in order to see how the value and policy functions at period 0 for the given inital resource compare to the actual ones we have shown above.



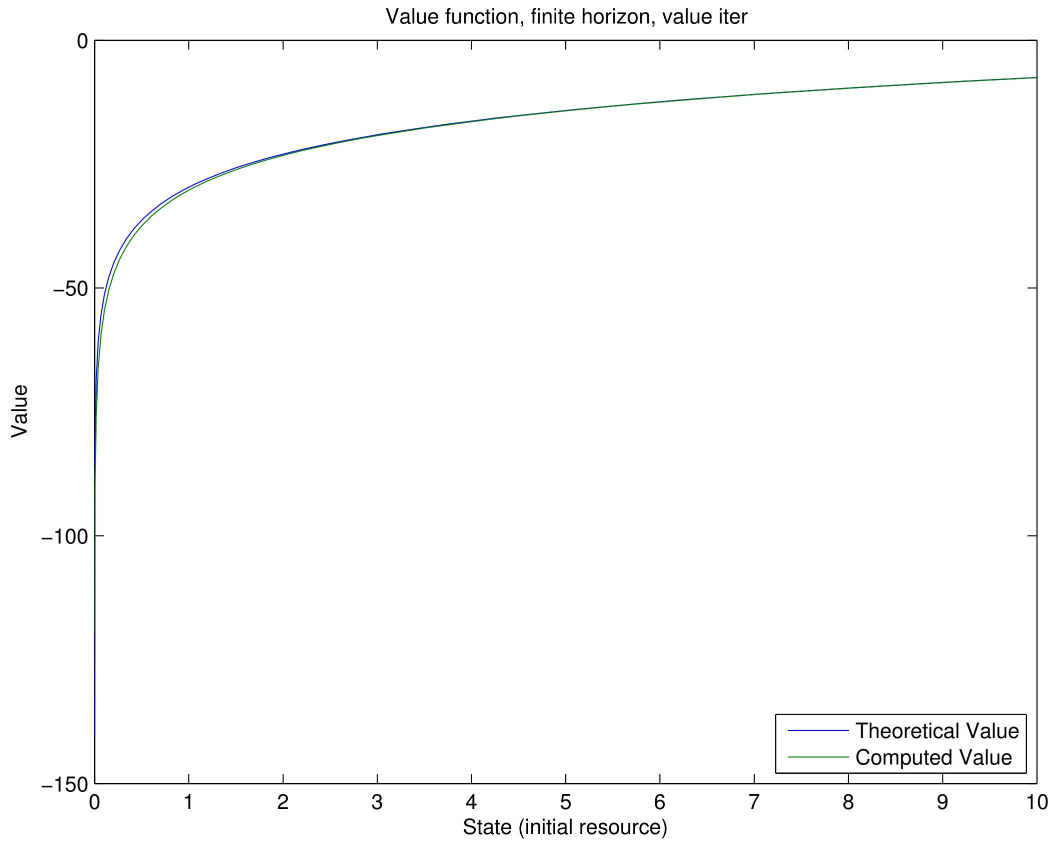


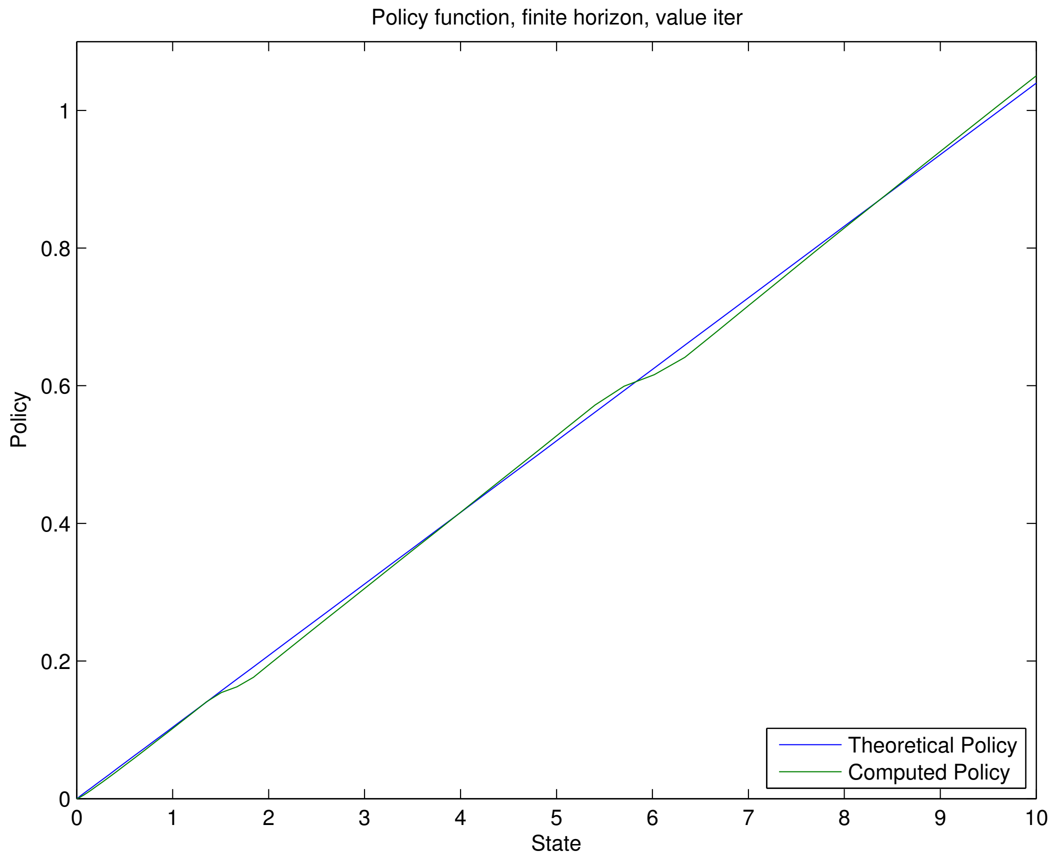
You may appreciate that there is a systematic disparity, more visible in the case of the policy function. The problem stems from the fact that the value function inherits the concavity property of the logarithm, so its derivative (rate of change) is very high for low values of the state, but low for high values. So when the value of the state is high enough a few grid points suffice to get a good approximation to the value function, but when the value is low we need a lot of grid points in order to get a decent approximation. Note that, even though our approximation is bad for low values of the state, but good for high values, the policy function has a systematic estimation error over the entire domain: this is due to the fact that our estimates are the result of a recursive procedure, by which initial high values of the state will depend on progressively low future values.

We may increase the number of grid points, but for problems with a large time horizon, or with an infinite one, the corresponding increase in computation time is much more than proportional. A simpler solution exists: oversampling low values with respect to high ones. The simplest way to do this is to apply a quadratic transformation to the grid points (making sure the extremes of the interval do not change):

```
state = linspace(min_state^(1/2),initial_resource^(1/2),grid).^2 ;
```

If we want more oversampling, we just need to increase the exponent. But just with quadratic oversampling in this case we get a good approximation to the value and policy functions. For problems in which we do not know the true value and policy functions, but we know that the value function is concave, it is probably better to play it safe and increase the exponent (for instance, John Stachursky uses an exponent 10 in the scripts of his book). This should depend on the initial functions used in our particular problem. If we have a problem in which the value function is convex, we would need to oversample for high values instead of low ones. As a general rule, the more we know about the problem, the better will be our numerical approximation, because we will be able to tailor our procedure for the peculiarities of that problem. Let us show the difference that this asymmetric grid makes in our computations:





Let us now present the code we have used in order to obtain those approximations, continuing from our definition of the grid based on oversampling for low values. The value function will be inferred from the images taken at the grid points (using linear interpolation for other points), so it will be a vector with `grid` points. We are going to use the Bellman equation in order to recursively define the value function, but since we only care about the value function at the initial period, we can do with just two vectors: one for the current value function and another for the previous one. Analogously, we will just record the policy function at the initial period.

```
%% Value function for current and previous periods
value = zeros(2, grid) ;
%% Optimal policy for first period
policy = zeros(1, grid) ;
```

For the value recursion algorithm, the beginning of the actual coding begins with setting the value function at the last period (alternatively, we might set it identically to 0 after the last period has ended).

```
%% Final value function (last period there is no maximization):
current = 1 ;
period = tmax - 1 ;
for i = 1:grid
    value(current, i) = utility(state(i));
end
```

Now we can set up the main loop in which value recursion is implemented. Observe that we have previously set `current = 1` and `period = tmax - 1`.

```
%% Main loop, backwards induction
while ( period >= 0 )
    period = period - 1;
    % Exchange 'previous' and 'current'
    previous = current ;
    current = 1 + ( previous == 1 );
    for i = 1:grid % state values
        % want: 0 < control < state
        min_control = min_state/10 ;
        clear w ; % (negative of) objective function in Bellman equation
        w = @(x) - utility(x) - beta * ...
              myinterp1(state,value(previous,:),state(i) - x) ;
        [ newpol, newval ] = fminbnd(w, min_control, state(i)-min_control);
        value(current,i) = - newval; % current value function
        if ( ~ period ) % record policy only in period 0
           policy(i) = newpol ;
        end
    end
end
```

Some comments on the code:

- In an algebraic expression, the logical test `( previous == 1 )` takes the integer value `1` or `0`, depending on whether it is true or false. In this case, this results in `current` taking the value in $\{1, 2\}$ which `previous` does not have. We might alternatively use, for example, the modulus operation (remainder of integer division), but this type of logical tests are generally very useful, for example when defining a piecewise linear function, as in
`f = @(x) ( x < 1 ) * ( 2 * x ) + ( x >= 1 ) * ( 2 + 3 * (x - 1) )`
- With linear interpolation, there is always a maximizer which is a point in the grid, but it seems more efficient, instead of finding the maximum of a vector, to use the `fminbnd` function, which is based on a "golden search" algorithm. Additionally, using anonymous functions the code is more compact and easier to interpret.
- For some reason, the `Matlab` built-in function `interp1` does not interpolate when the given point is not inside the interval of domain points. So we created a function `myinterp1` that ignores this restriction. Alternatively, for points outside the domain interval we might set the function to the value at the corresponding extreme, but our method should generally yield better estimates.
- In the minimization process, we make sure that the control variable is strictly positive and strictly below the state variable (so next period's state will be strictly positive).

The rest is just a matter of recording the results, comparing them to the theoretical ones, and possibly plotting them for easier visualization. The details are in the value iteration script for the finite horizon case: `http://ciep.itam.mx/~rtorres/progdin/cake_value_finite.m`.

Now, the method just implemented is the value iteration algorithm based on the Bellman equation. We might instead use *policy iteration*. In the finite case, a policy function depends on two arguments: the state, but also the time period. Additionally, policy iteration requires computing the value function (with the same two arguments) that corresponds to a given policy function. Since in our implementation of value iteration we did not record the full value and policy functions, we should expect that, in this finite-horizon problem, policy iteration spends more computing time than value iteration. In order to implement policy iteration, the variables that are different from the previous ones or new are:

```
%% tolerance for policy comparison
tol = 1e-05 ;
%% Policy (tmax periods: 0 \le t \le tmax-1)
%% (in Matlab indices start at 1, so indices will be shifted 1 position)
policy = zeros(tmax, grid) ;
%% Value function
value = zeros(tmax, grid) ;
```

Our first step will consist of setting an arbitrary initial policy:

```
%% An arbitrary initial policy
iterations = 0 ;
%% Last period all is consumed
period = tmax - 1;
for i = 1:grid
    policy(period+1, i) = state(i) ;
end
%% Previous periods: consume 0.5 initial state
for period = tmax-2:-1:0
    for i = 1:grid
        policy(period+1, i) = 0.5 * state(i) ;
    end
end
```

In the policy iteration routine, the first step will consist of computing the value function that corresponds to the given policy:

```
% Compute value of policy
% Last period
period = tmax - 1;
for i = 1:grid
  value(period+1,i) = utility(policy(period+1,i));
end
% periods tmax-2 to 0
for period = tmax-2:-1:0
  for i = 1:grid
      value(period+1,i) = utility(policy(period+1,i)) + ...
         beta * myinterp1(state, value(period+2,:), ...
                      state(i) - policy(period+1,i) ) ;
  end
end
```

The policy iteration routine begins with:

```
while ( iterations <= max_iterations )
  iterations = iterations + 1 ;
```

Then comes the value computation we showed above, and next the policy iteration proper:

```
  % Record whether there is a change in policy
  policies_differ = false ;
  % Perform one-step maximization process
  % (last period policy is unchanged)
  for period = tmax-2:-1:0
    for i = 1:grid
       % want: 0 < control < state
       min_control = min_state/10 ;
       clear w ; % (negative of) objective function
       w = @(x) - utility(x) - beta * ...
             myinterp1(state,value(period+2,:),state(i) - x) ;
       [ newpol, newval ] = fminbnd(w, min_control,state(i)-min_control);
       value(period+1,i) = - newval ;
       if ( abs( policy(period+1,i) - newpol ) > tol )
          policy(period+1, i) = newpol ;
          policies_differ = true;
       end
    end
  end
  if ( ~ policies_differ ), break; end
end
```

The core of the procedure consists of the one-step maximization routine, in which for each period and state value the policy is changed to that value of the control which maximizes the objective function consisting of the Bellman equation when next period's value is substituted by the value that corresponds to the previous policy. If two consecutive policies are the same (within a given tolerance), then we are done. The policy iteration script for the finite-horizon case is:
`http://ciep.itam.mx/~rtorres/progdin/cake_policy_finite.m`.

For our problem, the results of value and policy iteration are very similar, but while the first procedure takes 3.523029 seconds, the second takes 7.485390 seconds. The motive for this is explained above.

Let us now consider the problem with an *infinite horizon*. For *value iteration*, the environment set up is much the same as in the finite-horizon case, except for the fact that now, instead of a given time horizon, we have to define a tolerance level for comparison of successive value functions (at the grid points). We should also set a limit to the number of iterations that are allowed.

```
%% tolerance for value comparison
tol = 1e-06 ;
%% Max allowed number of iterations
max_iterations = 500 ;
```

Our first step consists of setting a starting point for the value iteration process:

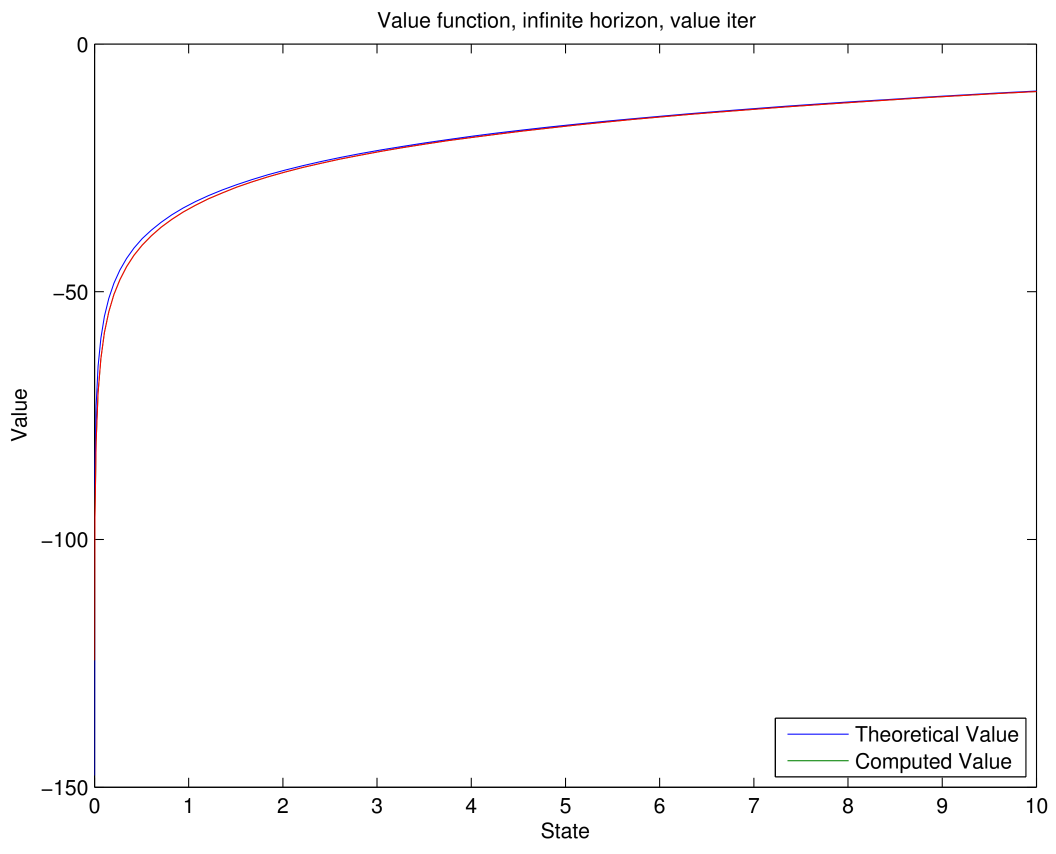
```
%% Initial iteration (there is no maximization)
iteration = 0 ;
current = 1;
for i = 1:grid
    policy(i) = state(i) ;
    value(current, i) = utility(state(i));
end
```

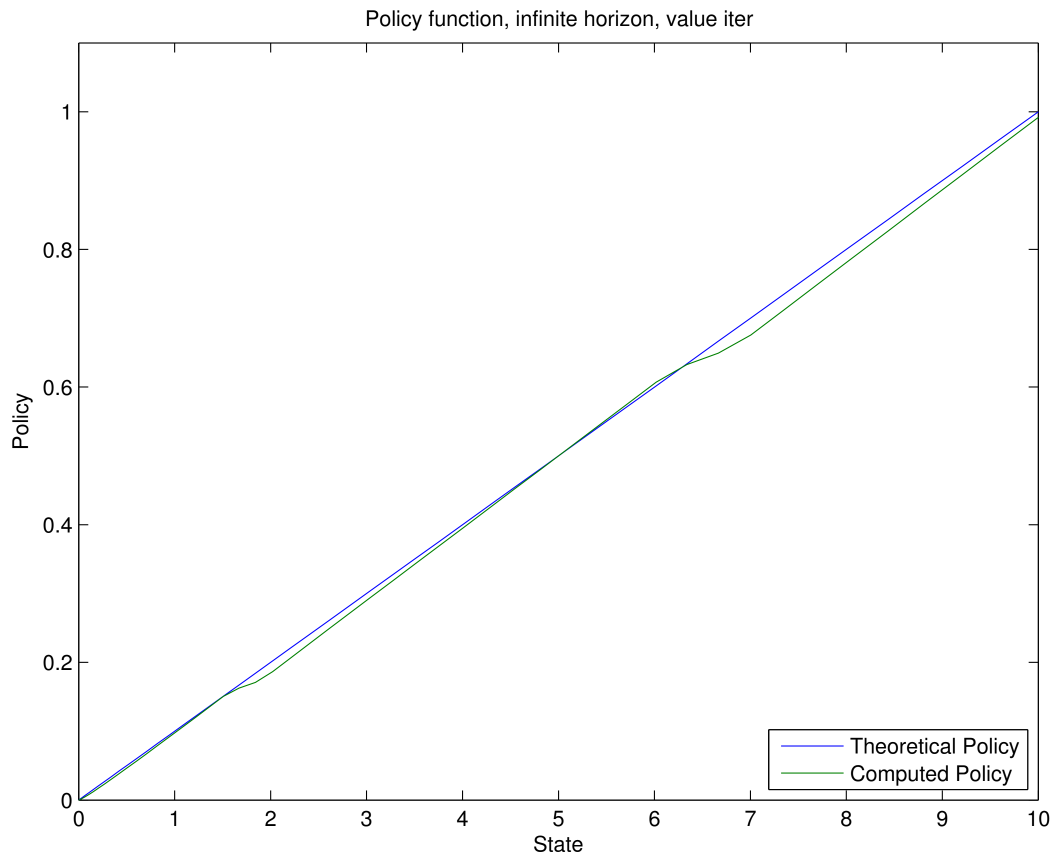
The value iteration routine is:

```
%% Main loop, backwards induction
while ( iteration < max_iterations )
    iteration = iteration + 1;
    values_differ = false ;
    % Exchange 'previous' and 'current'
    previous = current ;
    current = 1 + ( previous == 1 );
    for i = 1:grid % state values
        % want: 0 < control < state
        min_control = min_state/10 ;
        clear w ; % (negative of) objective function in Bellman equation
        w = @(x) - utility(x) - beta * ...
               myinterp1(state,value(previous,:),state(i) - x) ;
        [ newpol, newval ] = fminbnd(w, min_control, state(i)-min_control);
        value(current,i) = - newval; % current value function
        policy(i) = newpol ;         % current policy function
        if ( abs(value(current, i) - value(previous,i)) > tol )
            values_differ = true ;
        end
    end
    if ( ~ values_differ ), break; end ;
end
```

You may notice how similar it is to the routine in the finite horizon case. One difference is that here we do record the policy for each iteration. The normal termination occurs not when we reach `max_iterations` in the `while` test, but in the last test when we check whether two consecutive value functions are the same (within the defined tolerance).

Below are the resulting value function and stationary policy function, compared to the theoretical ones. Given the small number of grid points we took, the approximation is remarkably good. Of course, it would look much better (at this level of resolution practically indistinguishable) if we doubled or trebled the number of grid points.


Value function, infinite horizon, value iter
Value
State
Theoretical Value
Computed Value
0
−50
−100
−150
0
1
2
3
4
5
6
7
8
9
10


Policy function, infinite horizon, value iter
Policy
State
Theoretical Policy
Computed Policy
0
0.2
0.4
0.6
0.8
1
0
1
2
3
4
5
6
7
8
9
10

Finally, let us consider the *policy iteration* routine. Given a stationary policy, in order to compute the value associated with this policy we have two alternatives: (1) fix a large enough time horizon and use the corresponding value to approximate the one for the infinite horizon; or (2) do as before, but instead of using a fixed time horizon keep iterating until the distance between successive values falls within a given tolerance level. Here, we will use alternative (2). So our initial parameters will now include:

```
%% tolerance for policy comparison
pol_tol = 1e-04 ;
%% tolerance for value comparison
val_tol = 1e-04 ;
%% Max allowed number of iterations in order to find the policy
max_pol_iterations = 100 ;
%% Max iterations to compute values associated with a policy
max_val_iterations = 100 ;
```

The procedure begins by setting an arbitrary feasible policy:

```
pol_iterations = 0 ;
for i = 1:grid
    policy(i) = 0.5 * state(i) ;
end
```

As in the finite case, the first step in the policy iteration routine entails finding the value associated with that policy, which here involves approximating a fixed point:

```
values_differ = false ;
val_iterations = 0 ;
current = 1 ;
for i = 1:grid
  value(current, i) = utility(policy(i));
end
while ( val_iterations < max_val_iterations )
  val_iterations = val_iterations + 1 ;
  % exchange previous and current
  previous = current ;
  current = 1 + ( previous == 1 ) ;
  for i = 1:grid % state values
      value(current,i) = utility(policy(i)) + beta * ...
            myinterp1(state,value(previous,:), state(i) - policy(i) ) ;
      if ( abs(value(current, i) - value(previous,i)) > val_tol )
          values_differ = true ;
      end
  end
  if ( ~ values_differ ), break; end ;
end
```

The beginning of the policy iteration routine is:

```
while ( pol_iterations < max_pol_iterations )
  pol_iterations = pol_iterations + 1 ;
```

And after the value computation shown above the continuation is:

```
  % Record whether there is a change in policy
  policies_differ = false ;
  % Perform one-step maximization process
  for i = 1:grid
     % want: 0 < control < state
     min_control = min_state/10 ;
     clear w ; % (negative of) objective function
     w = @(x) - utility(x) - beta * ...
           myinterp1(state,value(current,:),state(i) - x) ;
     [ newpol, newval ] = fminbnd( w, min_control, state(i)-min_control );
     % We are interested in the maximizer, not the max value
     if ( abs( policy(i) - newpol ) > pol_tol )
         policy(i) = newpol ;
         policies_differ = true;
     end
  end
  if ( ~ policies_differ ), break; end
end
```

In this case, the stationarity of the policy function will render this procedure more efficient than value iteration. As a matter of fact, the number of policy iterations required is typically very small. For our scripts, value iteration converged in 156 iterations, and took 19.292574 seconds of computation time. On the other hand, policy iteration converged in 5 iterations, and took 3.362780 seconds. The results of both procedures are very similar. With `Octave`, we find the same results, but it takes more time: the value iteration takes 238.062357 seconds, while the policy iteration takes 39.076641 seconds. Naturally, all the times reported here depend on the computer, operating system, desktop environment, and program version, but we are interested in the relative times, rather than the absolute ones.

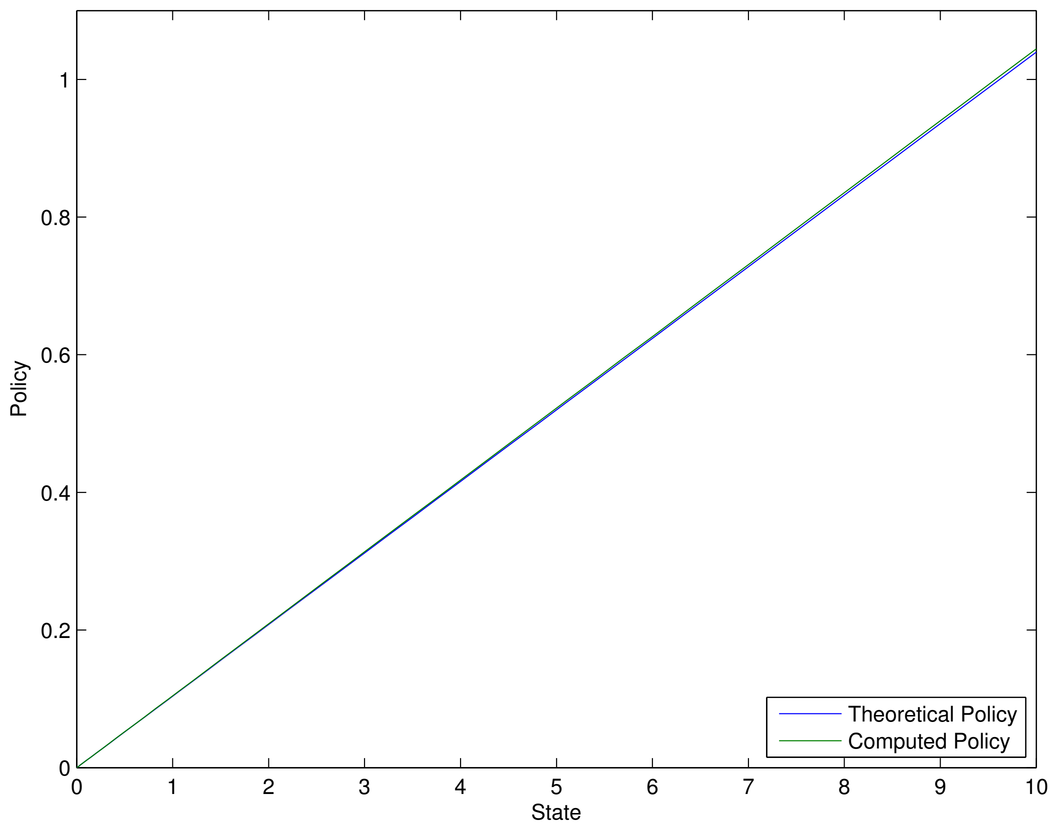
The value iteration script for the infinite-horizon case is:
`http://ciep.itam.mx/~rtorres/progdin/cake_value_infinite.m`.

And the policy iteration script for the infinite-horizon case:
`http://ciep.itam.mx/~rtorres/progdin/cake_policy_infinite.m`.

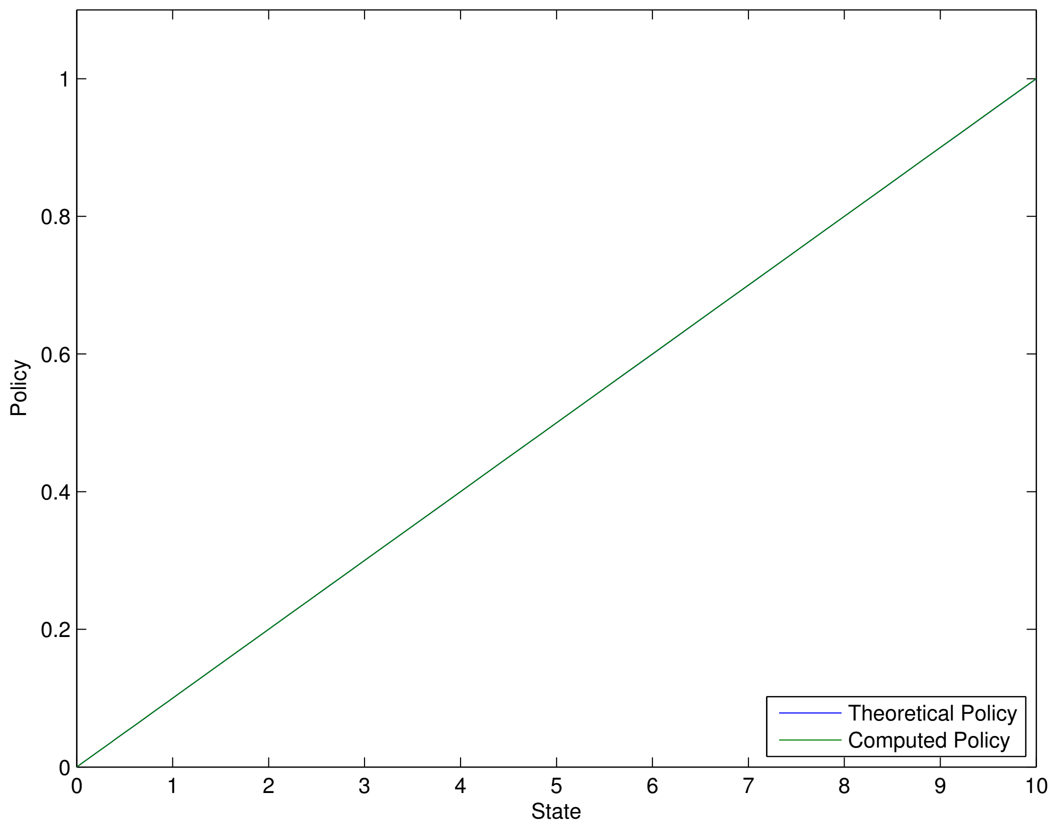
A final word on the computation. You may feel a little uneasy about the fact that, in both the finite and the infinite horizon cases, the computed policy has visible differences from the (linear) theoretical one. One way to get rid of this with minimal modification is to use cubic splines instead of linear interpolation: this amounts to replacing `myinterp1(a,b,c)` by `interp1(a,b,c,'spline')` (for this method the built-in function does not have the domain restriction). However, cubic splines need not be monotonic even though our datapoints are, so it is still better to trade some smoothness in exchange for monotonicity preservation and use piecewise cubic Hermite interpolating polynomials, appealing to the built-in function `pchip`, with syntax `pchip(a,b,c)`. If you do this, you will see that the computed policy function is essentially identical to the theoretical one. However, the computation time becomes, for the infinite horizon case with the value iteration algorithm, of 54.881263 seconds (14.402512 seconds with policy iteration). In the next page you may see the resulting policy estimations for the value iteration procedure in the case of both finite and infinite horizon.

An alternative, as we mention above, is to keep linear interpolation but increase the grid size and impose tighter tolerances. Both methods yield more precise approximations, at the cost of more computation time: which one is more efficient will depend on the problem we are dealing with. For smooth problems like the present one, smoothing things out with splines may be more efficient than the alternative, more brute force, approach.

Policy function, finite horizon, value iter, pchip approx



Policy function, infinite horizon, value iter, pchip approx


Policy
State
Theoretical Policy
Computed Policy