Introduction to using R for programming

Outline for today

- Computer programming & languages
- Program structure
- Important concepts
- Debugging
- R Packages & version control by Rich FitzJohn
- EXTRA: three reproducibility talks

Computer programming

Use a computer language to write a program, which does what you want it to do!

This can be to:

- Solve a theoretical model:
    - Analytical
    - Numerical
    - Individual-based simulations
- Generate ‘fake’ data for an experiment you are planning
- Make your own fun game
- Text mine Twitter
- Read and manipulate data and do stats on it
- ……

Computer languages

An artificial language to communicate instructions to a computer.

Many different ones (1000's):
Python, C, Perl, Ruby, R, Mathematica, …..

Unfortunately, different syntax often needed to do the same thing.

Which one to pick depends on what you want to use it for.

We will use R.
For some things great, not so much for others…

Don’t despair!

You can write code in e.g. C, have R to call a C compiler (‘runs’ the program) and have the output be send back to R!

Best of both worlds.

Starting computer programming

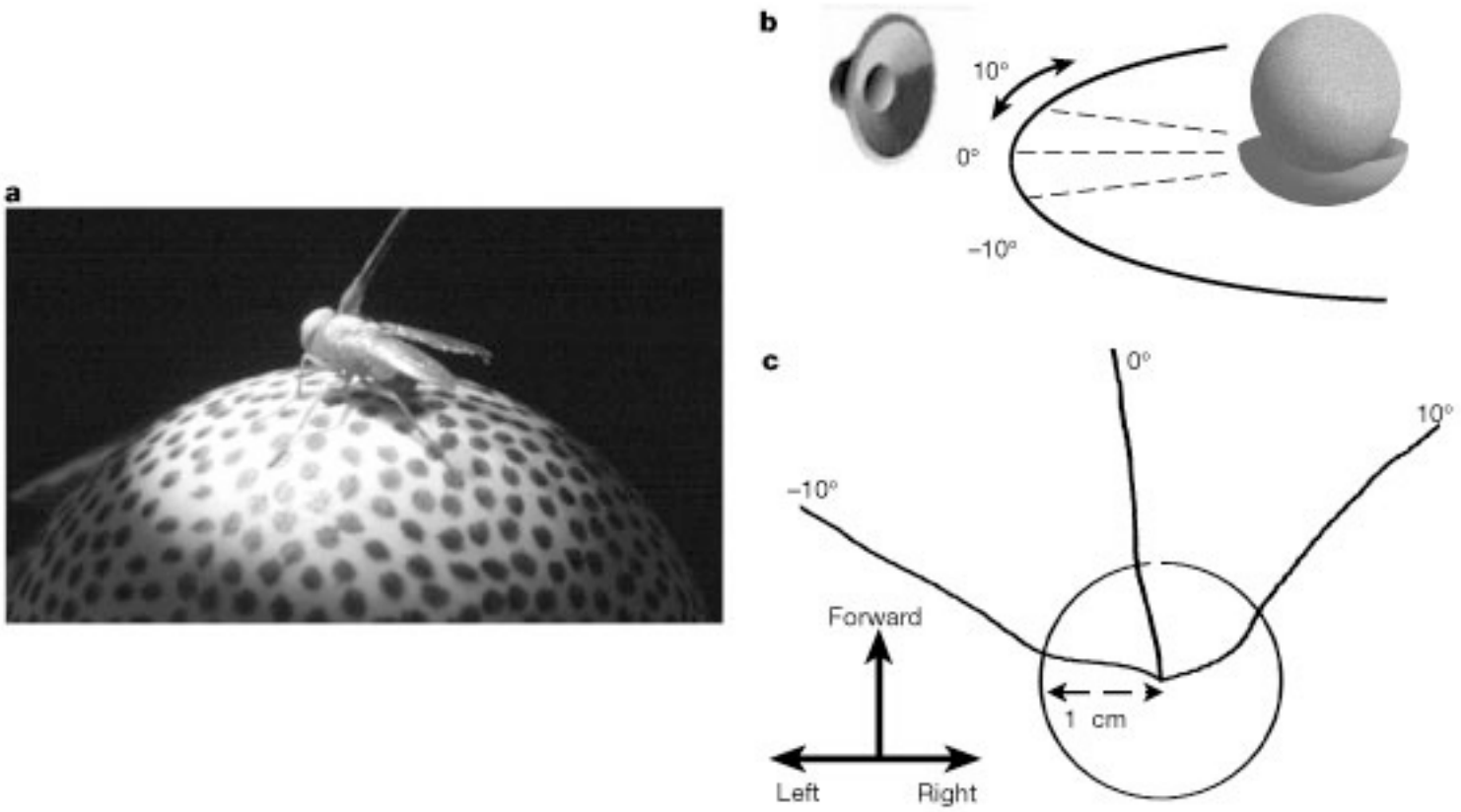
You have already started computer programmed.
`for()`,using `function(), etc`

An r script could be seen as a program.
You can run it all at once using `source ("scriptName.r") in your console.`

Example 1: analyzing cricket phonotaxis experiment.

Song is important for mate choice and playback experiments are used to investigate what song characteristics matter.

Record different songs types and play them back to a female, which is mounted on top of a 'spherical treadmill'.



Phonotaxis in flies: Mason et al 2001 Nature

Two sensors on the side of the sphere measure rotation, which you get with the time in the output file.

Nr	Cnt1X	Cnt1Y	Cnt2X	Cnt2Y	sec100	hour
1	22	110	30	-35	44	14
2	19	239	39	-36	93	14
3	41	329	110	-63	148	14
4	45	533	127	-102	198	14
5	27	765	105	-138	247	14
6	-20	1016	68	-152	297	14
7	-173	1285	-46	-157	346	14
8	-256	1564	-87	-179	396	14

First: visualize!

Second: create output summary needed for statistical analysis.

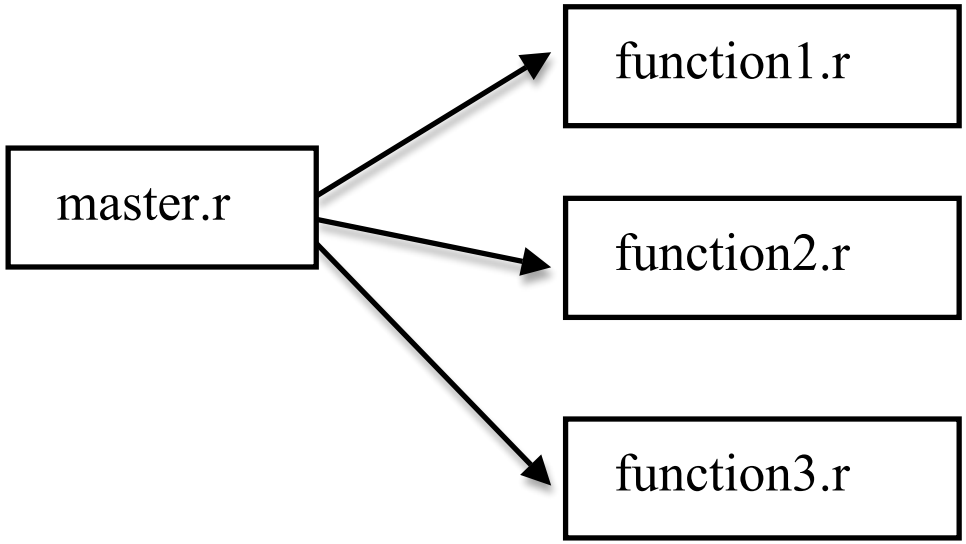
Structuring a program

Good idea to make your program modular.

The master file

Simple program: all code in one script file (master.r)

More complex program: store different program modules in different files.

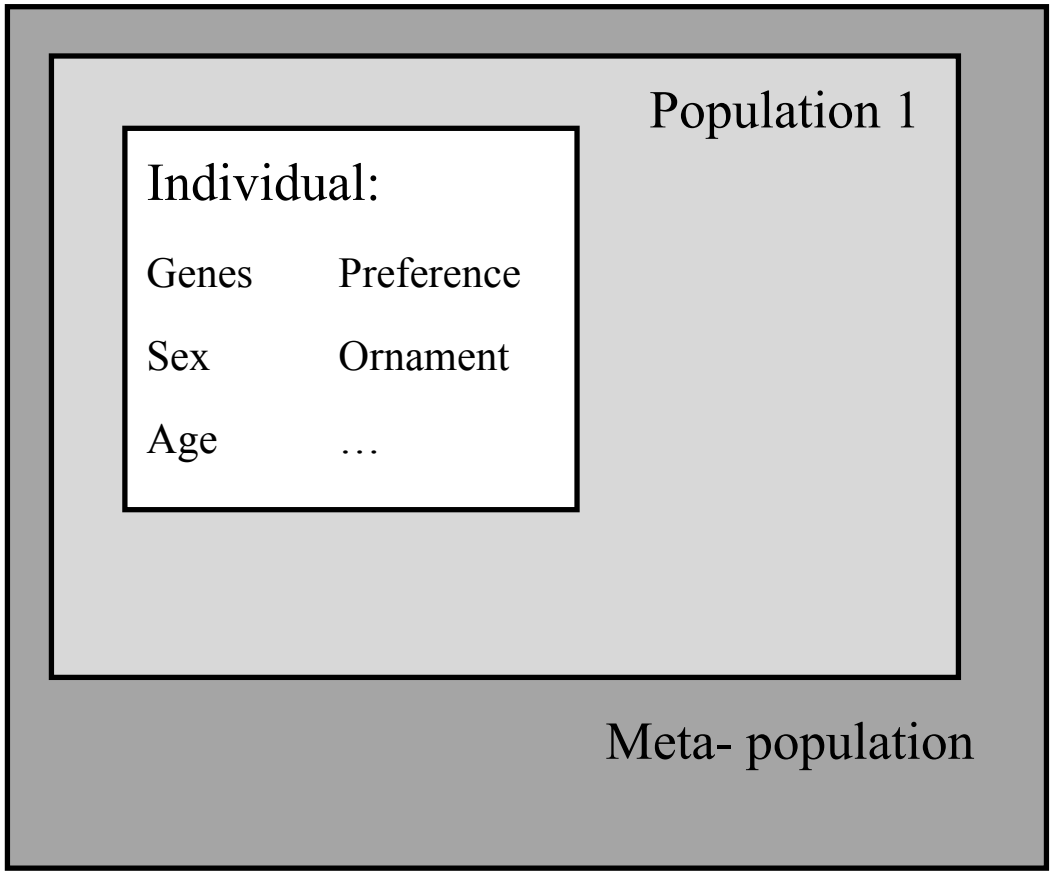


‘Structured’ programming

Structure can also be reflected in how you compartmentalize your code and/or output.

An example: when you run `lm()` you get an object returned, not all the data at once. This is very useful!

You can build your own program in a structured way:



Using vectors and lists

Lists can contain different data types.
Handy for making individual-based simulations.

Individual	*1*	*2*	*3*	*…*	*i*	
Sex	[“m”]	[“f”]	[“m”]	….	[“m”]	vector
Age	[1]	[3]	[2]	….	[1]	vector
GenViabil	[[1]] 0,1,..,1	[[2]] 1,1,..,0	[[2]] 0,0,..,0	….	[[2]] 0,1,..,0	list
GenOrnam	[[1]] 1,0,..,1	[[2]] 0,1,..,0	[[2]] 1,1,..,1	….	[[2]] 0,1,..,1	list
State	[“r”]	[“r”]	[“h”]	….	[“h”]	vector
Population	[1]	[1]	[2]	….	[x]	vector

On this structure we can use our functions (either standard or our own).

Remember: index in the vector represents an individual, different vectors provide different information of that indiv.

-viability selection: selection for genotypes with many 1's -> function which sums 1's per indiv, get the mean and lowest 10% removed.

-migration: proportion *m* gets a new population number.

-mate choice: often a little more complicated.
Genotype -> phenotype (could make a new vector for that).
Function to calculate attractiveness male for female.
Mate choice.

Scope of variables and functions

Scope of variables

local: a variable is only visible within a function body.

global: variables created outside a function are available both within and outside the function.

Functions

The meat of your programs will be functions, performing a specific part.
For example,
- determine which individuals survive in a population
- add mutation to a genotype
- run an statistical test and extract only the data you want

We already used many of R's build in functions (e.g. `sample()`).
We can also build our own!

```
function(){
   CODE
}
```

`apply()` family; become friends with them, they are great. Replace for loops with an apply member.

`apply(X, MARGIN,FUN,…) : data, 'over what', a function`

Applies a function to:

`apply`	an array over margins (e.g. matrix columns)
`lapply`	a list, returning a list of same length, applying FUN to each element of X

`sapply`	a user-friendly version of `lapply` returning a vector or matrix if appropriate
`tapply`	a cell or array identified by a unique combination of factor levels (e.g. treatment, or treatment and sex)

Example 2: scope and functions

Master file

This is the core of your program and should be simple to read:

```
initialize populations
for(i in 1:last generation ){
```

```
survival
mate choice
reproduction
}
```

Example 3:
Individual based simulations of how sexual selection may affect speciation.

Debugging

Often, writing the code is the fast part, finding the mistakes (debugging) is the tricky, time consuming part.

You have already been debugging (e.g. finding that typo).

The Principle of Confirmation
Fixing a buggy program is a process of confirming, one by one, that the many things you believe to be true about the code actually are true. When you find that one of your assumptions is not true, you have found a clue to the location (if not the exact nature) of a bug.

The Art of Debugging, with GDB, DDD, and Eclipse by Salzman & Matloff (2008)

Principle of confirmation

Check if what you assume is right: *guilty unless proven innocent*!

1) Carefully read through the code for e.g. typos.

2) Print output to screen using `print()` or `cat ()`

3) 'Antidebugging': `stopifnot()`
keep going unless a condition is not met (slow)

4) `traceback()` tells you where the error was

5) Other method which let you run through code line by line
- `browser()`
- `debug()`

Couple of debugging suggestions:

- start with the core parts of your code and work ‘up’
- keep a log file of changes at the top of your main file
- make sure you can (easily) run program without the debugging parts (breakpoints are handy for that)
- keep track of where you are in the code (use `print()`)
- if you use random numbers, use `set.seed()`. You can replicate the run (but change seeds between runs!)

Example 4: debugging

Abstract: If you are using R and you think you're in hell, this is a map for you.

from **R inferno** by Patrick Burns

Useful links:
R inferno : http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
Programming R general website: http://www.programmingr.com
Handy intro to R programming: http://zoonek2.free.fr/UNIX/48_R/02.html