dynprog

Unknown Author

October 08, 2013

Part I

Infinite Horizon Dynamic Programming

1 Notebook Prelimaries

In [5]:
```
from __future__ import division
import math, random, os, sys

qedir = '../../../../quant-econ/trunk/programs'
qedir = os.path.abspath(qedir)
if qedir not in sys.path:
    sys.path.append(qedir)

%matplotlib inline
import numpy as np
import scipy as sp
from scipy.optimize import fminbound
import matplotlib.pyplot as plt
```

2 Overview

Goal: solve simple infinite-horizon dynamic programming problems using Python.

Application: consumption in an optimal growth model, very closely following Sargent and Stachursky (2013) (SS).

Background reading: http://quant-econ.net/short_path.html#short-path and [stokey_lucas-1989], chapters 2–5.

3 A Growth Model

An agent (or economy) at time t owns capital $k_t \in R_+$. Period t production net of depreciation is $f(k_t) \in R_+$, which is consumed or saved. Next period's capital is determined by this period's saving. We can therefore specify the transition of the state of the economy as

(1)

$$k_{t+1} = f(k_t) - c_t$$

Feasibility

Given k_0, a consumption path is feasible if (1) implies a non-negative capital stock sequence, $(k_1, k_2, \dots)$.

We will think of consumption as being determined by a "policy function":

(2)

$$c_t = \sigma(k_t)$$

The policy function $\sigma : R_+ \to R_+$ is a feasible consumption policy if:

(3)

$$0 \leq \sigma(k) \leq f(k) \quad \forall k \in R_+$$

Given k_0 and a policy function σ, the sequence $\{k_t\}$ in (4) is determined by

(4 (compare SS5))

$$k_{t+1} = f(k_t) - \sigma(k_t)$$

Utility

In period t, our agent derives utility $u(c_t)$ from consumption c_t. Utility is discounted at rate $\beta \in (0, 1)$. So period t utility is discounted to period 0 as $\beta^t u(c_t)$. A consumption sequence $(c_0, c_1, c_2, \dots)$ is therefore valued at

(SS2)

$$\sum_{t=0}^{\infty} \beta^t u(c_t)$$

(Assume u is a strictly increasing, strictly concave utility function.)

3.1 Optimality

Over an infinite horizon, an optimizing agent chooses consumption to maximize discounted utility. An optimal path provides at least as much utility as any other feasible path. We would like to know when a solution exists.

Sufficient conditions for optimization often include continuity of the objective and compactness of the feasible set, so we should expect these to use these. In addition, in the infinite horizon case, we must bound the sum of discounted utilities. If we do so, [stokey_lucas-1989] (section 4.1) show that optimal consumption is a time-homogeneous function of the current state. That is, there exists a function σ such that $c_t = \sigma(k_t)$ for all t.

Determining the policy function σ is more useful than determining the optimal consumption sequence, because we can more easily apply the concept of a policy function to the stochastic case.

So we know it makes sense to reformulate the optimization problem as a search for a best policy. Letting Σ denote the set of feasible policies, the agent's decision problem can be rewritten as

(SS4)

$$\max_{\sigma \in \Sigma} \sum_{t=0}^{\infty} \beta^t u(\sigma(k_t))$$

3.2 Dynamic Programming

We can use dynamic programming to find the optimal policy.

Step 1: define the value function associated with this optimization problem. An optimal policy function attains the supremum in (6) for all $k_0 \in R_+$.

(SS6)

$$v^*(k_0) := \sup_{\sigma \in \Sigma} \sum_{t=0}^{\infty} \beta^t u(\sigma(k_t))$$

when {kt} is given by (4 (or SS5))

Recursive Reformulation: The Bellman Equation

(SS7)

$$v^*(k) = \max_{0 \leq c \leq f(k)} u(c) + \beta v^*(f(k) - c)$$

for all $k \in R_+$.

We optimize by choosing c to appropriately trade off current utility for discounted future utilities that depend on our savings.

Greedy Policy

Definition: A "greedy" algorithm makes a locally optimal choice at each step as a strategy for approximating a global optimum. Correspondingly, given a continuous function w on R_+, we say a policy $\sigma \in \Sigma$ is w-greedy if $\sigma(k)$ is a solution to

(SS8)

$$\max_{0 \leq c \leq f(k)} u(c) + \beta w(f(k) - c)$$

for all $k \in R_+$.

Theoretical Results (as summarized by SS)

Well-established results exist for bounded u, as long as f and u are continuous. For example, an optimal policy exists, and the value function v^* is finite, bounded, continuous and satisfies the Bellman equation. Helpfully, a policy is optimal iff it is v^*-greedy. (See e.g. [EDTC2009]_ proposition 10.1.13.)

This suggests a computational approach to finding the optimal policy:

- compute v^*
- solve for a v^*-greedy policy

For any k, the second step becomes simple: solving the one-dimensional optimization problem on the RHS of (7) But first, we must obtain the value function

3.3 Bellman Operator

The Bellman operator maps a function w into a new function Tw, as follows:

(SS9)

$$Tw(k) := \max_{0 \leq c \leq k} u(c) + \beta w(f(k) - c)$$

Note that while this looks a lot like the Bellman equation, it is a very different animal. Iteratively applying T from some starting function w produces a sequence $(w, Tw, T(Tw), \dots)$

3.4 Value Function Iteration

The value function v^* may be obtained by an iterative technique:

Approach: start with a guess (an initial function w) and successively improve it as $w = Tw$.

Improvement step: apply the "Bellman operator".

Unbounded Utility

We can ensure this is a sequence of continuous bounded functions that converges uniformly to v^*. (For a proof see lemma 10.1.20 of [EDTC2009].)

However, this theoretical result is often shown under the assumption that the utility function is bounded, and economists often work with unbounded utility functions. The result has been extended to utility functions that are bounded below. (See e.g. section 12.2 of [EDTC2009].)

For utility functions that are unbounded both below and above the situation is more complicated. Recent work on deterministic problems includes [Kamihigashi2012] and [MV2010].

.. [EDTC2009] Economic Dynamics: Theory and Computation by John Stachurski

.. [Kamihigashi2012] Kamihigashi, T. (2012). Elementary Results on Solutions to the Bellman Equation of Dynamic Programming: Existence, Uniqueness, and Convergence, Kobe University RIEB DP2012-31.

.. [MV2010] Martins-da-Rocha, V.F., Vailakis, Y., 2010, Existence and uniqueness of a fixed point for local contractions, Econometrica, 78, 1127–1141.

3.5 Value Function Iteration

Suppose we would like to compute the value function by iterating with the Bellman operator. Given an inital guess w and a transformation T, we would like to implement the following algorithm:

Until a stopping condition is satisfied, set $w = Tw$.

Our transformation $w \mapsto Tw$ is the solution to (9).

Analytical Solution

Let $f(k) = k^\alpha$ and $u(c) = \ln(c)$.

This particular problem allows an exact analytical solution ([RMT3]_ section 3.1.2; also, the online notes):

(SS10)

$$v^*(k) = c1 + c2 \ln(k)$$

where

$$c1 := \frac{\ln(1-\alpha\beta)}{1-\beta} + \frac{\ln(\alpha\beta)\alpha\beta}{(1-\alpha\beta)(1-\beta)} \qquad c2 := \frac{\alpha}{1-\alpha\beta}$$

Here we replicate this solution numerically, using fitted value function iteration.

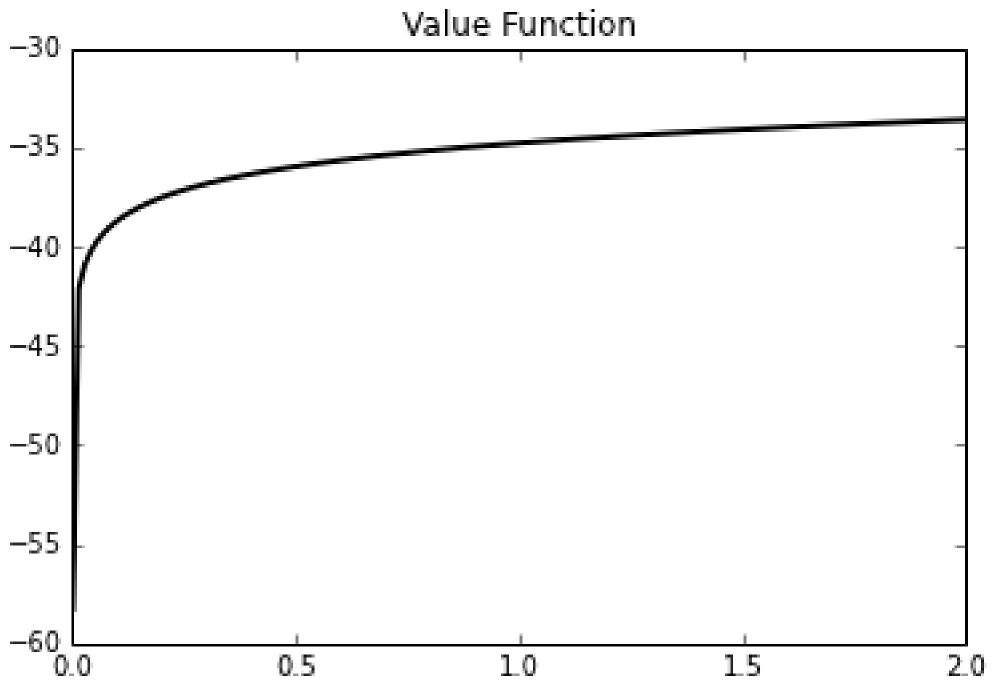
.. [RMT3]Recursive Macroeconomic Theory, 3rd edition, by Lars Ljungqvist and Thomas J. Sargent

In [51]:
```
def exact_solution(alpha,beta):
    """Return function, the exact solution."""
    ab = alpha * beta
    c1 = (math.log(1 - ab) + math.log(ab) * ab / (1 - ab)) / (1 - beta)
    c2 = alpha / (1 - ab)
    return lambda k: c1 + c2 * np.log(k)

#parameters and grid
prms01 = dict(alpha = 0.65,beta=0.95) #match params in SS
kgrid01 = np.linspace(1e-6, 2, 150) #match grid in SS

#plot the exact solution
v_star = exact_solution(**prms01)
fig, ax = plt.subplots(1,1)
ax.plot(kgrid01, v_star(kgrid01), 'k-', lw=2)
ax.set_title('Value Function')

fig.show()
```



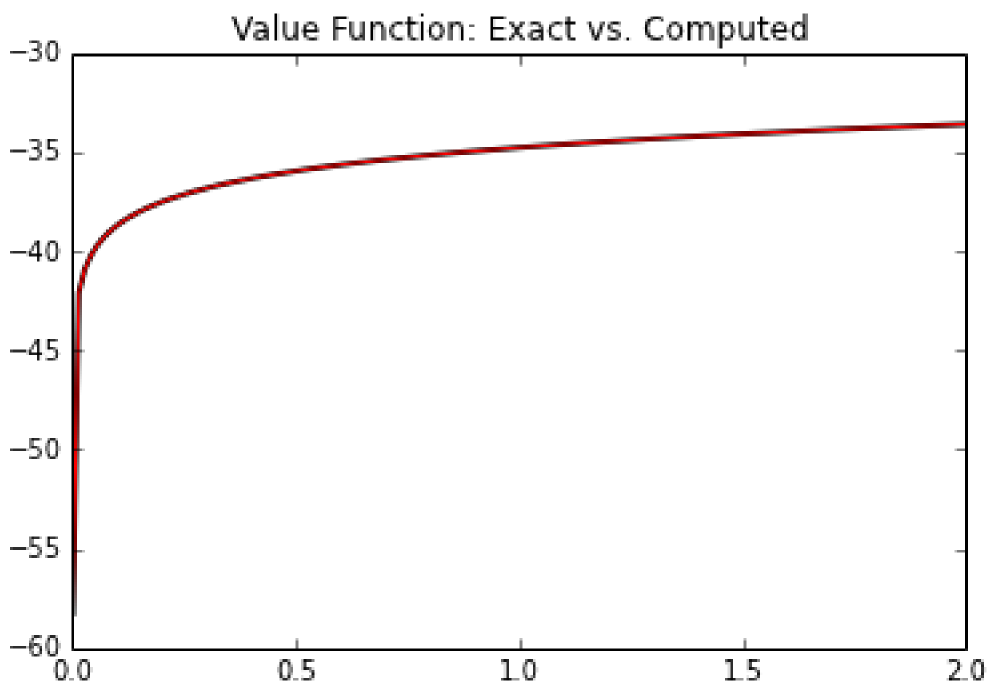
In [50]:
```
class GrowthModel01(object):
    def __init__(self, f, u, beta):
        self.f = f  #function of one variable, net prodn
        self.u = u  #function of one variable, utility
        self.beta = beta  #float in (0,1), the discount rate
    def vdirect(self, c, k, w):
        u, f, beta = self.u, self.f, self.beta
        return u(c) + beta * w(f(k)-c)
    def c(self, k, w):  # w-greedy consumption policy
        f2max = lambda c: self.vdirect(c, k, w)
        return fminbound(lambda c: -f2max(c), 1e-6, self.f(k))
    def vindirect(self, k, w):
        cstar = self.c(k, w)
        return self.vdirect(cstar, k, w)

gm01 = GrowthModel01(lambda k: k**prms01['alpha'], np.log, prms01['beta'])
vvals = [gm01.vindirect(k, v_star) for k in kgrid01]

#plot the exact solution
fig, ax = plt.subplots(1,1)
```

```
ax.plot(kgrid01, v_star(kgrid01), 'k-', lw=2)
ax.plot(kgrid01, vvals, 'r-', lw=1)
ax.set_title('Value Function: Exact vs. Computed')

fig.show()
```



3.6 Computing a Fitted Value Iteration

Problem: function iterates often can neither be calculated exactly nor stored on a computer. Even if our initial w is a known function, the new function Tw usually is not. So unless Tw has a very special structure, we apparently need to do the impossible: store $Tw(k)$ for every $k \in R_+$.

Pragmatic alternative: fitted value-function iteration stores the value of Tw only only at chosen grid points, $(k_1, \ldots, k_I)$.

Our algorithm becomes as follows. We choose a grid $(k_1, \ldots, k_I)$ and an initial guess $(w_1, \ldots, w_I)$, our initial values of the function w on the grid. Then until some stopping condition is satisfied,

1. Extend our function to the state space R+ by interpolating the points $(w_1, \ldots, w_I)$
2. For each grid point k_i, solve (9) for the interpolated function $\hat{w}$, producing $T\hat{w}(k_i)$
3. Set $(w_1, \ldots, w_I) = (T\hat{w}(k_1), \ldots, T\hat{w}(k_I))$ and return to step 1

We need a function approximation strategy for step 1. This must

- produce a good approximation to Tw
- combine well with the broader iteration algorithm described above

Our choice: continuous piecewise linear interpolation.

3.7 Piecewise linear interpolation

Piecewise linear interpolation preserves useful shape properties such as monotonicity and concavity (or convexity). (See http://quant-econ.net/_downloads/3ndp.pdf for a discussion).

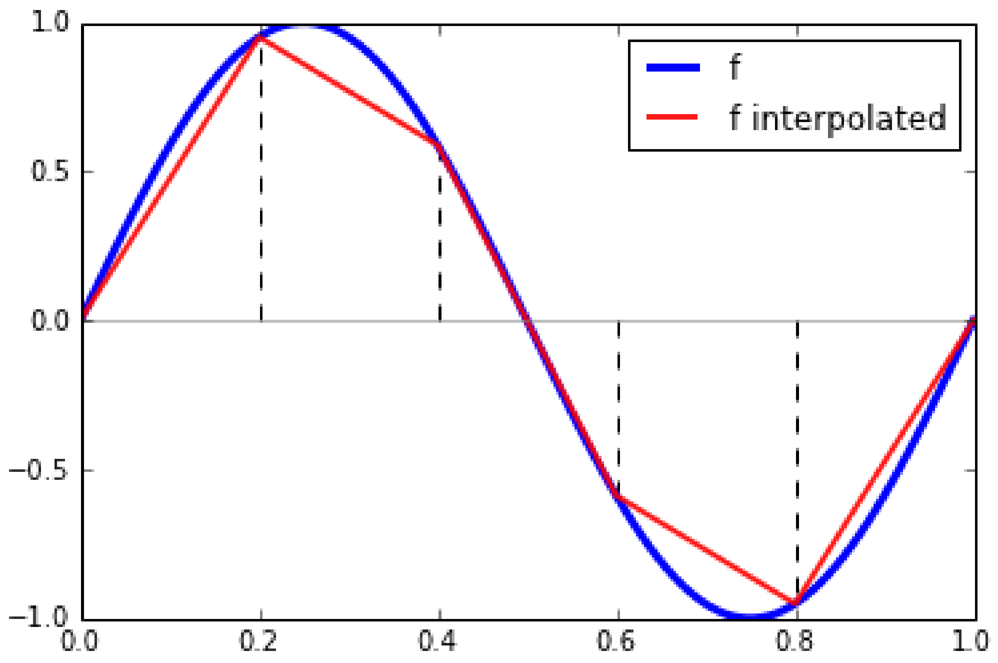
In [52]:
```
"""Illustrate piecewise linear interpolation using NumPy's linear interpolation functi
(See http://docs.scipy.org/doc/numpy/reference/generated/numpy.interp.html for documen
def f(x): return np.sin(math.pi * 2 * x)

grid = np.linspace(0, 1, 6)
def f_interp(x): return np.interp(x, grid, f(grid))

#illustrative chart
domain = np.linspace(0, 1, 201)
fig, ax = plt.subplots(1,1)
ax.plot(domain, f(domain), 'b-', lw=3, label='f')
ax.plot(domain, f_interp(domain), 'r-', lw=2, alpha=0.9, label='f interpolated')

#add some nice details to the chart
ax.axhline(lw=0.5, color='0.5')
ax.vlines(grid, np.zeros_like(grid), f(grid), linestyle='dashed')
ax.set_xlim(0, 1)
ax.legend(loc='upper right')

fig.show()
```



3.8 Bellman Operator on a Grid

Here we use piecewise linear interpolation to approximate the Bellman operator on a grid.

Implementation Goals

We would like to input a GrowthModel01 instance, which provides a description of a model (technology, preferences, etc.), and get the Bellman operator associated with that model.

While we cannot readily operate directly on functions, we can operate on functions defined on a finite grid. So in addition to the model description, we will need a specfication of the grid we will use.

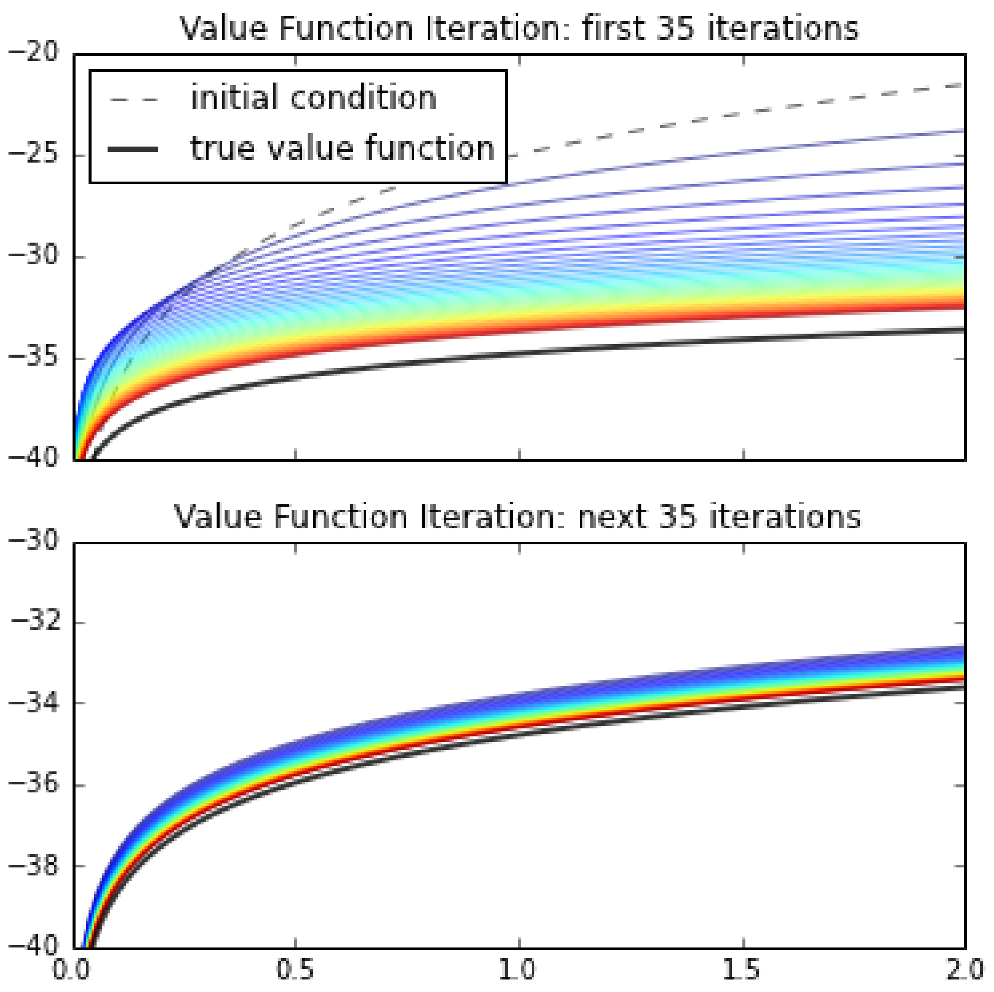
In [31]:
```
def get_bellman_operator(gm, grid):
    """Return function, the approximate Bellman operator."""
    def T(wvals):
        """Return array, the updated values on the grid,
        given `wvals`, the initial values on the grid."""
        w = lambda k: np.interp(k, grid, wvals) #SS's Aw
        Tw = np.empty_like(wvals)
        # set Tw[i] = max_c { u(c) + beta vnext(f(k_i) - c)}
        for i, k in enumerate(grid):
            Tw[i] = gm.vindirect(k, w)  #pleasingly parallel
        return Tw
    return T
```

Next we plot successive functions produced by our fitted value function iteration. Hotter colors represent higher iterates. The true value function v^* is the thick, black line. The sequence of iterates coverges towards v^*. Increasing the number of iterations produces further improvement.

SS comment: "knowledge of the functional form of v^* for this model has influenced our choice of the initial condition ... In more realistic problems such information is not available, and convergence will probably take longer."

In [34]:
```
"""
Illustrate a basic numerical solution of the optimal growth problem
via value function iteration.  (For more detail, see SS's optgrowth.py.)
"""
def plot_value_iterations(n, gm, grid, wvals):
    bellman_operator = get_bellman_operator(gm, grid)
    fig, (ax1,ax2) = plt.subplots(2,1,sharex=True, figsize=(6,6))
    ax1.plot(grid, wvals, 'k--', lw=1, alpha=0.6, label='initial condition')
    for i in range(n):
        wvals = bellman_operator(wvals)
        ax1.plot(grid, wvals, color=plt.cm.jet(i / n), lw=1, alpha=0.6)
    for i in range(n):
        wvals = bellman_operator(wvals)
        ax2.plot(grid, wvals, color=plt.cm.jet(i / n), lw=1, alpha=0.6)
    ax1.plot(grid, v_star(grid), 'k-', lw=2, alpha=0.8, label='true value function')
    ax2.plot(grid, v_star(grid), 'k-', lw=2, alpha=0.8, label='true value function')
    ax1.set_ylim(-40,-20)
    ax2.set_ylim(-40,-30)
    ax1.set_xlim(np.min(grid), np.max(grid))
    ax1.legend(loc='upper left')
    ax1.set_title('Value Function Iteration: first {} iterations'.format(n))
    ax2.set_title('Value Function Iteration: next {} iterations'.format(n))

if __name__ == '__main__':
    winit = 5 * np.log(kgrid01) - 25  #Take informed initial condition from SS
    plot_value_iterations(35, gm01, kgrid01, winit)
```



3.9 When to Stop

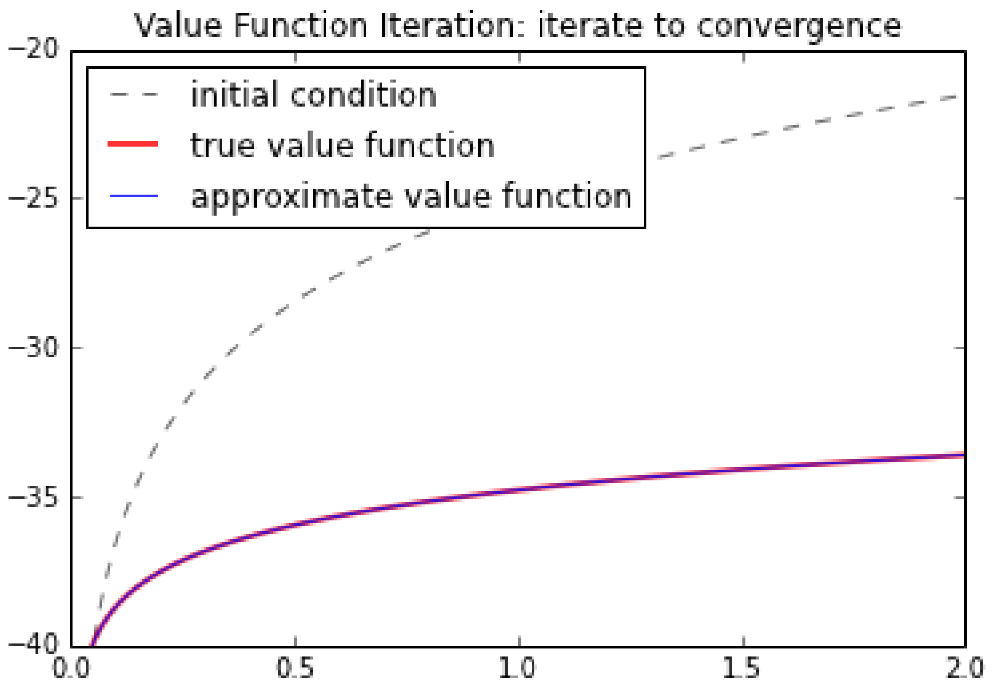
If we are going to compute an approximate value function, we have to decide when to stop. Iteration from an initial condition until stopping is a classic example of a recurrence relation. Here we give a simple functional representation.

In [42]:
```
def fixedpt1d(f, v, itermax=200, tol=1e-3):
    ct, error = 0, 1+tol  #initializations
    while ct < itermax and error > tol:
        ct += 1
        vnext = f(v)
        error = np.max(np.abs(vnext-v))
        v = vnext
    if ct==itermax:
        print 'convergence failed in {} iterations'.format(ct)
    else: print 'convergence in {} iterations'.format(ct)
    return v

def plot_value_approximation(gm, grid, wvals):
    bellman_operator = get_bellman_operator(gm, grid)
    fig, ax = plt.subplots(1,1)
    ax.plot(grid, wvals, 'k--', lw=1, alpha=0.6, label='initial condition')
    ax.plot(grid, v_star(grid), 'r-', lw=2, alpha=0.8, label='true value function')
    v = fixedpt1d(bellman_operator, wvals)
    ax.plot(grid, v, 'b-', lw=1, alpha=0.8, label='approximate value function')
    ax.set_ylim(-40,-20)
    ax.set_title('Value Function Iteration: iterate to convergence')
    ax.legend(loc='upper left')

if __name__ == '__main__':
    winit = 5 * np.log(kgrid01) - 25  #Take informed initial condition from SS
    plot_value_approximation(gm01, kgrid01, winit)
```

```
convergence in 161 iterations
```



4 The Policy Function

Our illustrative problem has an analytical solution for the optimal consumption policy:

$$\sigma(k) = (1 - \alpha\beta)k\alpha$$

This allows us to compare a numerical solution to the analytical solution.

Suppose we iterate the fitted value function until it convergences. This gives us an approximation to v^*. We can then use this to compute an approximate optimal policy as the greedy strategy associated implied by our approximation to v^*.The next figure compares the numerical solution to this exact solution

In the three figures, the approximation to v∗ is obtained by running the loop in the fitted value function algorithm 2, 4 and 6 times respectively

Even with as few as 6 iterates, the numerical result is quite close to the true policy

In [47]:
```
alpha, beta = prms01['alpha'], prms01['beta']
true_sigma = (1 - alpha * beta) * kgrid01**alpha
winit = 5 * gm01.u(kgrid01) - 25  # Initial condition
bellman_operator = get_bellman_operator(gm01, kgrid01)
```

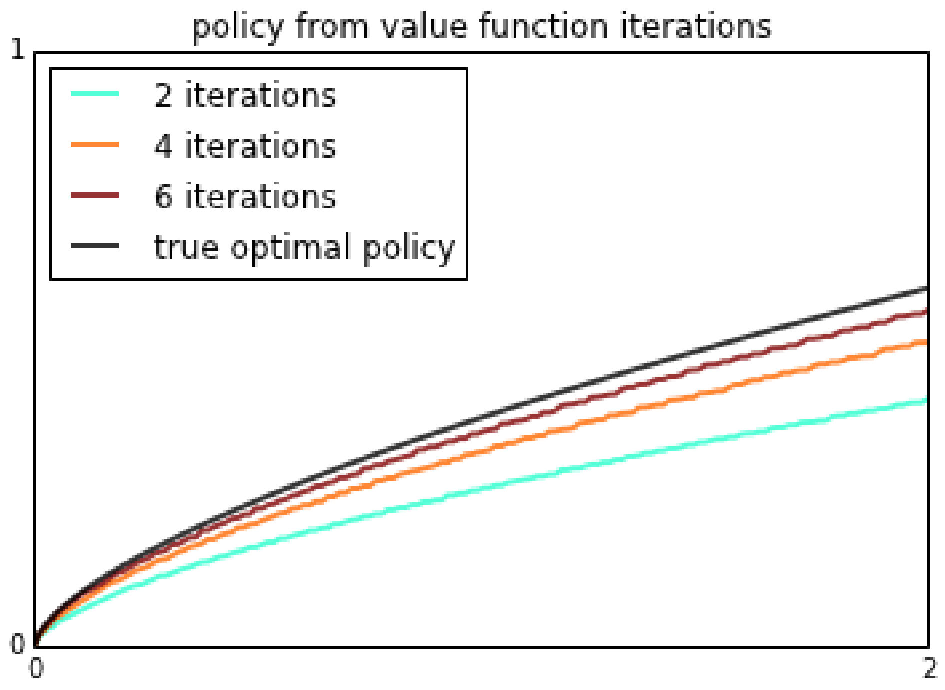
```
fig, ax = plt.subplots(1, 1)

for niter in (2, 4, 6):  #match SS illustrations
    vvals_approx = fixedpt1d(bellman_operator, winit, itermax=niter)
    vinterp = lambda k: np.interp(k, kgrid01, vvals_approx)
    sigma = [gm01.c(k, vinterp) for k in kgrid01]
    ax.plot(kgrid01, sigma, color=plt.cm.jet(niter / 5), lw=2, alpha=0.8, label='{} it

ax.plot(kgrid01, true_sigma, 'k-', lw=2, alpha=0.8, label='true optimal policy')
ax.legend(loc='upper left')
ax.set_title('policy from value function iterations')
ax.set_ylim(0, 1)
ax.set_xlim(0, 2)
ax.set_yticks((0, 1))
ax.set_xticks((0, 2))

plt.show()
```

```
convergence failed in 2 iterations
convergence failed in 4 iterations
convergence failed in 6 iterations
```



5 Exercises

5.1 SS Exercise 1

Replicate the optimal policy figure shown above

Use the same parameters and initial condition found in optgrowth.py

Solution: see SS

5.2 SS Exercise 2

Once an optimal consumption policy σ is given, the dynamics for the capital stock follow (5)

The next figure shows the first 25 elements of this sequence for three different discount factors (and hence three different policies)

*http://www.quant-econ.net/*images/solution_og_ex2.png

In each sequence, the initial condition is k0=0.1

The discount factors are discount_factors = (0.9, 0.94, 0.98)

Otherwise, the parameters and primitives are the same as found in optgrowth.py

Replicate the figure

Solution: See SS