Optimal Growth, Euler Equation

Kenji Sato

January 18, 2017

1 Purpose

In this note, we are going to solve the optimal growth model by simulating the first-order dynamics. The limitation of this method is also discussed briefly.

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 Dynamics characterized by the first-order conditions

The model can be solved by various methods and we applied a technique developed in the optimal control theory. The dynamics is given by

$$\dot{k} = f(k) - \delta k - c$$
$$\dot{c} = \frac{c(f'(k) - \delta - \rho)}{\theta}$$

A path that satisfies the above differential equation is optimal if it satisfies the transversality condition as well:

$$\lim_{t\to\infty} e^{-\rho t} u'(c) k = 0$$

Typically, a path that converges to the steady state is the optimal path. The steady state is characterized by the following two static equations.

$$c = f(k) - \delta k$$
$$f(k) = \delta + \rho$$

The (unique) steady state values for k and c are denoted by k^* and c^*, respectively.

5 Computation

5.1 Setup

We are going to use the following R packages.

```r
library(ggplot2)
library(ggthemes)
library(tibble)
```

Here's the parameters for the model:

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

k0 = 1.0

# 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

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

```r
# u': Not used

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

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

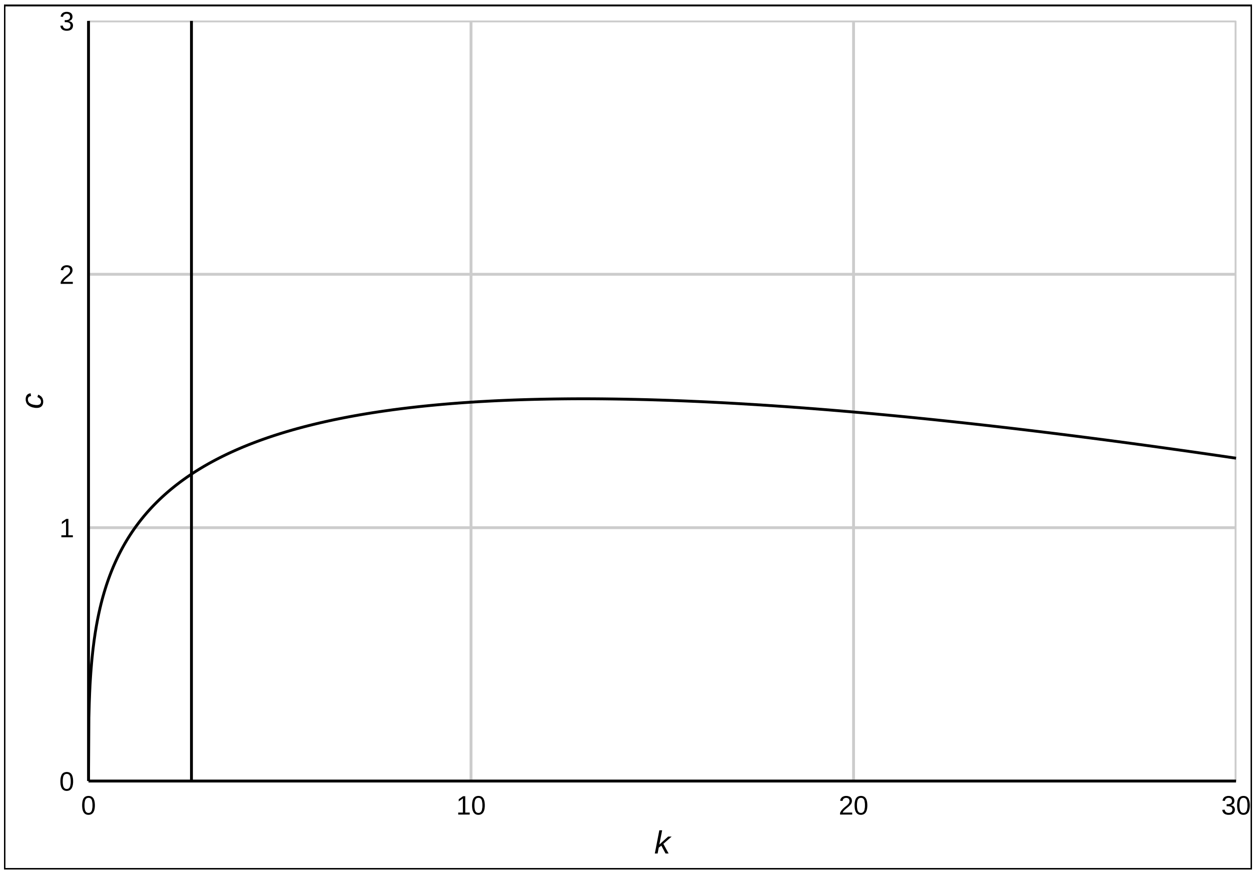
We can then solve the steady state k^*:

```
k_star = inv_df(delta + rho)
k_star
```

```
## [1] 2.6918
```

Define the parameters for the plot area.

```
kmin = 0.0; kmax = 30.0
cmin = 0.0; cmax = 3.0

kgrid = seq(kmin, kmax, by=0.01)

p = qplot(kgrid, c_steady(kgrid), geom='line') +
  geom_vline(xintercept=k_star) + theme_gdocs() +
  labs(x='k', y='c') +
  scale_x_continuous(expand=c(0,0)) +
  scale_y_continuous(expand=c(0,0), limits=c(cmin, cmax))

p
```



5.2 Simulation Code

Now we discretize and code the above differential equations.

$$
\begin{aligned}
\frac{k(t+\Delta t)-k(t)}{\Delta t} &= f(k(t)) - \delta k(t) - c(t) \\
\frac{c(t+\Delta t)-c(t)}{\Delta t} &= \frac{c(t)\left[f'(k(t)-\delta-\rho)\right]}{\theta}
\end{aligned}
$$

or, equivalently,

$$
\begin{aligned}
k(t+\Delta t) &= k(t) + \Delta t\left[f(k(t)) - \delta k(t) - c(t)\right] \\
c(t+\Delta t) &= c(t)\left[\,]1 + \Delta t\frac{f'(k(t)) - \delta - \rho}{\theta}\right].
\end{aligned}
$$

Let's define R functions for this update rule.

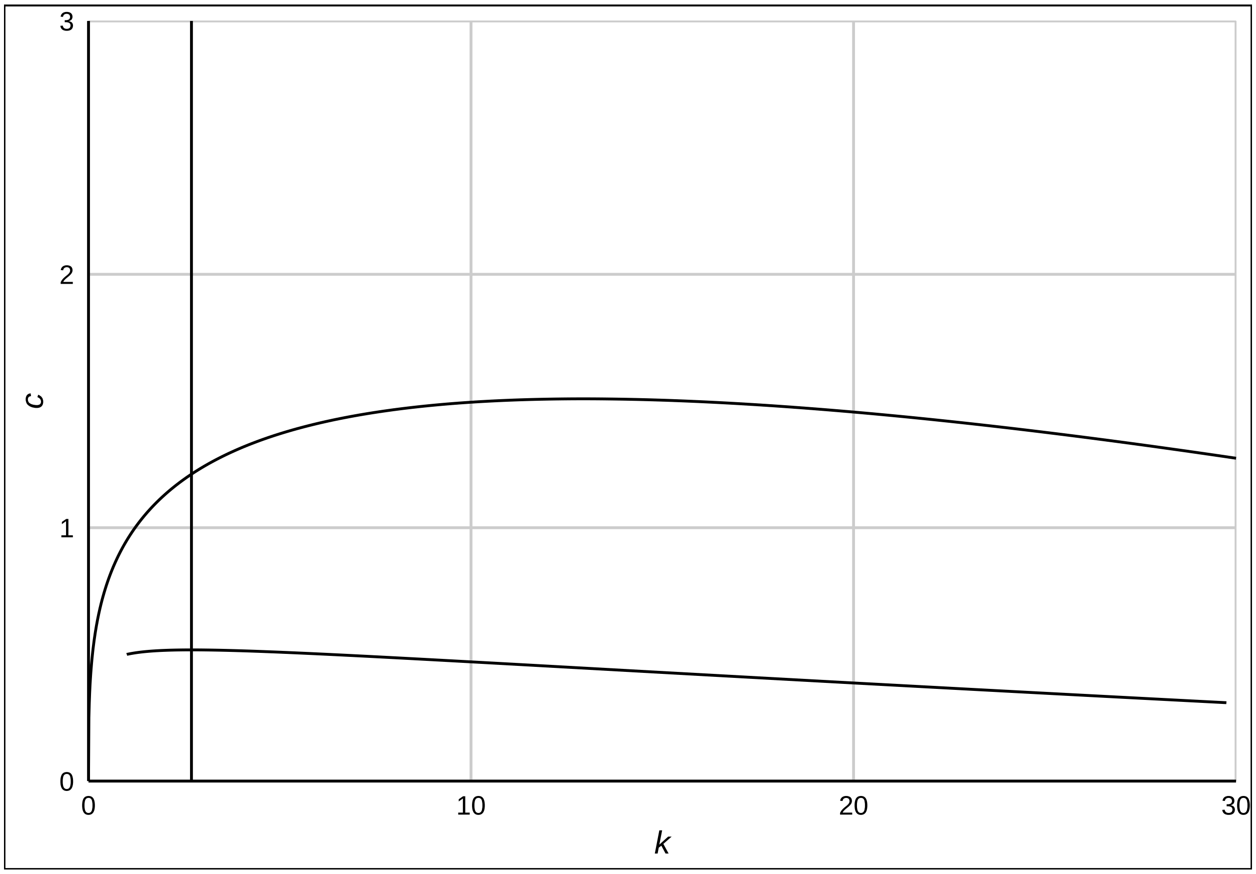
```
update_k = function(k, c, dt=0.01){
  return(k + dt * (f(k) - delta * k - c))
}

update_c = function(k, c, dt=0.01){
  return(c * (1 + dt * (df(k) - delta - rho) / theta))
}
```

To run the simulation, we have to specify a guess about $c(0)$ and the terminal condition for the simulation.

```
c0 = 0.5     # Guess for the non-predetermined
steps = 3000 # number of steps (terminal condition)

simulation = tibble(k=numeric(steps), c=numeric(steps))

for (i in seq_len(steps)){
  if (i == 1){
    # Initialize
    simulation[1,'k'] = k0   # given
    simulation[1, 'c'] = c0 # guess
  } else {
    simulation[i, 'k'] = update_k(simulation[i - 1, 'k'],
                                  simulation[i - 1, 'c'])
    simulation[i, 'c'] = update_c(simulation[i - 1, 'k'],
                                  simulation[i - 1, 'c'])
  }
}
```

To see the result, plot the simulation data over the plot area that we have created above.

```
p + geom_path(data=simulation, aes(x=k, y=c))
```



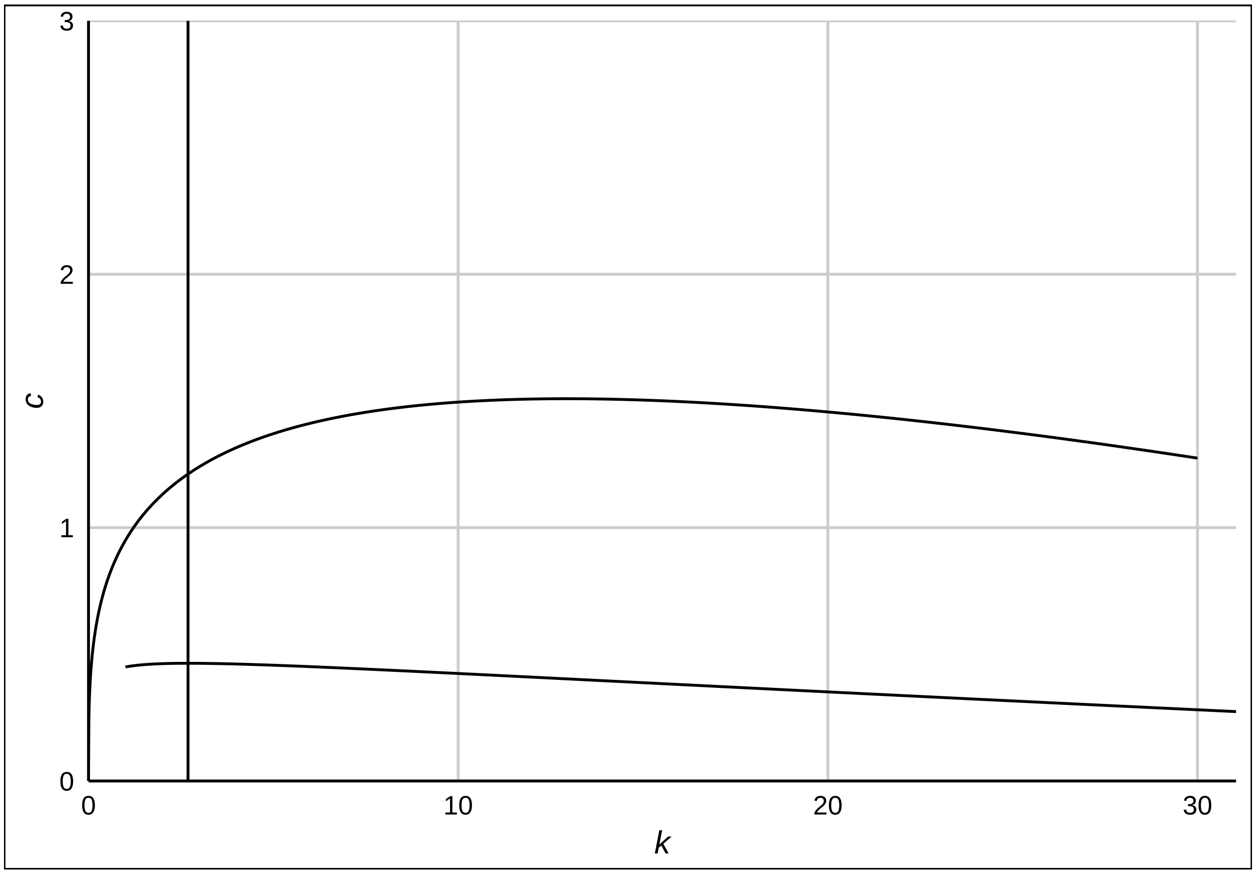
5.3 Automation

Let's define a function that performs the above simulation.

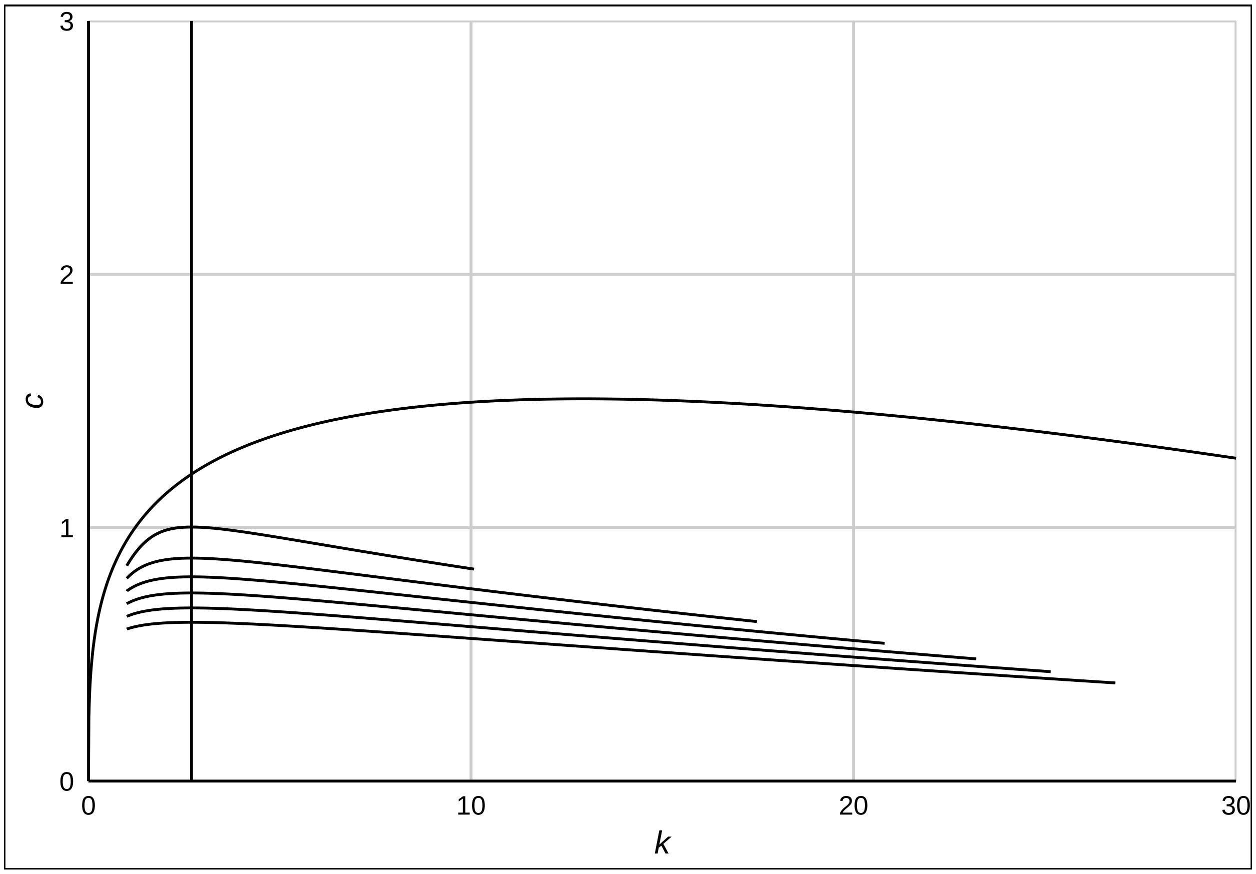
```
simulate = function(k0, c0, steps, dt=0.01){
  simulation = tibble(k=numeric(steps), c=numeric(steps))

  for (i in seq_len(steps)){
    if (i == 1){
      # Initialize
      simulation[1,'k'] = k0    # given
      simulation[1, 'c'] = c0 # guess
    } else {
      k_new = update_k(simulation[i - 1, 'k'], simulation[i - 1, 'c'], dt)
      if (k_new < 0) break

      simulation[i, 'k'] = k_new
      simulation[i, 'c'] = update_c(simulation[i - 1, 'k'],
                                    simulation[i - 1, 'c'], dt)
    }
  }
  return(simulation)
}
```

The new simulation code looks like:

```
simulation = simulate(k0=k0, c0=0.45, steps=3000)
p + geom_path(data=simulation, aes(x=k, y=c))
```

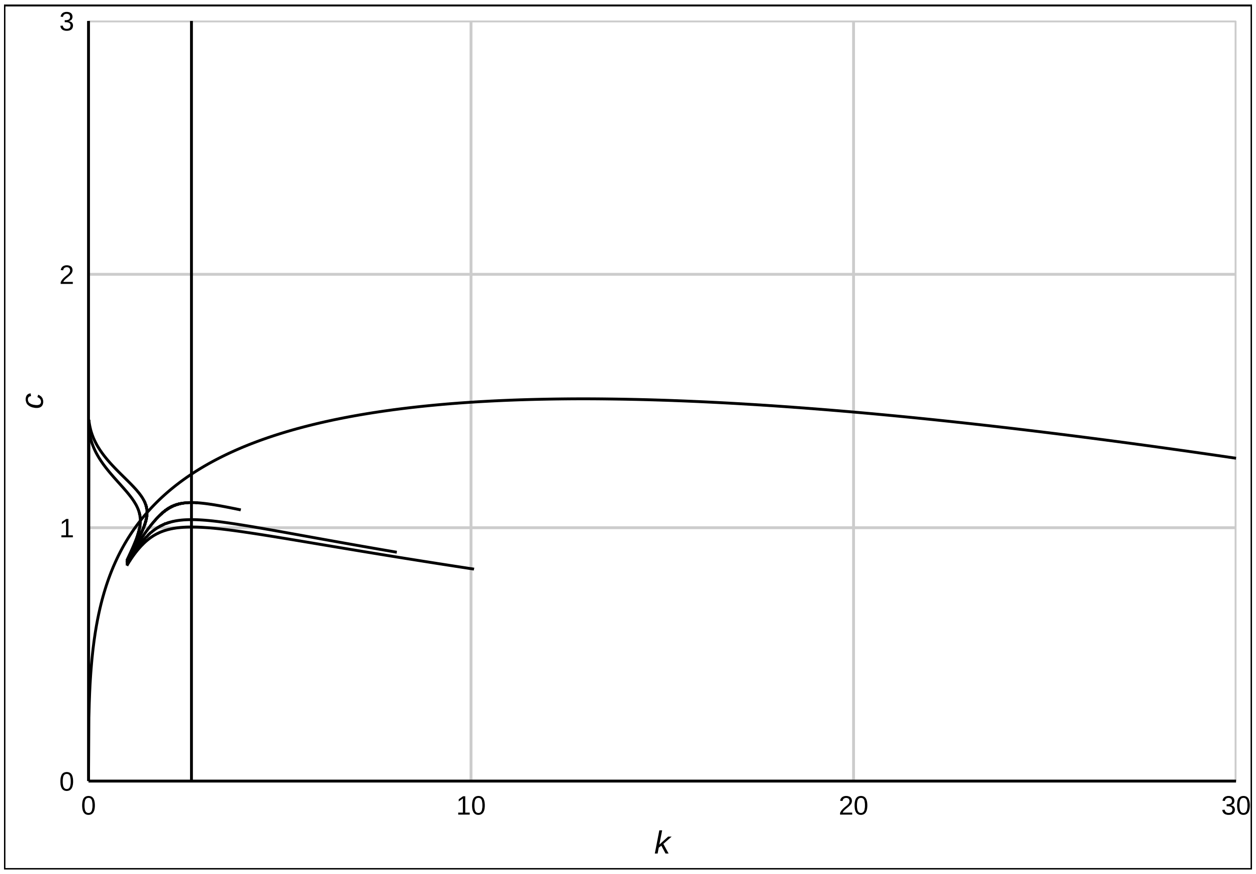


```
p_below = p

c0 = c(.6, .65, .7, .75, .8, .85)
for (i in seq_along(c0)){
  simulation = simulate(k0=k0, c0=c0[i], steps=3000)
  p_below = p_below + geom_path(data=simulation, aes(x=k, y=c))
}

p_below
```



Do you see that the development of the economy gets slower as c_0 becomes larger. This is a good sign of approaching to the steady state because at the steady state, the economy moves slowest (no movement).

```
p_above = p

c0 = c(.85, .855, .86, .865, .87)
for (i in seq_along(c0)){
  simulation = simulate(k0=k0, c0=c0[i], steps=3000)
  p_above = p_above + geom_path(data=simulation, aes(x=k, y=c))
}

p_above
```



5.4 Further automation: Binary search for the optimal consumption

It seems like there is the optimal c_0 between .860 and .865.

Let's find the optimal `c0` (or its approximation). In the following `search_policy` function, we will exploit several facts about the optimal path and non-optimal paths.

- Along a non-optimal path, either `c` or `k` is not monotone.
  - We assume that our initial guess lies on the appropriate region; in the north-east or south-west of the steady state.
  - If `c` starts to decrease, then the initial guess for `c0` was too small. Set bigger `c0` and run another simulation.
  - If `k` starts to decrease, then the initial guess for `c0` was too large. Set smaller `c0` and run another simulation.
- Along the optimal path, both `c` and `k` are monotone (increasing or descreasing).
  - If none of them don't decrease for sufficient long steps, we understand that we have chosen a correct guess. This step count should be a function argument.

Because we need to check monotonicity, we can't reuse the `simulation` function above. Such a lack of foresight!

```
search_policy = function(k0, min_c0, max_c0, max_step,
                         dt=0.01, max_iter=1e4){
  # ----------------------------------------------------------------------
  # Find the optimal consumption policy for a given initial capital stock k0
  # ----------------------------------------------------------------------
  #
```

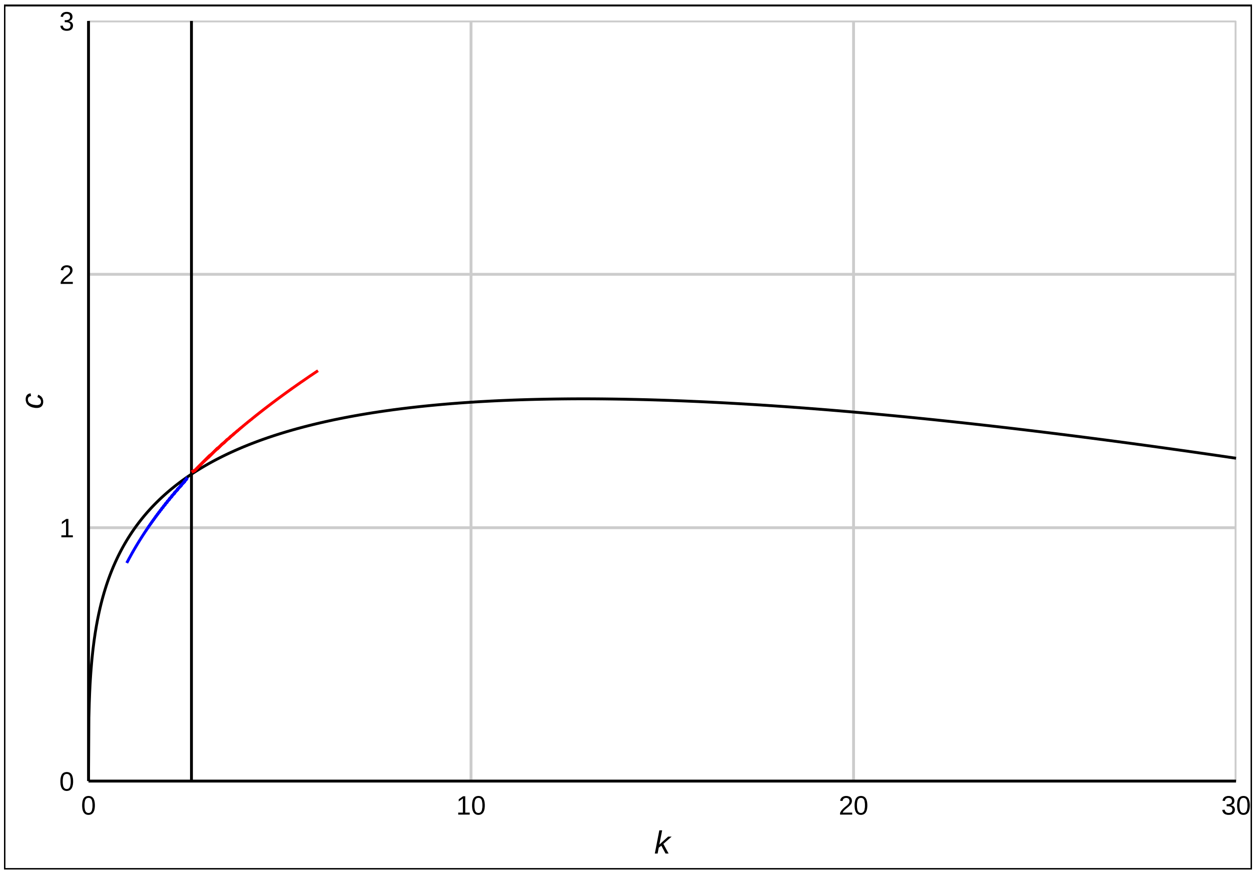
```r
# Parameters
# ===========
# k0: numeric
#     initial capital stock
#
# min_c0: numeric
# max_c0: numeric
#     The interval [cmin, cmax] is the range of guess which the analyst
#     (you) believes that the optimal c0 lies between.
#
# max_step: integer
#     The max step count for which monotonicity is verified.
#     If a simulation maintains monotonicity for sufficiently long,
#     it is understood that this path is very close to optimal.
#
# dt: numeric
#     Step size of simulation passed to `update_c()` and `update_k()`.
#
# max_iter: integer
#     Iteration limit.
#
# Returns
# =======
#
# optimal_path: tibble
#     optimal path that starts from (k0, c0)
#
# Example
# ========
#
# To be given...
#
#---------------------------------------------------------------------------

counter = 0
path = as.matrix(tibble(k=numeric(max_step), c=numeric(max_step)))

while (TRUE && counter < max_iter) {
  counter = counter + 1

  c0 = (min_c0 + max_c0) / 2

  path[1, 'k'] = k0
  path[1, 'c'] = c0

  for (i in seq_len(max_step - 1)) {
    path[i + 1, 'k'] = update_k(path[i, 'k'], path[i, 'c'], dt)
    path[i + 1, 'c'] = update_c(path[i, 'k'], path[i, 'c'], dt)
  }
  valid_nrow = min(sum(!is.na(path[,'k'])), sum(!is.na(path[,'c'])))
  valid_path = path[1:valid_nrow, ]

  k_increasing = all(valid_path[2:valid_nrow, 'k'] -
                          valid_path[1:valid_nrow - 1, 'k'] >= 0)
    k_decreasing = all(valid_path[2:valid_nrow, 'k'] -
                          valid_path[1:valid_nrow - 1, 'k'] <= 0)
    c_increasing = all(valid_path[2:valid_nrow, 'c'] -
                          valid_path[1:valid_nrow - 1, 'c'] >= 0)
    c_decreasing = all(valid_path[2:valid_nrow, 'c'] -
                          valid_path[1:valid_nrow - 1, 'c'] <= 0)

    increasing = k_increasing && c_increasing
    decreasing = k_decreasing && c_increasing

    if (increasing || decreasing) {
      break
    } else if (k_increasing && !c_increasing) {
      # Too small c0
      min_c0 = c0
    } else if (k_decreasing && !c_decreasing) {
      # Too large c0
      max_c0 = c0
    } else if (k_increasing && c_decreasing) {
      # Too small c0
      min_c0 = c0
    } else if (k_decreasing && c_increasing) {
      # Too large c0
      max_c0 = c0
    } else if (c_increasing && !k_increasing) {
      # Too large c0
      max_c0 = c0
    } else if (c_decreasing && !k_decreasing) {
      # Too small c0
      min_c0 = c0
    }
  }

  optimal_path = as_tibble(valid_path)
  return(optimal_path)
}
```

A simulation from the south west.

```r
optimum = search_policy(k0=k0, min_c0=.860, max_c0=.865, max_step=5000)
p_optimum = p + geom_path(data=optimum, aes(x=k, y=c), color="blue")
```

One from the north east.

```r
optimum_from_above = search_policy(k0=6, min_c0=1.619, max_c0=1.621,
                                   max_step=7000, max_iter=2000)
p_optimum + geom_path(data=optimum_from_above, aes(x=k, y=c), color="red")
```



6 Limitation

Note that the above code takes long time to find the optimal policy and we need to narrow down the initial guesses sufficiently. This is due to the fundamental instability of the optimal path.

In practice, we use a method based on Dynacmic Programming, which will be discussed elsewhere.