Scientific programming in R

Dr. Henry Löffler-Wirth
Maria Schmidt
Dr. Lena Sünke Mortensen

Interdisciplinary Center for Bioinformatics - IZBI, University Leipzig

Course material: http://izbifs.izbi.uni-leipzig.de/~wirth/R/Day3.pdf
Organization


Day 1: R syntax, very basic programming techniques (vectors, loops, conditions)

Day 2: Data handling, data types, visualization basics and simple statistics

Day 3: Programming projects

Day 4: Apply-functions, object-orientated programming, graph handling

Day 5: Interactive Apps with Shiny
Programming project I – Balloon Packaging Machine

Henry bought a fully-automated balloon packaging machine (BPM).

The machine consists of 10 boxes, each containing 1-10 balloons, and a loadout storage.

```plaintext
balloon boxes: 4 8 3 10 7 3 7 5 1 6
loadout: 0
```

Machine is working: One box is selected and emptied into loadout. Then box is refilled with random number of balloons [1:10].

```plaintext
balloon boxes: 4 8 3 10 6 3 7 5 1 6
loadout: 7
```

Whenever loadout contains at least 20 balloons, they will be moved into a pack.

```plaintext
loadout: 20
→
loadout: 0
```

Packs with less than 20 balloons are not ready yet, packs with more than 20 balloons reduce profit.
Programming project I – Balloon Packaging Machine

Henry destroyed the control-packaging-unit (CPU). Therefore, our machine randomly selects the boxes for loadout.

Implement the BPM functionality in R:

• What values should be stored? Put them into suited data structures.

• Initialize a brand new BPM (boxes randomly filled, loadout empty). Hint: Use `sample(1:10, 1)` to generate a random number [1,10].

• Code for emptying a random box into loadout and refilling the box.

• Code for emptying loadout and producing a balloon pack.

We got a huge order: 1.000.000 packs of balloons (minimum 20 per pack).

• Simulate the packaging process until one pack is ready. Is everything working fine?

• Simulate the packaging process until 1.000.000 packs are ready.

• How many balloons have been packed? How many balloons are in each pack on average?

• Repair the CPU: Modify box selection to minimize loss of benefit / spare balloons. Is there a perfect strategy?
A **cellular automaton** is a discrete model consisting of a regular grid of **cells**, each in one of a finite number of **states**, such as *on* and *off*.

For each cell, a set of cells called its **neighborhood** is defined relative to the specified cell.

An initial state (time $t = 1$) is selected by assigning a state for each cell.

A new generation is created (advancing $t$ by 1), according to some fixed **rule**.

Rules determine the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood.

Typically, the rule for updating the state of cells is the same for each cell and does not change over time, and is applied to the whole grid simultaneously.

(source: wikipedia.org)
Simple cellular automaton: Wolfram's universe

Rule:

<table>
<thead>
<tr>
<th>t</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

one-dimensional 'world'
Programming project II – Cellular automata

Tasks & steps:

• How to store the world at each time step? Matrix with rows=positions and cols=time?

• Initialize the world empty with only one living cell at $t=1$. Start small, e.g. size of 11, and 10 steps.

• Visualize the world. `image()` is handy for matrix visualization.

Next generation of cells should be generated using a simplified rule: A new cell is born, if there is **exact** one living cell in the neighborhood triplet at $t-1$.

• Write some code: at $t=1$, look into the neighborhood of one arbitrary position in the world (this position + left & right neighbor) and count living cells

• If there is one living cell in the neighborhood, a new living cell will be placed in this position at $t=2$

• Repeat for all positions in the world

• Repeat this updating for 10 time steps

Working? Make your world bigger and simulation longer!
Optional task 1:
• At the big bang (t=1), initialize cells randomly (50% dead, 50% alive)

Optional task 2:
• Increase the neighborhood to 2 cells to left & right

Optional task 3:
• Change the rule: a new cell is born if there were exactly 2 or 4 living cells in the neighborhood

Optional task *:
• If you are too fast and bored, ask us for the two-dimensional version of this cellular automaton!
Programming project II* – Conway's Game of Life

The **Game of Life** is a cellular automaton devised by the British mathematician John Horton Conway in 1970.

The universe of the Game of Life is an two-dimensional orthogonal grid of square *cells*, each of which is in one of two possible states, *alive* or *dead*. Every cell interacts with its eight *neighbours*, which are the cells that are horizontally, vertically, or diagonally adjacent.

At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by over-population.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The initial pattern constitutes the *seed* of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a *tick* (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

(source: wikipedia.org)
Programming project II* – Conway's Game of Life

Tasks & steps

• Initialize a world matrix. Dimension and relative amount of living cells shall be variable.

• Visualize the world.

• Write a function counting living cells in neighborhood of a given position (no toroidal borders).

• Write a loop realizing changes of the world in each tick. Recall that all events occur simultaneously. Plot the world in each tick. Hint: `Sys.sleep(s)` pauses execution for s seconds.