

POL 676 Computational Modeling
Oleg Smirnov (oleg.smirnov@stonybrook.edu)

Week 4 Lecture Notes and Homework Assignment

INTRODUCTION TO R
Installation and GUI overview.

OVERVIEW

R is an integrated suite of software facilities for data manipulation, calculation and graphical display.

Among other things it has

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either directly at the computer or on hardcopy, and
- a well developed, simple and effective programming language (called 'S') which includes conditionals, loops, user defined recursive functions and input and output facilities.

It has developed rapidly, and has been extended by a large collection of packages.

HOMEWORK Part I: R sample session from the Manual.

HELP in the Manual:	http://cran.r-project.org/doc/manuals/R-intro.html
Help on a "function":	> ?function
Help on a "keyword":	> help.search("keyword")
Help on a "keyword" online:	> RSiteSearch("keyword")
Help on a "keyword" online:	http://www.google.com

COMMANDS 101

Separation:	semicolon or newline
Comments:	> # comment
Assignment:	= or <-
Incomplete command output:	> +
Previous command(s)	use the vertical arrow key

OPEN file with saved R code via GUI

The function sink,

```
> sink("record.lis")
```

will divert all subsequent output from the console to an external file, record.lis. The command

```
> sink()
```

restores it to the console once again.

OBJECTS in memory:	> objects()
Remove object x:	> rm(x)
Remove objects x, y and z	> rm(x,y,z)

To set up a VECTOR named x, say, consisting of five numbers, namely 10.4, 5.6, 3.1, 6.4 and 21.7, use the R command

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
or
> x = c(10.4, 5.6, 3.1, 6.4, 21.7)
```

Vectors can be also created via commands like rnorm, e.g.

```
> x = rnorm(10)
```

Basic vector OPERATIONS

```
> y = c(x, 0, x)
```

MATH: +, -, *, /, ^ and common functions such as log, sqrt, etc.

Also: max(x), min(x), length(x), sum(x), prod(x), mean(x), var(x), sort(x)

SEQUENCES:

```
Example: > x=1:5
          > x=c(1,2,3,4,5)
          > x=seq(from=1,to=5)
          > x=seq(1,5)
          > x=seq(from=1,by=1,length=5)
          or
          or
          or
```

A related function is rep() which can be used for replicating an object in various complicated ways. The simplest form is

```
> s5 <- rep(x, times=5)
```

which will put five copies of x end-to-end in s5. Another useful version is

```
> s6 <- rep(x, each=5)
```

which repeats each element of x five times before moving on to the next.

ELEMENT *i* in a vector x: > x[i]

Creating and initializing arrays:

```
> #population array pop of size N and two attributes (fitness, type)
> #initially assigning zero to all attributes
> pop=array(0,dim=c(N,2))
> #starting population is uniformly random with 7 possible types
> pop[1:N,2]=as.integer(1+7*runif(N))
```

TRANSPOSE (very useful command!) > xt = t(x)

Here xt is the transpose of x.

Basic PROGRAMMING

The language has available a conditional construction of the form

```
> if (expr_1) expr_2 else expr_3
```

where `expr_1` must evaluate to a single logical value and the result of the entire expression is then evident.

```
Example: > # if the type of individual x is 1 then Coop is 1
> if (pop[x,2]==1) Coop=1
```

Logical operators:

```
AND:      &
OR:       |
NOT:      !
```

There is also a for loop construction which has the form

```
> for (name in expr_1) expr_2
or
> for (name in expr_1) {
> expr_2
> }
```

where `name` is the loop variable. `expr_1` is a vector expression, (often a sequence like 1:20), and `expr_2` is often a grouped expression with its sub-expressions written in terms of the dummy name. `expr_2` is repeatedly evaluated as `name` ranges through the values in the vector result of `expr_1`.

```
Example: > for (x in 1:N) {
> #whatever happens with ech individual in N
> }
```

Important recommendation!

Whenever possible avoid using IF and FOR constructions and instead try to use simple vector operations!

Working through an example:

```
# COEQUALS extended simulation

# parameters
# b-benefit form altruism, c-cost of altruism, R-number of rounds
# N-population size, G-number of generations, mr-mutation rate
# np - no play option payoff (note: mutual defection payoff is zero)

b=4;c=1;R=4;N=100;G=2000;mr=0.005;np=1;

#arrays for type operations
Lottery=array(0,dim=c(N,7))
Tot=array(0,dim=c(7))
P=array(0,dim=c(7))
AF=array(0,dim=c(7))
```

```

#population array, fitness temp, type, fitness final
pop=array(0,dim=c(N,3))

#data array to record proportions of types
# 1-cooperators, 2-defectors, 3-coequals, 4-anticoequals,
# 5-cooperate with poor only, 6-cooperate with rich only, 7-no play
data=array(0,dim=c(G,7))

# starting fitness (ensured to be always positive)
pop[1:N,1]=1+R*c
pop[1:N,3]=0

# if implemented then the starting population is 100% coequals
#pop[1:N,2]=3

# if implemented then the starting population is uniformly random
pop[1:N,2]=as.integer(1+7*runif(N))

#cycle of generations
for (g in 1:G) {

#cycle of rounds
for (r in 1:R) {

#cycle of individuals in the population
for (x in 1:N) {
NoPlay=0;Coop1=0;Coop2=0
y=as.integer(1+N*runif(1))

# TYPE 1: pure cooperators
if (pop[x,2]==1) Coop1=1
if (pop[y,2]==1) Coop2=1

# TYPE 2: pure defectors
# no need for code here: defection by default

# TYPE 3: classic coequals
if (pop[x,2]==3 & pop[x,3]==pop[y,3]) Coop1=1
if (pop[y,2]==3 & pop[y,3]==pop[x,3]) Coop2=1

# TYPE 4: anti-coequals
if (pop[x,2]==4 & pop[x,3]!=pop[y,3]) Coop1=1
if (pop[y,2]==4 & pop[y,3]!=pop[x,3]) Coop2=1

# TYPE 5: cooperate with poorer only
if (pop[x,2]==5 & pop[x,3]>pop[y,3]) Coop1=1
if (pop[y,2]==5 & pop[y,3]>pop[x,3]) Coop2=1

# TYPE 6: cooperate with richer only
if (pop[x,2]==6 & pop[x,3]<pop[y,3]) Coop1=1
if (pop[y,2]==6 & pop[y,3]<pop[x,3]) Coop2=1

# TYPE 7: no play option
if (pop[x,2]==7) NoPlay=1
if (pop[y,2]==7) NoPlay=1

# x-player interaction payoff; yes, there is no y-player interaction
# technical reasons... this should not bias the result
pop[x,1]=pop[x,1] + (1-NoPlay)*(Coop2*b - Coop1*c) + NoPlay*np
}

#update longer term fitness with the round results
pop[1:N,3]=pop[1:N,3]+pop[1:N,1]

```

```

}

#average population fitness
AFP=sum(pop[1:N,3])/N

#replicator dynamics (aggregate level proportions)
for (i in 1:7) {
Tot[i]=sum(pop[1:N,2]==i)
P[i]=data[g,i]=Tot[i]/N
if (Tot[i]>0) AF[i]=sum(pop[pop[1:N,2]==i,3])/Tot[i]
#discrete RP
P[i]=P[i]*(AF[i]/AFP)
}

#individual types given aggregate level proportions (probabilities)
#stochastic replicator dynamics via transposed multinomial random number
generation
Lottery=t(rmultinom(N, size = 1, prob=c(P[1],P[2],P[3],P[4],P[5],P[6],P[7])))
for (i in 1:N) {
pop[i,2]=which.max(Lottery[i,1:7])
}

#code for mutation:
# 1) find how many mutants in the population (uniformly probabilistic form 0
to N*mr*2)
# 2) pick randomly that many individuals and mutate
mutants=round(N*mr*runif(1)*2)
for (i in 1:mutants) {
mutant=as.integer(1+N*runif(1))
pop[mutant,2]=as.integer(1+7*runif(1))
}

#reset starting fitness(es)
pop[1:N,1]=1+R*c
pop[1:N,3]=0
}

matplot(1:G,cbind(data[1:G,1],data[1:G,2],data[1:G,3],data[1:G,4],data[1:G,5]
,data[1:G,6],data[1:G,7]))
done=1

```

Homework Part II: Your first simulation in R.

A population of size N plays Paper-Scissors-Rock game over G generations (1 game per generation). The evolutionary dynamic is the discrete replicator dynamics without mutation.

Assume that winning gives an individual 3 units of fitness, draw 2 units of fitness, and loss 1 unit of fitness.

Create two models. (1) the opponent is chosen randomly.
(2) the population is located along a single dimension (on a line).

Qualitatively compare the dynamics in two models. (for your simulation runs use the following parameters: $N=200$, $G=200$).