Optimal Growth, Dynamic Programming

Kenji Sato

January 17, 2017

1 Purpose

In this note, we are going to solve the optimal growth model by a dynamic programming approach. The method developed here is going to solve the limitations discussed HERE

2 Model

We consider the following optimal growth model in the simplest form.

$$\begin{aligned}&\max \int_0^\infty e^{-\rho t} u(c) dt \\ &\text{subject to} \\ &\dot{k} = f(k) - \delta k - c, \\ &k(0) = k_0 : \text{ given},\end{aligned}$$

Here is a summary of the variables of interest and fundamental variables.

Parameters	Type	Interpretation
c	endogenous variable	consumption per capita
k	endogenous variable	capital per capita
f	parameter (function)	production function
u	parameter (function)	utility function
k_0	parameter	initial condition
$\delta > 0$	parameter	depreciation rate
$\rho > 0$	parameter	discount rate

3 Assumption

We are going to make simplifying assumptions.

3.1 Cobb–Douglas production

We assume

$$f(k) = k^\alpha, \qquad 0 < \alpha < 1.$$

3.2 CRRA utility function

We assume

$$u(c) = \frac{c^{1-\theta} - 1}{1-\theta}, \quad \theta > 0, \quad \theta \neq 1$$

Notice that

$$\theta = \frac{cu'(c)}{-u''(c)},$$

the right-hand side of which is called the coefficient of relative risk aversion. CRRA function has a **c**onstant coefficient of **r**elative **r**isk **a**version.

The above function form is not well defined for $\theta = 1$. When $\theta = 1$ we define

$$u(c) = \ln c$$

Verify that the coefficient of relative risk aversion is 1 for this case.

3.3 Summary

Under our assumptions, we can update the summary table.

Parameters	Type	Interpretation
c	endogenous variable	consumption per capita
k	endogenous variable	capital per capita
$\alpha \in (0,1)$	parameter	capital share
$\theta > 0$	parameter	coefficient of relative risk aversion
k_0	parameter	initial condition
$\delta > 0$	parameter	depreciation rate
$\rho > 0$	parameter	discount rate

4 Dynamic Programming

By the method of dynamic programing we know that the value that the objective function attains along the optimal path is a function of initial $k(0) = x$. This function is denoted by

$$v(x) = \int_0^\infty e^{-\rho t} u(c) dt,$$

where the right-hand side is computed along the optimal path. Function v, called the value function, satisfies the following Hamilton-Jacobi-Bellman equation:

$$\rho v(x) = \max_c \left[u(c) + v'(x) \left\{ f(x) - \delta x - c \right\} \right], \quad x \geq 0.$$

Since the utility maximizing consumption is attained at

$$c_{\text{opt}}(x) = (u')^{-1}(v'(x)),$$

having max in the equation doesn't make things hard. ($(u')^{-1}(\cdot)$ is the inverse function of indreasing $u'(\cdot)$).

Here is a trick to compute v iteratively. Consider the following update rule:

$$\frac{v_1(x) - v_0(x)}{\Delta} + \rho v_0(x) = \max_c \left[u(c) + v_0'(x)\left\{f(x) - \delta x - c\right\}\right],$$

where v_0 is the initial guess of the value and v_1 is the updated guess:

$$\begin{aligned} v_1(x) &= v_0(x) + \Delta\left(-\rho v_0(x) + \max_c \left[u(c) + v_0'(x)\left\{f(x) - \delta x - c\right\}\right]\right) \\ &= v_0(x) + \Delta\left[-\rho v_0(x) + u\left(c_{\text{opt}}(x)\right) + v_0'(x)\left\{f(x) - \delta x - c_{\text{opt}}(x)\right\}\right] \\ &= (1 - \rho\Delta)v_0(x) + \Delta\left[u\left(c_{\text{opt}}(x)\right) + v_0'(x)\left\{f(x) - \delta x - c_{\text{opt}}(x)\right\}\right] \end{aligned}$$

If you continue this procedure of updating the guess from v_0 to v_1 and to $v_2, \ldots$, until v_n and v_{n+1} are sufficiently close to each other (Technically speaking this is not sufficient for convergence), you will get a good approximation of the true value function v.

I will demonstrate this solution method here. You don't have to fully understand the theory at this stage. I adapted the code (for this explicit method) by B. Moll http://www.princeton.edu/~moll/HACTproject/HACT_Additional_Codes.pdf

5 Computation

5.1 Setup

We are going to use the following R packages.

```
library(ggplot2)
library(ggthemes)
library(tibble)
library(dplyr)
```

Here's the parameters for the model:

```
# Parameters
alpha = 0.3
theta = 5
delta = 0.05
rho = 0.1

# Functions
f = function(k){
  return(k^alpha)
}

u = function(c){
  return((c^(1 - theta) - 1) / (1 - theta))
}

# dk/dt = 0 locus
c_steady = function(k){
  return(f(k) - delta * k)
}
```

Note that the inverse function, $(f')^{-1}$, of $f'(k) = \alpha k^{\alpha-1}$ is given by

$$(f')^{-1}(r) = \left(\frac{\alpha}{r}\right)^{\frac{1}{1-\alpha}}$$

The inverse function of u' is given by

$$(u')^{-1}(y) = y^{-1/\theta}$$

```
# u'
du = function(c){
  return(c ^ (-theta))
}

# Inverse of u'
inv_du = function(y){
  return(y ^ (-1 / theta))
}

# f'
df = function(k){
  return(alpha * k ^ (alpha - 1))
}

# Inverse of f'
inv_df = function(r){
  return((alpha / (delta + rho)) ^ (1 / (1 - alpha)))
}
```

We need to specify the range of k we work on.

```
ngrid = 250
k = seq(0.01, 10, length.out=ngrid)
dk = k[2] - k[1]
```

The following code is an implementation of the above update rule. For the detailed discussion, see Moll's note.

```
Delta = 0.25 * dk / max(f(k) - delta * k)

update_explicit = function(v){
  # Update rule for the explicit method
  #
  # Parameter
  # ---------
  # v: numeric vector
  #     Guess of the value function
  #
  # Return
  # ------
  #
  # vnew: numeric vector
  #     Updated guess of the value function
  #
  # dv: numeric vector
  #     Numerical derivative of v
  #
```

```
  n = length(v)
  vnew = numeric(n)
  dv = numeric(n)

  # forward difference
  dvf = numeric(n)
  dvf[1: n - 1] = (v[2: n] - v[1: n - 1]) / dk

  # backward difference
  dvb = numeric(n)
  dvb[2: n] = (v[2: n] - v[1: n - 1]) / dk
  dvb[1] = du(f(k[1])) - delta * k[1]

  muf = f(k) - delta * k - inv_du(dvf)
  mub = f(k) - delta * k - inv_du(dvb)

  dv = ifelse(muf > 0,
              dvf,
              ifelse(mub < 0,
                     dvb,
                     ifelse(muf < 0 & mub > 0,
                            du(f(k) - delta * k),
                            0))
              )

  c = inv_du(dv)
  vnew = (1 - rho * Delta) * v +
            Delta * (u(c) + dv * (f(k) - delta * k - c))
  return(list(v=vnew, dv=dv))
}
```

The value function is attained after several application of this function.

```
num_iter = 20000
v0 = u(k)

for (i in 1:num_iter){
  v_dv = update_explicit(v0)
  v = v_dv$v
  dv = v_dv$dv
  if (isTRUE(all.equal(v, v0))) break
  v0 = v
}
print(i)
```

```
## [1] 14204
```

Since you have the value function and its derivative now, you can compute the policy function (or the optimal consumption function) by

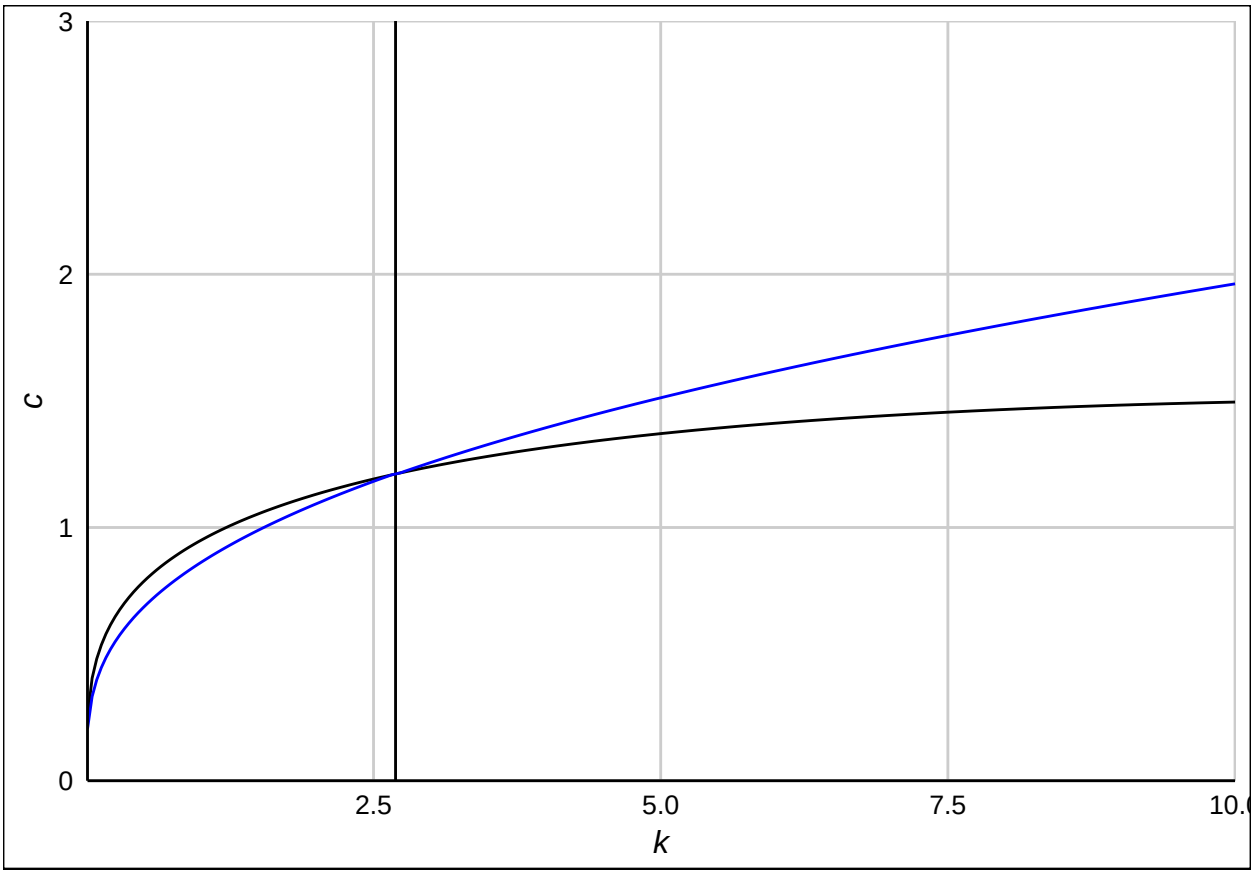
$$c(k) = (u')^{-1}(v'(k))$$

```
# The modified golden rule capital stock
k_mg = ((delta + rho) / alpha) ** (1/(alpha - 1))
```

```
data = tibble(k = k) %>%
  mutate(locus = f(k) - delta * k,
         c = inv_du(dv))

# Plot area
p = ggplot(data) + theme_gdocs() + labs(x='k', y='c') +
  scale_x_continuous(expand=c(0,0)) +
  scale_y_continuous(expand=c(0,0), limits=c(0, 3)) +
  geom_line(aes(x=k, y=locus)) +
  geom_vline(xintercept=k_mg)

# Plot the consumption function
p + geom_line(aes(x=k, y=c), color='blue')
```



6 Conclusion

With the dynamic programming approach, we can compute the policy function more efficiently. The explicit method discussed here has an issue of convergence and so we carefully choose Δ and the grid size for k.

It is desirable to compare with an implicit method for solving the Hamilton-Jacobi-Bellman equation, which is beyond our scope.