hw05: Simulation

Kenji Sato[*]

21 April, 2017

1 Overview

Purpose

In this note, you will learn how to write a simulation code. As will be discussed later, the most important components of simulation are

- update rule, and
- repetitive application of it.

You will learn how to elegantly define the update rule by using functional programming scheme and how to do the latter with `for` loop.

As an assignment, you are asked to write code to simulate linear systems.

Instructions

In this assignment, you will

- clone the assignment repository and make a working branch (eg. `solution` branch);
- solve the problems in Section 6;
  - **write the solution in `R/solution.R`**;
- knit `solution.Rmd`, which reads `R/solution.R`;
- commit `solution.Rmd` and `solution.pdf`; and
- open a Pull Request.

[*]Kobe University. Email: mail@kenjisato.jp

2 On Simulation

Consider the following dynamical system:

$$x_{t+1} = G(x_t, u_t), \quad t = 0, 1, \ldots$$

where all components in x_0 are given. This equation is sometimes called a "state equation." A set of variables is called the **state** if the dynamics after time t is fully determined by x_t and the exogenous inputs $u_t, u_{t+1}, \ldots$ Here, we interpret x as the output of the system. We desire to know the input-output relationship $u \mapsto x$.[1]

Regarding the behavior of the system, we may want to answer the following questions.

- What happens when an impulsive shock ($u_0 = 1, u_1 = 0, u_2 = 0, \ldots$) is input to the system?
  - Which directions does each of the output variables move?
- What happens when bounded shocks are continuously input?
  - Do we observe bounded output?

You can, at least partially, answer the above questions by writing simulation code and observing the simulated behavior of the system. Simulation is beneficial even when you work on a purely theoretical study:

- You can complement your understanding on the underlying economic models by running simulation.
- A theoretical conjecture that contradicts with a simulation exercise would certainly be false (assuming that the code is correct).

Since "simulation" means step-by-step computation of $x_1, x_2, \ldots$ starting from x_0, the most important building blocks in simulation, or dynamic systems in general, are

- The function that transforms old values into new values.
- The recursive application of this function starting from the initial conditions for fixed or indefinite periods of time.

Before diving into the dynamic model, it's beneficial to learn basics of **functional programming**, which is discussed in Section 3. Then we move on to the study of iteration in Section 4. In Section 5, you will learn how to write simulation code for autonomous linear systems in R. (Autnonomous in the sense that there is no input.)

[1] Think about the following question: How does the GDP change (let x denote the deviation) if the central bank raises the interest rate by u?

3 Functional Programming

R is a functional programming language, in which a function is a first class object. That is, R allows us to define

- functions that take functions as parameter;
- functions that return a function; and
- functions that transform a function into another function.

Let's see examples.

3.1 Function that takes a function as a paramter

`apply()` and its siblings are very frequently used functions that take a function as a parameter.

```
(A <- matrix(1:9, nrow = 3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
apply(A, 1, sum)
```

```
## [1] 12 15 18
```

In this example, `sum` is a function and it is passed to `apply()` as the third argument. Notice that it must be `sum` not `sum()`.

3.2 Function that returns a function

Let's define such a function that returns another function. All economic students are fond of Cobb-Douglas functions. Here is its per-capita form:

$$f(k) = k^{\alpha}.$$

The simplest way to defined this mathematical concept as a R function may be something like

```
alpha <- 0.3
f <- function(k) k ^ alpha
f(3)
```

```
## [1] 1.390389
```

The above code would do but I suggest to write it differently.

```r
cobb_douglas <- function(alpha) {
  f <- function(k) k ^ alpha
  # return(f)  ## R implicitly returns f
}
f <- cobb_douglas(alpha = 0.3)
f(3)
```

```
## [1] 1.390389
```

The `cobb_douglas()` takes `alpha` as a parameter and returns the production function as a R function. In the latter code, you don't require global parameter `alpha`, which is considered to be a good practice.

3.3 Function that takes and returns functions

Recall that the reduced-form utility function for a one-sector model is defined by

$$u(x, y) = U(f(x) - y)$$

Did you notice that u takes two parameters U, utility function, and f, production function? If you did, you should be able to write a clean code as below:

```r
reduced_utility <- function(U, f) {
  u <- function(x, y) U(f(x) - y)
}

u_log <- reduced_utility(log, cobb_douglas(0.3))
u_log(3, 1)
```

```
## [1] -0.9406112
```

3.4 Closure

This might be an advanced topic and you can probably skip the present section with no harm.

A function that returns a function can have internal states. As an example, let's code the one-dimensional linear dynamical system, or AR(1) model.

$$x_{t+1} = ax_t + u_t$$

```r
linsys_1d <- function(a, x) {
  # AR(1) model
  # a: coefficient
  # x: initial value

  state <- x
  update_rule <- function(u) {
    out <- state
    state <<- a * state + u
    return(out)
  }
  return(update_rule)
}

update <- linsys_1d(a = 0.8, x = 0.2)
```

This function differs from `cobb_douglas` in one respect; it has a local variable named `state`. A local variable defined within a higher order function is called "closure." Usually, a local variable within a function is trashed after the function call finishes because there exists no reference to that variable. Closures, in contrast, are persistent. As long as a reference to the returned function (`update` in our case) exists, the reference to the closure remains intact. Note that the assignment to `state` in the inner function `update_rule` uses the special assignment operator `<<-`, which assigns the right-hand side to the variables that was defined outside the inner function.

We can use this function to simulate an AR(1) model.

```r
update(u = 1)
```

```
## [1] 0.2
```

```r
update(u = 0)
```

```
## [1] 1.16
```

```r
update(u = 0)
```

```
## [1] 0.928
```

```r
update(u = 0)
```

```
## [1] 0.7424
```

Every time `update()` is invoked, the internal state, `state`, updates. This is why, `update()` returns different values each time although there is no randomness involved.

3.5 Summary

Functional programming is a very powerful tool and so you should start using it now. A section of Hadley Wickhams's *Advanced R* has more examples on this topic (http://adv-r.had.co.nz/Functional-programming.html).

4 Loops

Let's consider a problem of recursively multiplying 0.8 starting from 100. That is, we want to compute $x_0 = 100$, $x_1 = 0.8 \times x_0, \ldots, x_n = 0.8 \times x_{n-1}$ up until $n = N$. If N is small, say 3, the following code might suffice.

```
# Bad Code

x0 <- 100
x1 <- 0.8 * x0
x2 <- 0.8 * x1
x3 <- 0.8 * x2
x <- c(x0, x1, x2, x3)
x
```

```
## [1] 100.0  80.0  64.0  51.2
```

What if $N = 10000000$?

4.1 `for`

The above code example is bad even if N is small. Use `for` loop instead as in

```
# Good Code

x0 <- 100
N <- 3  # 1.
x <- numeric(N + 1)  # 2.

for (n in 1:(N + 1)) {   # 3.
  if (n == 1) {
    x[n] <- x0           # 4.
  } else {
    x[n] <- 0.8 * x[n - 1]  # 5.
  }
}
x
```

```
## [1] 100.0  80.0  64.0  51.2
```

1. `N` is the number of iteration.
2. `x` is an atomic vector of zeros with length `N + 1` (plus one for the initial value). This vector is used as a data store. You are advised to prepare a vector or matrix to store whole outcomes. See Circle 2 of Patrick Burns (2011) *R Infeno*.
3. The code between `{` and `}` is executed for `N + 1` times. `i` is incrementally increases as the loop continues.
4. `x[1]` is set to x_0, the initial value.
5. `x[n]` is set to `0.8 * x[n - 1]`, which naturally corresponds to the equation $x_n = 0.8 \times x_{n-1}$.

What's the benefit of "Good Code" over "Bad Code"? The "Good Code" is much easier to maintain:

- If you change your mind and want to increase N to 5, you can simply modify the second line.
- If you want to change 0.8 to 0.6, you can simply change the line in the `else` block.

Codes easy to maintain are easy to find bugs in them.

4.2 `while`

If you want to continue iteration while a certain condition is met, `while` is the perfect choice for you.

In the following code, value of the variable `a` is printed (by `cat()`) while `a > 0` holds. Since `a` starts from a positive number (`a <- 3`) and decremented by 1 in each step (`a <- a -`), `a > 0` will eventually be violated. At that point, the iteration terminates and execution goes out of the braces `{ }`

```r
a <- 3
while (a > 0) {
  cat(a, "\n")
  a <- a - 1
}
```

```
## 3
## 2
## 1
```

Guess what happens if you execute this code?

Hint. You see something going wrong? Type ESC or restart R session.

5 Exercise

Let us consider a linear system without input,

$$x_{t+1} = Ax_t,$$

where A is given by

```
A <- matrix(c(
  0.8, 1,
  0, 0.2
), nrow = 2, byrow = TRUE)
A
```

```
##      [,1] [,2]
## [1,]  0.8  1.0
## [2,]  0.0  0.2
```

Simulate this system for three alternative initial values

```
x10 <- matrix(c(1, 0))
x20 <- matrix(c(0, 10))
x30 <- matrix(c(-0.5, -0.5))
```

Plot the time series of the first elements of x_t.

Solution

The dynamic system is defined as follows.

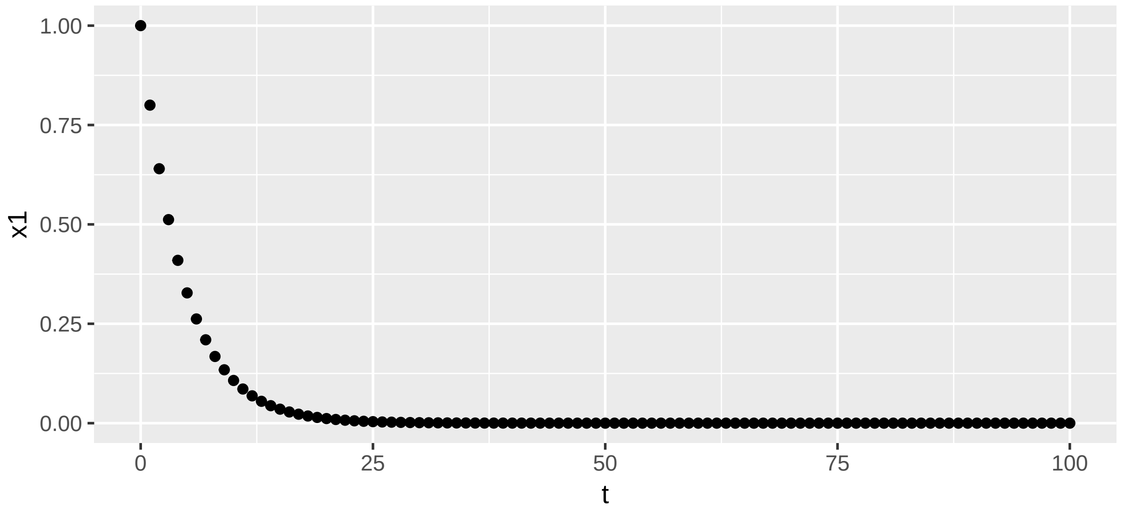
```
linear_autonomous <- function(a) {
  function(x) a %*% x
}
```

You can abstract the looping by the following function.

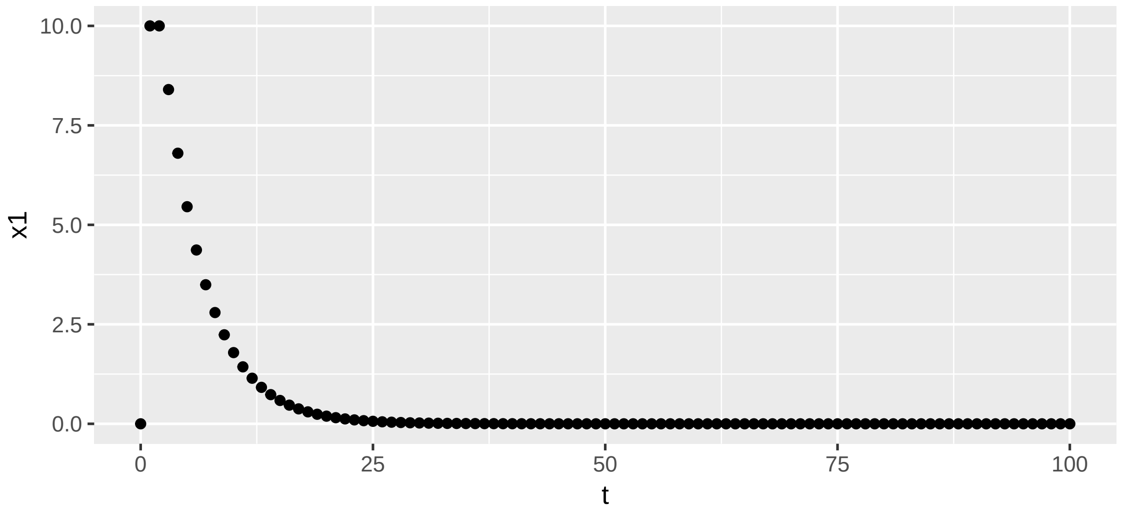
```
simulate <- function(x0, update_rule, nperiods) {
  result <- matrix(0, nperiods + 1, length(x0))
  result[1, ] <- x0

  for (t in 1:nperiods) {
    result[t + 1, ] <- update_rule(result[t, ])
  }
  result
}
```

Here are the results.

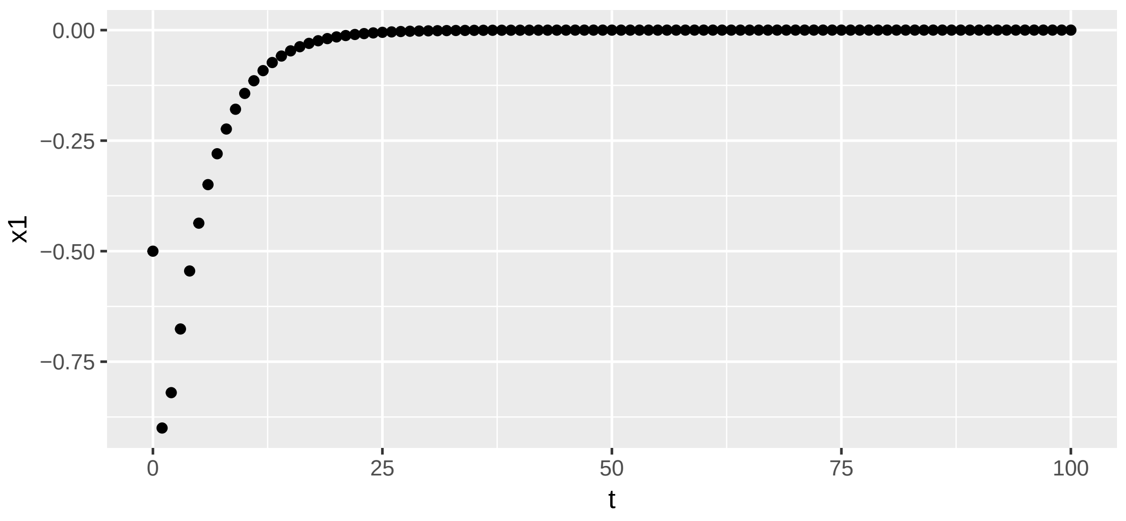
```r
tmax <- 100
t <- 0:tmax

result <- simulate(x10, linear_autonomous(A), nperiods = tmax)
x1 <- result[, 1]
ggplot2::qplot(t, x1)
```



```r
result <- simulate(x20, linear_autonomous(A), nperiods = tmax)
x1 <- result[, 1]
ggplot2::qplot(t, x1)
```



```r
result <- simulate(x30, linear_autonomous(A), nperiods = tmax)
x1 <- result[, 1]
ggplot2::qplot(t, x1)
```



0.00
−0.25
−0.50
−0.75
x1
0
25
50
75
100
t

6 Problems

Consider the following dynamical system:

$$x_{t+1} = Ax_t + Bu_t, \quad t = 0, 1, \ldots$$

where all components in x_0 are given.

Problem 1: Define `lssr()` function

Define a function `lssr()`, in `R/solution.R`, that meets the following specifications. (`lssr` stands for **l**inear **s**tate **s**pace **r**epresentation.)

1. `lssr()` takes two conformal matrices, `A` and `B` as parameters that correspond to A and B in $x_{t+1} = Ax_t + Bu_t$.
2. `lssr()` returns a function `g()`, which takes x_t and u_t as parameters and returns x_{t+1}.

To test that your code is correct, simulate the dynamic system with the following parameters

```r
## DO NOT EDIT

A <- matrix(c(
  0.0, 1.0, 0.0,
  0.0, 0.0, 1.0,
  -0.3, 0.7, 0.5
), nrow = 3, byrow = TRUE)

B <- matrix(c(
  0.0, 0.0, 1.0
))

x0 <- matrix(c(
  0.4678439, 0.5641670, 0.6803218
))
```

and with random inputs

```r
## DO NOT EDIT

# set.seed(100)
u <- function(n) {
  # Random numbers uniformly distributed between -0.5 and 0.5
  runif(n) - 0.5
}
```

Plot the first element in x_t against t.

In particular, write the function, `lssr()`, so that the following code works properly.

```
g <- lssr(A, B)
nperiods <- 1000
inputs <- u(nperiods)
result <- matrix(0, nperiods + 1, dim(A)[2])

result[1, ] <- x0
for (t in 1:nperiods) {
  result[t + 1, ] <- g(result[t, ], inputs[t])
}

ggplot2::qplot(0:nperiods, result[, 1], geom = 'line')
```

Problem 2: Simulation for different parameters

Choose different parameters and do the same exercise as above.

Problem 3: Optional

Rewrite the simulation code by using closure.