Part 1 Section 3: R Language Interfacing

Aaron Robotham

Load libraries needed for this chapter:

```
library(magicaxis, quietly=TRUE)
library(Rcpp, quietly=TRUE)
library(microbenchmark, quietly=TRUE)
library(parallel, quietly=TRUE)
library(doParallel, quietly=TRUE)
library(foreach, quietly=TRUE)
library(doSNOW, quietly=TRUE)
library(bigmemory, quietly=TRUE)
```

R talks Python

There are often arguments online about the supremacy of various programming languages, and in data science there are often comparisons between **R** and **Python**. I will not get too bogged down in the pros and cons, but the good news is you can easily work with the best of both worlds using various well supported packages.

On the **R** the most effort in recent years has been put into the **reticulate** package. This allows **R** to interface with **Python** and create bespoke **Python** environments for data analysis, all within a normal **R** session. Here we start by pointing **R** towards the system **Python** that we wish to use (**Anaconda** in this case):

```
library(reticulate, quietly=TRUE)
```

We can then import modules that we already have installed within **Python**. Obviously these need to be installed on your system for this to work:

```
py_require("numpy")
py_require("pandas")
```

```
np = import("numpy", convert=FALSE)
pd = import("pandas", convert=FALSE)
```

As a test we will write out a simple table using **R** and read it back in using **Pandas**, where all **Pandas** function are exposed through the *pd* object we pointed to above:

```
write.csv(data.frame(x=1:10,y=21:30), file='../data/test.csv')
test = pd$read_csv('../data/test.csv')
test['x']
test['y']
```

The above is being accessed in **Python**, hence the formatting looks a bit different to what we have seen so far.

We can fully convert the **Python** table to and **R data.frame** with a built in conversion function:

```
test2 = py_to_r(test)
test2$x
test2$y
```

The object *test2* is now a normal **R** object, that behaves just like you would expect such an object to behave. There are sensible type conversions that cover the vast majority of **R** and **Python** use cases,

and of course you can convert in the other direction also:

```
test3 = r_to_py(test2)
test3['x']
test3['y']
```

It is worth noting that the above table, whilst similar numerically, has actually converted the integers to floats, so care should be taken when converting between **R** and **Python**.

If you get into machine learning with **R**, most of the interfacing is actually done through **Python** and its mature libraries for **Keres** and **TensorFlow**, so knowing some of the basics of **reticulate** is useful. However, in this course we will not be making any further explicit use of **Python** to tackle any of the problems we encounter. It is also required that solutions to assignments etc are always provided in vanilla **R**, so do not treat **reticulate** as a back door route to providing essentially **Python** solutions.

R Talks...

R also speaks to other popular high-level (non-compiled, interpreted) languages like **Perl**, **Ruby** and **Julia**. Your mileage will vary since some of these languages are more popular today than others, and not all interfaces are present on CRAN (since that has pretty stringent compatibility tests that very old code often fails).

Getting Faster: Rcpp

R allows for fairly deep recursion in that a function can call itself a lot of times. Doing this creates a lot of overhead though, and can seriously slow code down. Basically, deep looping in **R** (like most interpreted languages) is pretty slow. Take this simple Fibonacci sequence function written in fairly vanilla **R**, where it returns the x^th Fibonacci sequence number, where it goes 1,1,2,3,5,8 etc.

```
fibonacci_R = function(n) {
  if (n < 2) return(n)
  return(fibonacci_R(n-1) + fibonacci_R(n-2))
}
```

Figure out for yourself what this function is actually doing- it is quite clever (I did not write it BTW).

From the above we expect the 6^{th} value to be 8:

```
fibonacci_R(6)
```

```
## [1] 8
```

We can check the computation time for this recursive algorithm with the **system.time** function:

```
system.time(fibonacci_R(10))
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(fibonacci_R(20))
```

```
##    user  system elapsed
##   0.003   0.000   0.003
```

```
system.time(fibonacci_R(30))
```

```
##    user  system elapsed
##   0.338   0.002   0.339
```

Notice that the code rapidly diverges in run time!

We can use the **Rcpp** package to rewrite the Fibonacci function in **C++** code. All the fiddly header and and binder stuff (including type matching) is handled by the **Rcpp** package. The whole design of the

package is such that you can write code that look *very* similar to **R** code bar a few modifications here and there (explicit typing), and some small boiler plate telling it how to export the function.

To use **Rcpp**, Windows users might need to go through a couple of additional steps depending on how their system is set up, but most likely you will need to at least install *Rtools*, which are available at https://cran.r-project.org/bin/windows/Rtools/ and follow the instructions about how to link these into your system path. You will know it is working because the following will not be empty:

```
Sys.which("make")
```

So, back to our Fibonacci algorithm- here is our **Rcpp** rewrite:

```
#include <Rcpp.h>

// [[Rcpp::export]]
int fibonacci_Rcpp(int n) {
  if (n < 2) return(n);
  return(fibonacci_Rcpp(n-1) + fibonacci_Rcpp(n-2));
}
```

This looks very nearly identical to our earlier **R** code! This is the whole purpose of **Rcpp**, it abstracts away most of the less familiar (to **R** users) parts of **C++** and focuses on the parts that really matter algorithmically. The small differences here are:

- Inside our **Rmarkdown** document we need to include the *Rcpp.h* header. Those of you who know **C** will be familiar with this, but in short this gives us access to the **Rcpp** library within the code chunk (meaning we do not need to write explicit paths to all functions).
- We need to tell **Rcpp** to *export* the function- this is so we can see and use it in our session.
- We have to tell it what type of output will be returns (*int* in this case).
- We have to declare the types of the argument inputs (*int* in this case).
- We do not declare we are defining a function by writing 'function' (which we do in **R**).
- We need to put explicit line ending semi-colons. You can write these in **R**, but they are not required.

But given those small differences, the actual inner code that makes up the function is effectively identical.

We can now run the same timings:

```
system.time(fibonacci_Rcpp(10))
```

```
##    user  system elapsed
##       0       0       0
```

```
system.time(fibonacci_Rcpp(20))
```

```
##    user  system elapsed
##   0.001   0.000   0.001
```

```
system.time(fibonacci_Rcpp(30))
```

```
##    user  system elapsed
##   0.002   0.000   0.002
```

Wow!! It is able to compute the 30^{th} Fibonacci number (832040) in less time than base **R** took to compute the 20^{th} (6765).

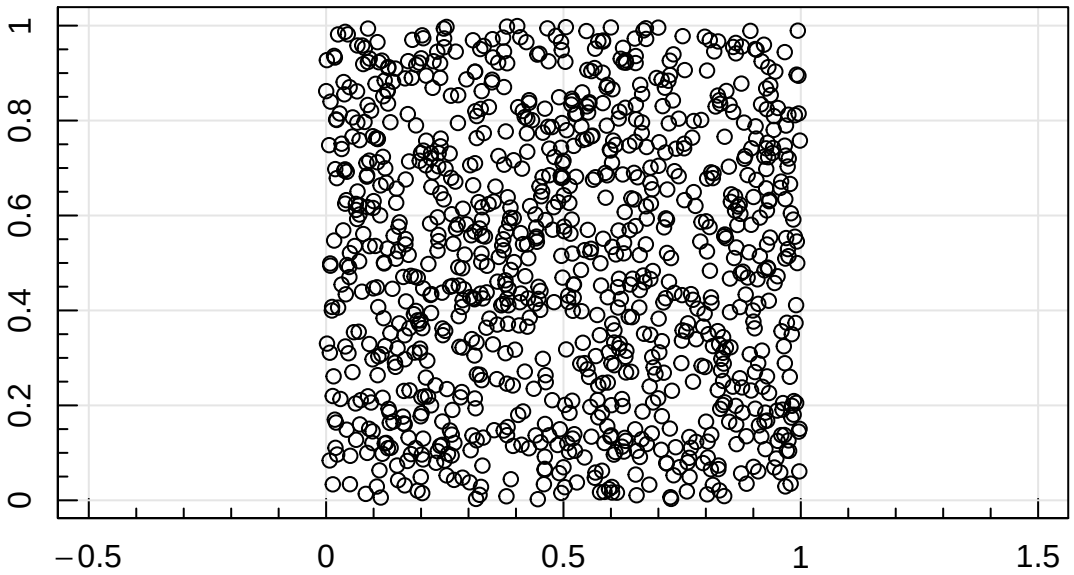
Beating R at its Own Game?

Sometimes you start to wonder if you can speed up your code by replacing base **R** functions with your own **Rcpp** functions. I will save you a lot of time and frustration by telling you that, for the most part, you will struggle. But as the saying goes, "Trust but Verify". I.e. We will try a couple anyway.

Distance Matrices

As a first example we are going to measure the separation between 1000 Uniformly random points in 2D. First we make our fake data:

```
random_xy = cbind(runif(1000), runif(1000))
magplot(random_xy, asp=1)
```



Here **runif** is an **R** function that generates randomly Uniform samples (1000 in this case), by default between 0 and 1 (we will look into various **R** distributions later).

We can check how long it takes to measure all 1,000,000 possible separations using **R**'s built in **dist** function.

```
microbenchmark(dist(random_xy))
```

```
## Warning in microbenchmark(dist(random_xy)): less accurate nanosecond times to
## avoid potential integer overflows
```

```
## Unit: microseconds
##              expr     min      lq     mean   median       uq     max neval
##  dist(random_xy) 869.405 990.355 1137.499 1014.893 1084.716 2856.06   100
```

And now we rewrite it again using **Rcpp**.

```
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::NumericMatrix dist_Rcpp(Rcpp::NumericMatrix x){
  int nrow = x.nrow();
  Rcpp::NumericMatrix m(nrow,nrow);
  for (int i = 0; i < nrow; i++) {
    for (int j = 0; j < nrow; j++) {
      m(i,j) = sqrt(pow(x(i,0)-x(j,0),2) + pow(x(i,1)-x(j,1),2));
    }
  }
  return m;
}
```

We can check the speed again.

```
microbenchmark(dist_Rcpp(random_xy))
```

```
## Unit: milliseconds
##                 expr      min       lq     mean  median       uq      max
```

```
##  dist_Rcpp(random_xy) 1.099661 1.167209 1.464695 1.22711 1.343857 3.028506
##  neval
##    100
```

So it is actually a very similar speed (a bit slower in fact). This is because the **dist** function uses **C** deep down, and it uses a lot of clever optimisations that are hard to appreciate unless you get deep into code development (they are also hard to understand). The advantage of our approach is the code is clear in what it is doing, and still pretty fast. Where you sometimes start to see speed improvements is when writing code that only handles inputs of a single type and without having to handle NAs and NaNs etc. All base **R** code has to be able to deal with all **R** types and special values, which can slow code down a lot. The danger is you might create code that breaks when, e.g., passed an integer matrix not a numeric one.

Column and Row Sums

R comes with routine to compute columns and row sums. This is a useful way to also check whether the order of matrix access makes much difference at the **Rcpp** level.

As a terrible starting point, let us do this explicitly in **R** (naturally, never do this!):

```
colsum_R = function(x){
  out = rep(0,dim(x)[2])
  for (i in 1:dim(x)[1]){
    for (j in 1:dim(x)[2]){
      out[j] = out[j] + x[i,j]
    }
  }
  return(out)
}
```

Now we will try some more reasonable **Rcpp** versions. First we will compute column sums where the outer loop is via rows:

```
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::NumericVector colsum_Rcppv1(Rcpp::NumericMatrix x){
  int nrow = x.nrow();
  int ncol = x.ncol();
  Rcpp::NumericVector out(ncol);
  for (int i = 0; i < nrow; i++) {
    for (int j = 0; j < ncol; j++) {
      out(j) += x(i,j);
    }
  }
  return out;
}
```

Next we will compute column sums where the outer loop is via columns:

```
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::NumericVector colsum_Rcppv2(Rcpp::NumericMatrix x){
  int nrow = x.nrow();
  int ncol = x.ncol();
  Rcpp::NumericVector out(ncol);
  for (int j = 0; j < ncol; j++) {
    for (int i = 0; i < nrow; i++) {
      out(j) += x(i,j);
    }
  }
```

```
  return out;
}
```

We now make some test data (a 1,000 x 1,000 matrix with 1 million random Uniform numbers between 0 and 1):

```
tempmat = matrix(runif(1e6),1e3)
```

```
microbenchmark(
  colsum_R(tempmat),
  colSums(tempmat),
  colsum_Rcppv1(tempmat),
  colsum_Rcppv2(tempmat)
)
```

```
## Unit: microseconds
##                     expr       min         lq      mean     median         uq
##       colsum_R(tempmat) 33818.481 33972.6000 34199.266 34069.5035 34182.0485
##        colSums(tempmat)   604.258   607.4765   616.130   610.2235   618.1775
##  colsum_Rcppv1(tempmat)  1702.730  1710.7250  1744.379  1727.9040  1750.3310
##  colsum_Rcppv2(tempmat)  1703.550  1707.0350  1993.343  1722.1230  1862.0765
##        max neval
##  42888.214   100
##    748.865   100
##   2092.230   100
##   2967.539   100
```

And the results show a couple of things. The explicit **R** code is horribly slow! This really shows the weak point of interpreted languages like **R**: deeply nested loops. The built in **R** routines appear to be *much* faster than anything else, and version 2 of our **Rcpp** code is a bit faster than version 1. This second result suggests that **Rcpp** stores matrix data in a column major format (i.e. adjacent elements in RAM are adjacent rows). In fact, we already know this from the previous lectures, but here is the implication of that reality.

The reason the base **R** code is so much faster here is because it is built against parallel matrix libraries on modern machine with multiple cores (pretty much all computers these days). We could go to the effort to recreate this parallel computation in **Rcpp**, but the code gets much more complicated to write and comprehend.

We will check this with row sums now (ignoring the explicit **R** code this time). First we will compute row sums where the outer loop is via rows:

```
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::NumericVector rowsum_Rcppv1(Rcpp::NumericMatrix x){
  int nrow = x.nrow();
  int ncol = x.ncol();
  Rcpp::NumericVector out(nrow);
  for (int i = 0; i < nrow; i++) {
    for (int j = 0; j < ncol; j++) {
      out(i) += x(i,j);
    }
  }
  return out;
}
```

Next we will compute row sums where the outer loop is via columns:

```
#include <Rcpp.h>

// [[Rcpp::export]]
```

```
Rcpp::NumericVector rowsum_Rcppv2(Rcpp::NumericMatrix x){
  int nrow = x.nrow();
  int ncol = x.ncol();
  Rcpp::NumericVector out(nrow);
  for (int j = 0; j < ncol; j++) {
    for (int i = 0; i < nrow; i++) {
      out(i) += x(i,j);
    }
  }
  return out;
}
```

And now we check the speeds again:

```
microbenchmark(
  rowSums(tempmat),
  rowsum_Rcppv1(tempmat),
  rowsum_Rcppv2(tempmat)
)
```

```
## Unit: microseconds
##                       expr      min        lq     mean    median        uq      max
##          rowSums(tempmat)  125.624  126.4645  133.439  128.8835  132.553  182.737
##  rowsum_Rcppv1(tempmat) 1703.222 1707.9780 1761.572 1726.9200 1748.650 2681.482
##  rowsum_Rcppv2(tempmat) 1702.976 1712.0575 1777.445 1728.8470 1751.294 2785.253
##  neval
##    100
##    100
##    100
```

As we expected from the last chapter, column summing is faster than row summing using the base **R** functions, and we see similar speeds for our **Rcpp** code, which is much slower than the **R** code again. Update: as of 2025, row summing is actually faster than column summing, and I am not sure why!

There is a lot of online material available that helps get people going with **Rcpp**, but even without using the advanced capabilities (of which there are lots) or much **C**/**C++** knowledge, you should be able to rewrite simple tasks in-line. But be warned, in many case you will do better to just use the available base **R** functions. As a general comment, people often refer to interpreted languages like **R** and **Python** as being "slow". This is only really true for explicitly nested loops or deeply recursive function calls (like the Fibonacci sequence at the beginning). For many types of data analysis that execute fairly vanilla "base" processing functions and operations (like row sums and means etc) you will find them both to be extremely fast even compared to **C** and **C++**.

Segfault!

A very common issue when using a low-level language like **C**/**C++** is that your code compiles without warning, but often (and sometimes haphazardly) crashes when running. Maybe sometimes the code will run, but other times you see a dreaded *segfault* warning on you screen. In the vast majority of cases this is due to memory management issues where you are trying to access (or write to) memory that is out-of-bound.

A classic bug would be you have a vector that is 100 elements long and you want to access the 100^{th} element. Now in **R** you can use the subset syntax example[100] (being 1-indexed), but if you did this in **C**/**C++** you would be accessing beyond the end of the vector because these languages are 0-indexed! This means you need to write something like example(99). In general round brackets are 'safer' for use in **Rcpp** (it does more check on the limits of the objects) and square brackets are a bit faster and more prone to out-of-bounds crashes. You can definitely still enduce segafaults with either notation if not careful though.

So, if you are having a segfault issue, double and triple check you vector/matrix allocations and accessing logic. That is the cause of 99% of such problems.

Other Low-Level Interfaces

R can also natively interface with **Fortran** (some of **R** is written in **Fortran** even!) This is not such a popular language outside of academia these days, but there is still a lot of astronomy software that uses it.

On the more modern side, **R** has recently gained an interface to the popular and powerful **Rust** programming language via the **rextendr** package. Digging into how this works is beyond the scope of this course (and my expertise), but keen students might want to do some Googling and have a play. The main reason **Rust** has become popular is because it is extremely memory-safe, and handles a lot of the book-keeping that cause the notorious **C**/**C++** segfault errors at runtime. Basically, if it compiles without an error then you can also guarantee it will not segfault (this will seem like a miracle after your N^{th} segfault code crash).

The Game of Life (no, not that one)

In 1970 British mathematician John Conway came up with a simple cellular automaton game that creates surprisingly complex "life" on a game grid. The original rules are very simple and based on local adjacency (where local adjacency includes diagonal offsets, so a single cell has 8 neighbours on an infinite grid):

1) Any *live* cell with two or three live neighbours survives.
2) Any *dead* cell with three live neighbours becomes a live cell.
3) All other *live* cells die in the next generation. Similarly, all other *dead* cells stay dead.

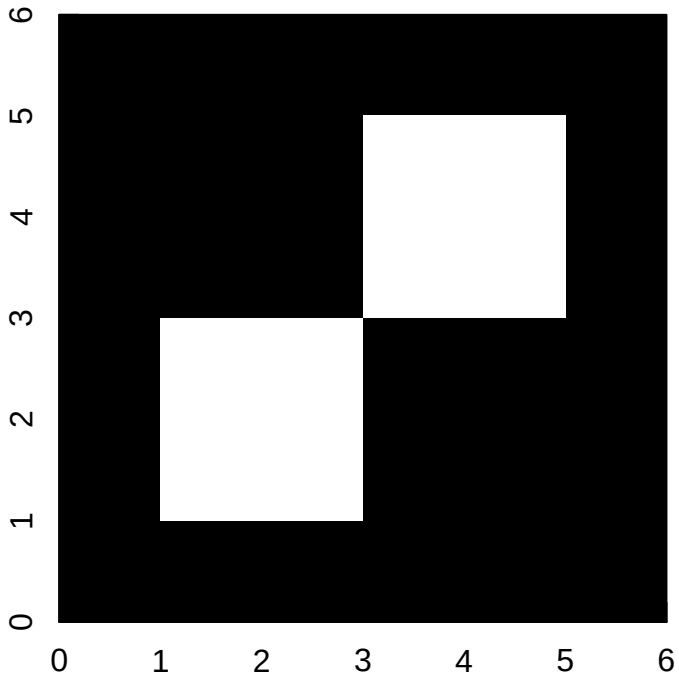
So let us first code this in pure **R** using matrix operations to determine the nearby neighbours:

```
evolve_n_R = function(game_mat, steps=1){
  dim_x = dim(game_mat)[1]
  dim_x_m1 = dim_x - 1L
  dim_x_m2 = dim_x - 2L
  dim_y = dim(game_mat)[2]
  dim_y_m1 = dim_y - 1L
  dim_y_m2 = dim_y - 2L

  for(i in 1:steps){
    BL = game_mat[1:dim_x_m2, 1:dim_y_m2]
    ML = game_mat[1:dim_x_m2, 2:dim_y_m1]
    TL = game_mat[1:dim_x_m2, 3:dim_y]
    TM = game_mat[2:dim_x_m1, 3:dim_y]
    TR = game_mat[3:dim_x, 3:dim_y]
    MR = game_mat[3:dim_x, 2:dim_y_m1]
    BR = game_mat[3:dim_x, 1:dim_y_m2]
    BM = game_mat[2:dim_x_m1, 1:dim_y_m2]

    state_mat = BL + ML + TL + TM + TR + MR + BR + BM

    game_mat[2:dim_x_m1, 2:dim_y_m1][state_mat < 2L] = 0L
    game_mat[2:dim_x_m1, 2:dim_y_m1][state_mat == 3L] = 1L
    game_mat[2:dim_x_m1, 2:dim_y_m1][state_mat > 3L] = 0L
  }

  return(game_mat)
}
```

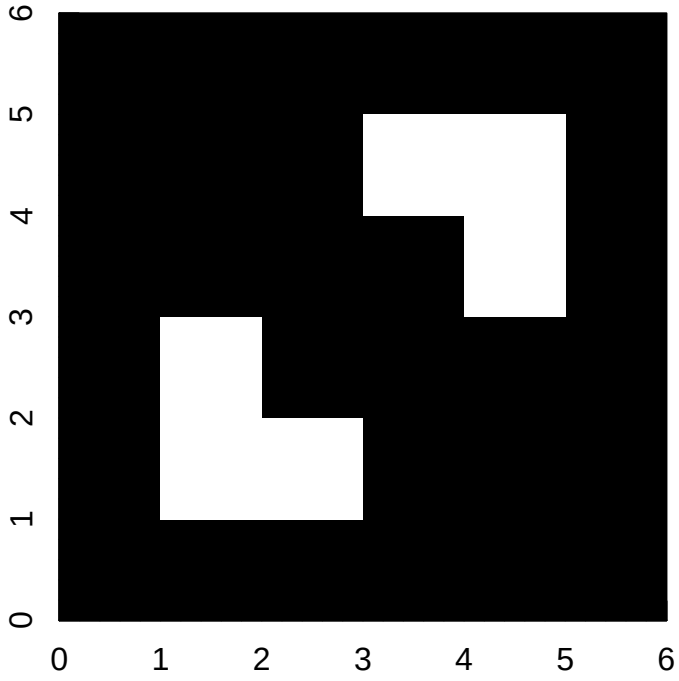
We can start with a simple example that is known to oscillate (called a beacon):

```
beacon = matrix(0L, 6, 6)
beacon[2:3,2:3] = 1L
```

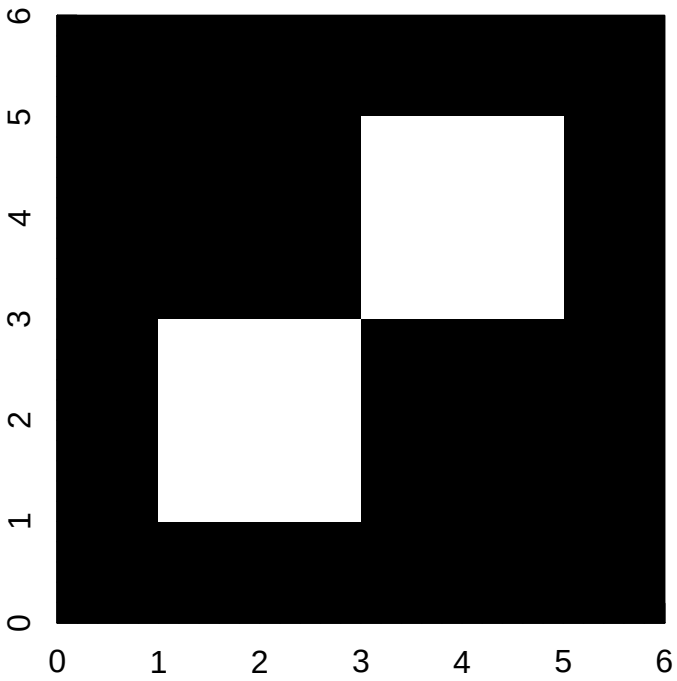
```
beacon[4:5,4:5] = 1L
par(mar=c(3.1,3.1,1.1,1.1))
magimage(beacon)
```




```
beacon = evolve_n_R(beacon)
par(mar=c(3.1,3.1,1.1,1.1))
magimage(beacon)
```



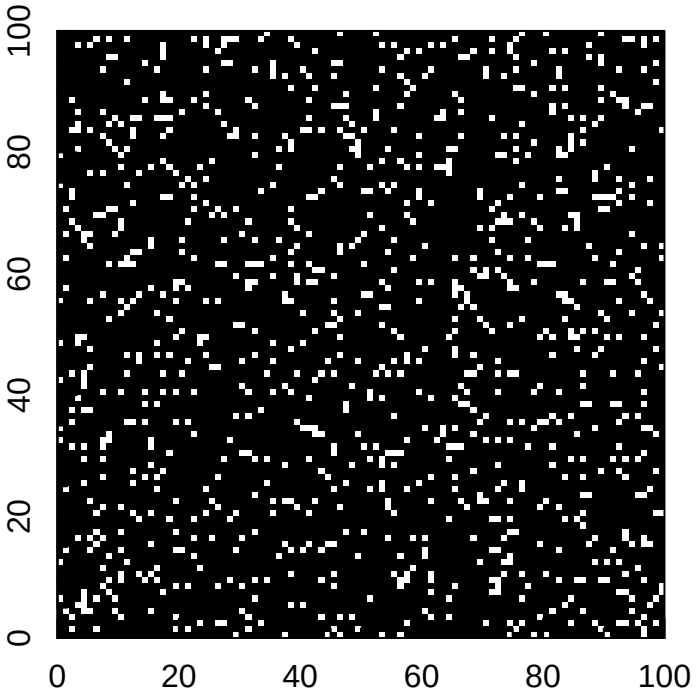

```
beacon = evolve_n_R(beacon)
par(mar=c(3.1,3.1,1.1,1.1))
magimage(beacon)
```



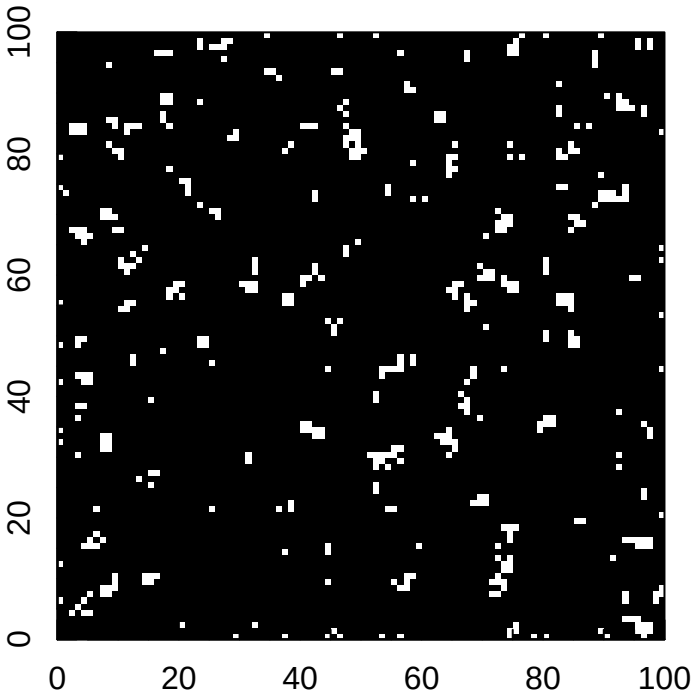
Next we can make a larger random example in R and evolve it a bit:

```
set.seed(666)
game_mat = matrix(0L, 1e2, 1e2)
game_mat[sample(prod(dim(game_mat)),1e3)] = 1L
```

```
par(mar=c(3.1,3.1,1.1,1.1))
magimage(game_mat)
```

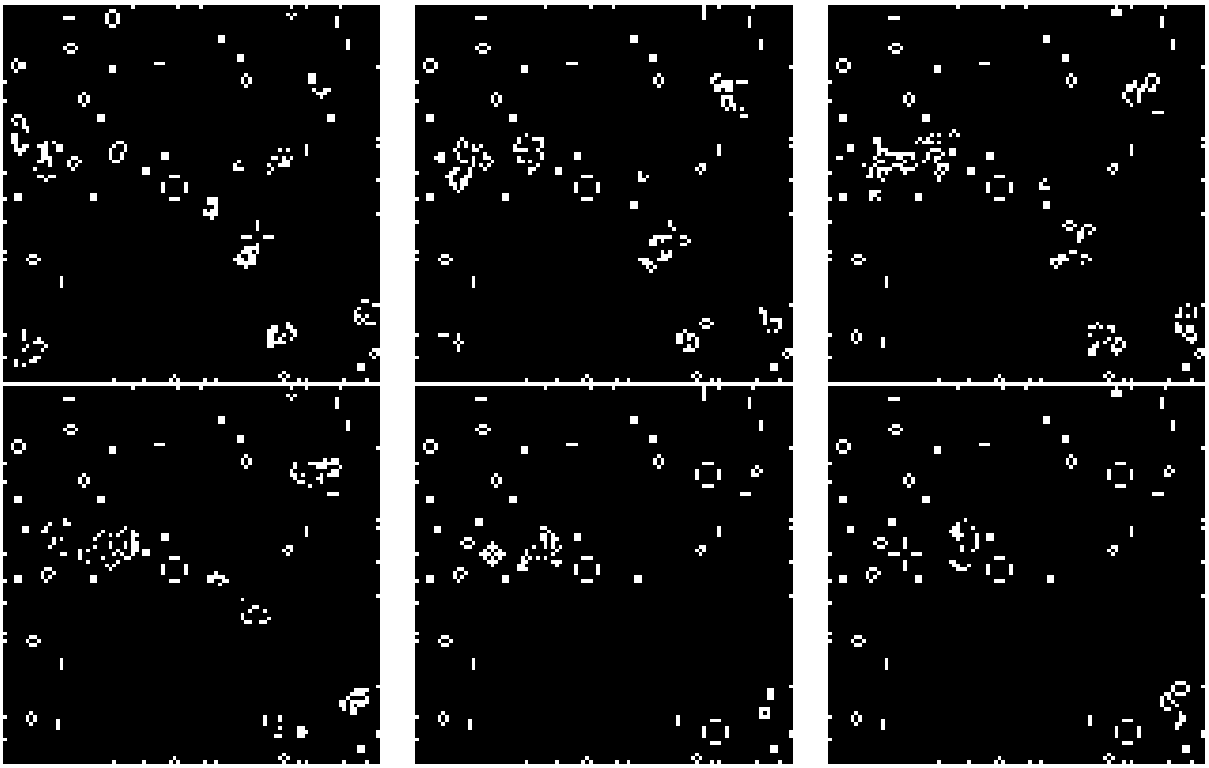


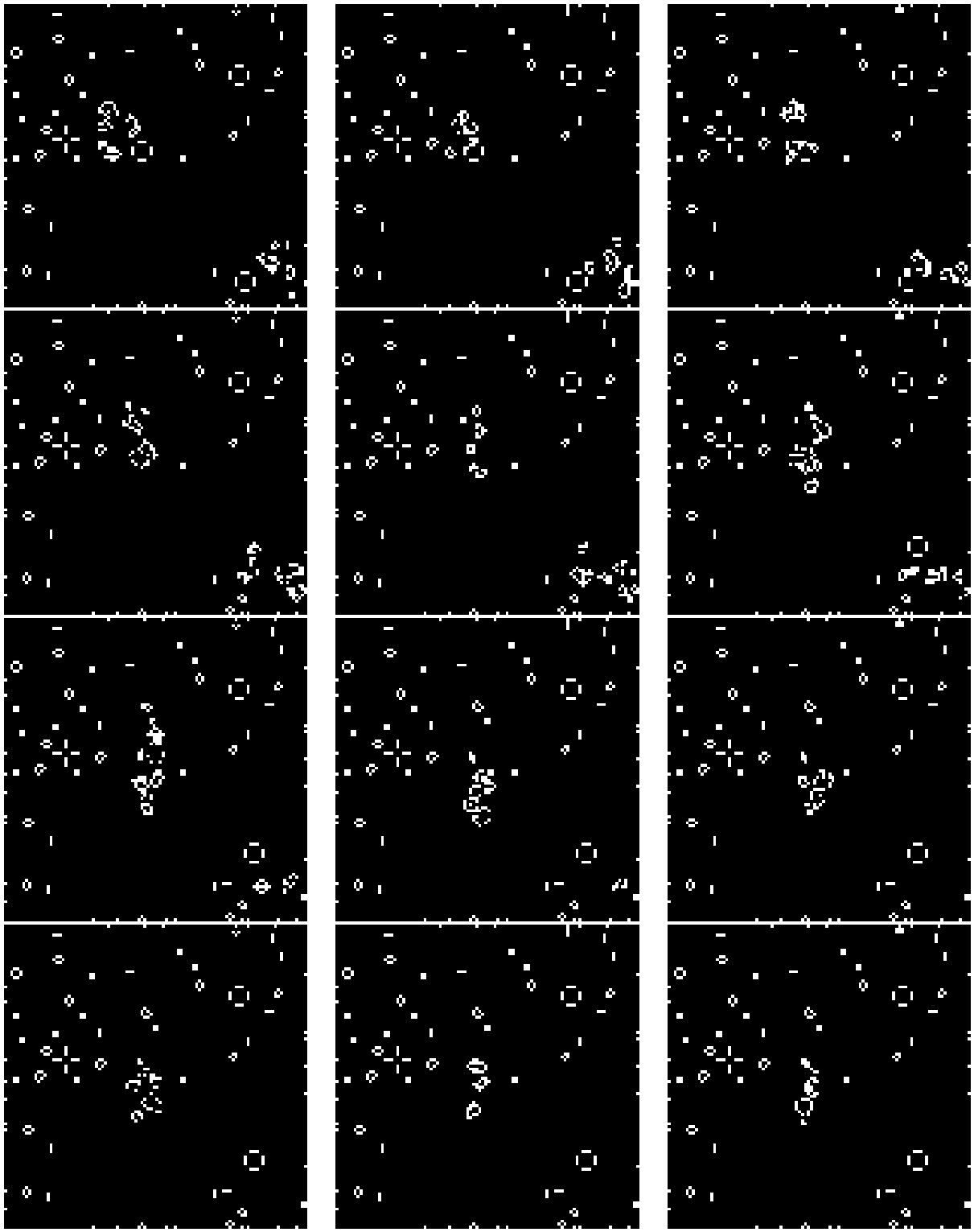

```
game_mat = evolve_n_R(game_mat)
par(mar=c(3.1,3.1,1.1,1.1))
magimage(game_mat)
```

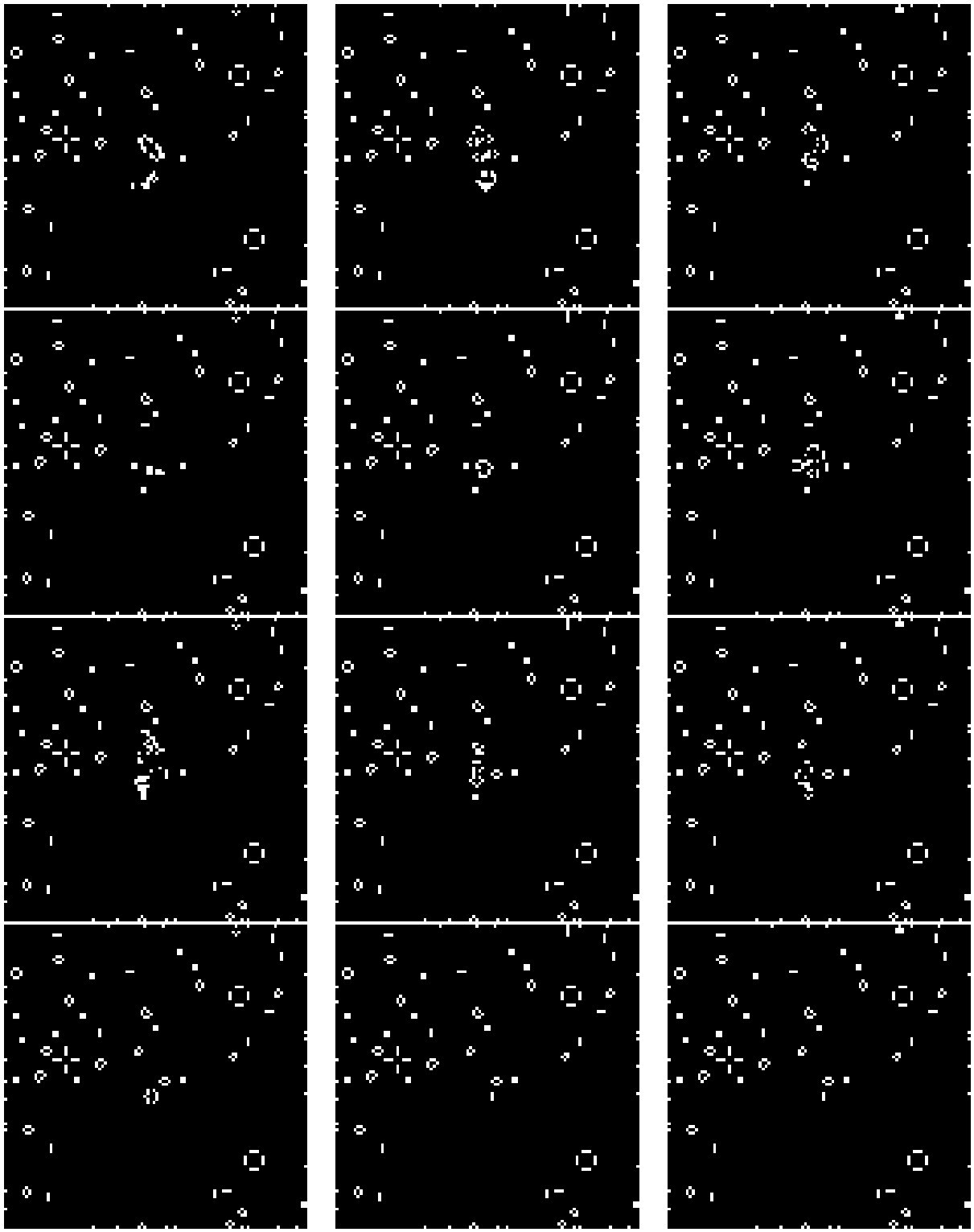


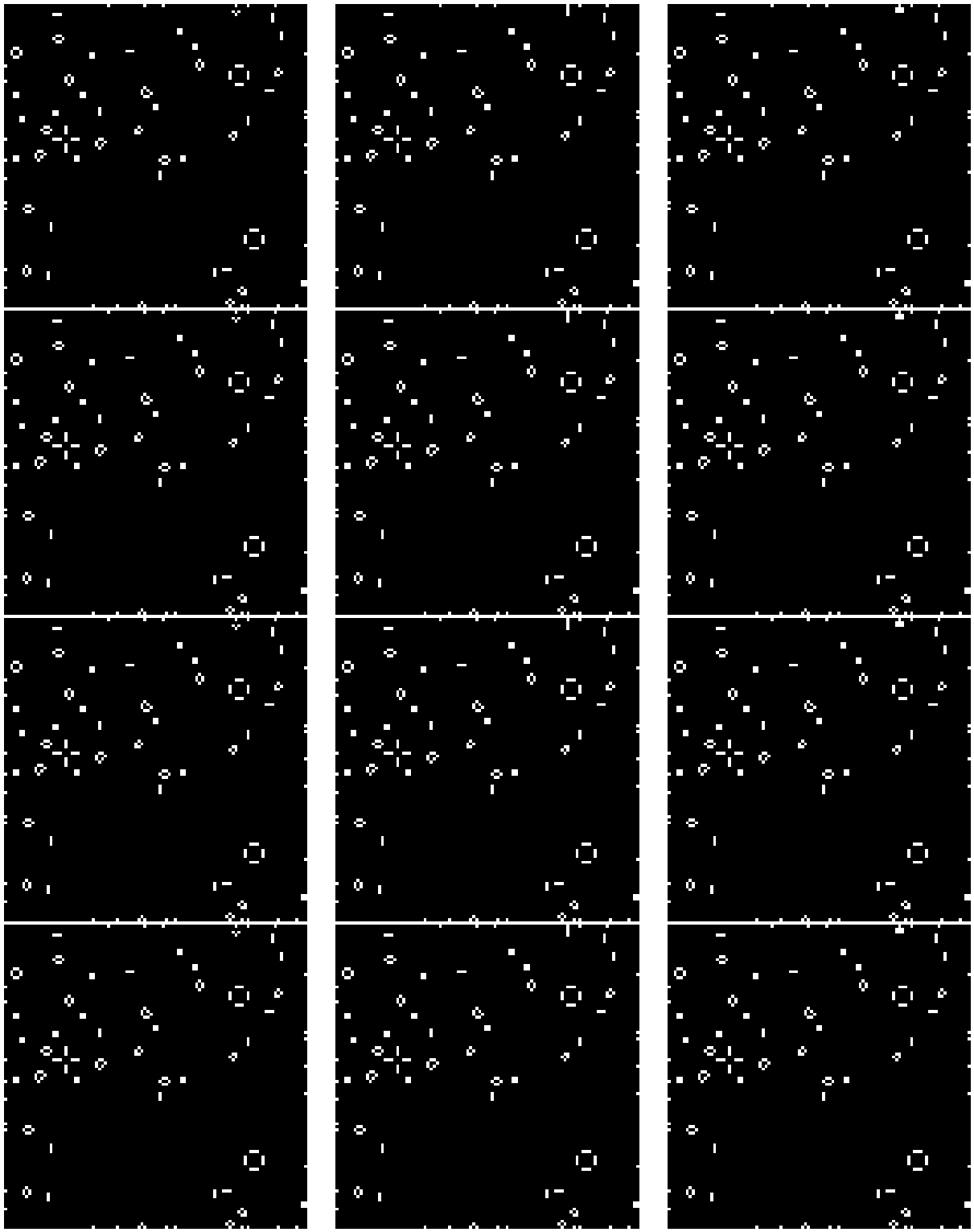
It is hard to see differences between single steps, so batches of 10 make it clearer:

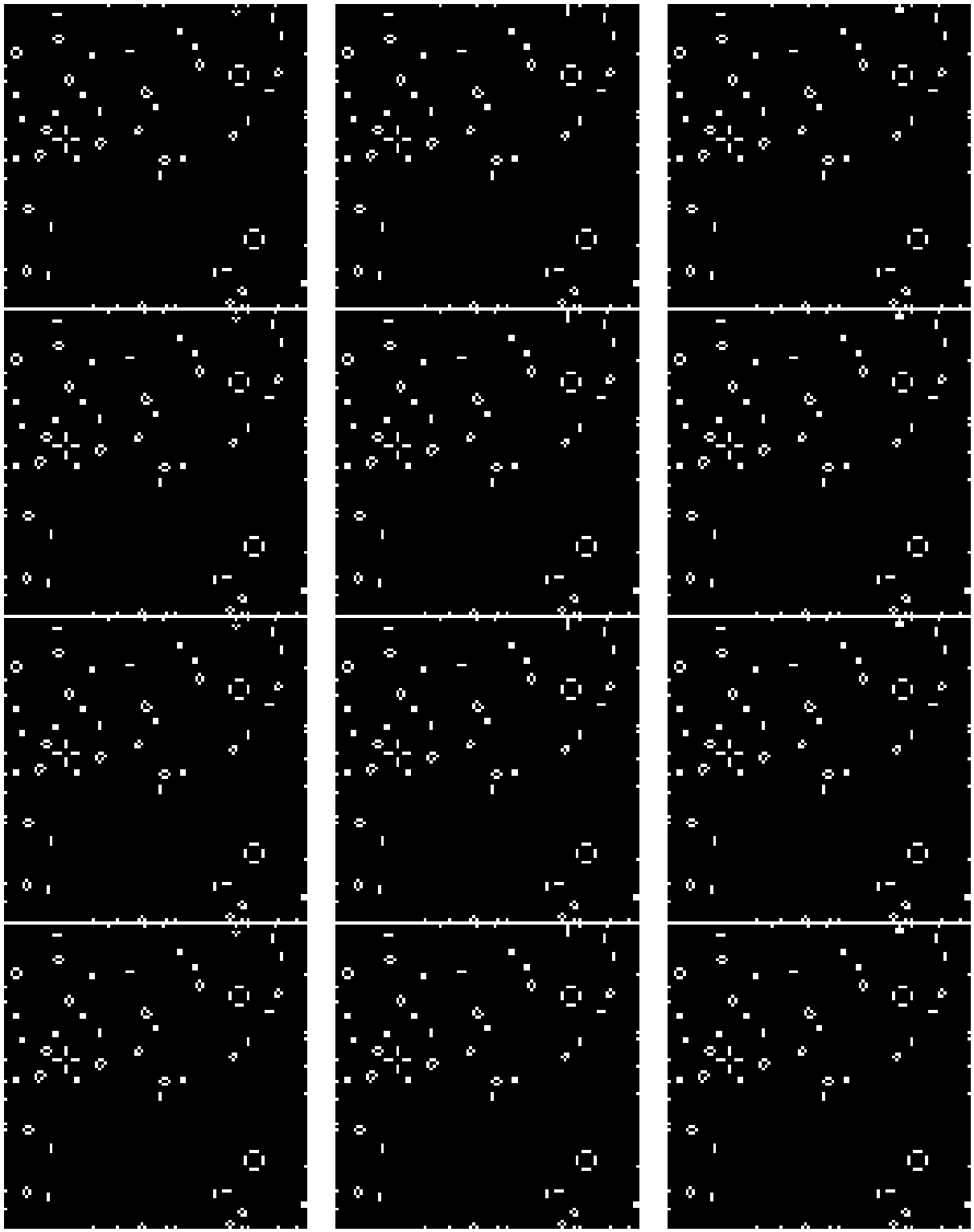
```
sum_game = sum(game_mat)
for(i in 1:1000){
  game_mat = evolve_n_R(game_mat)
  if(i  %% 10 ==0){
    par(mar=c(0.1,0.1,0.1,0.1))
    magimage(game_mat, axes=FALSE)
  }
  sum_game = c(sum_game, sum(game_mat))
}
```

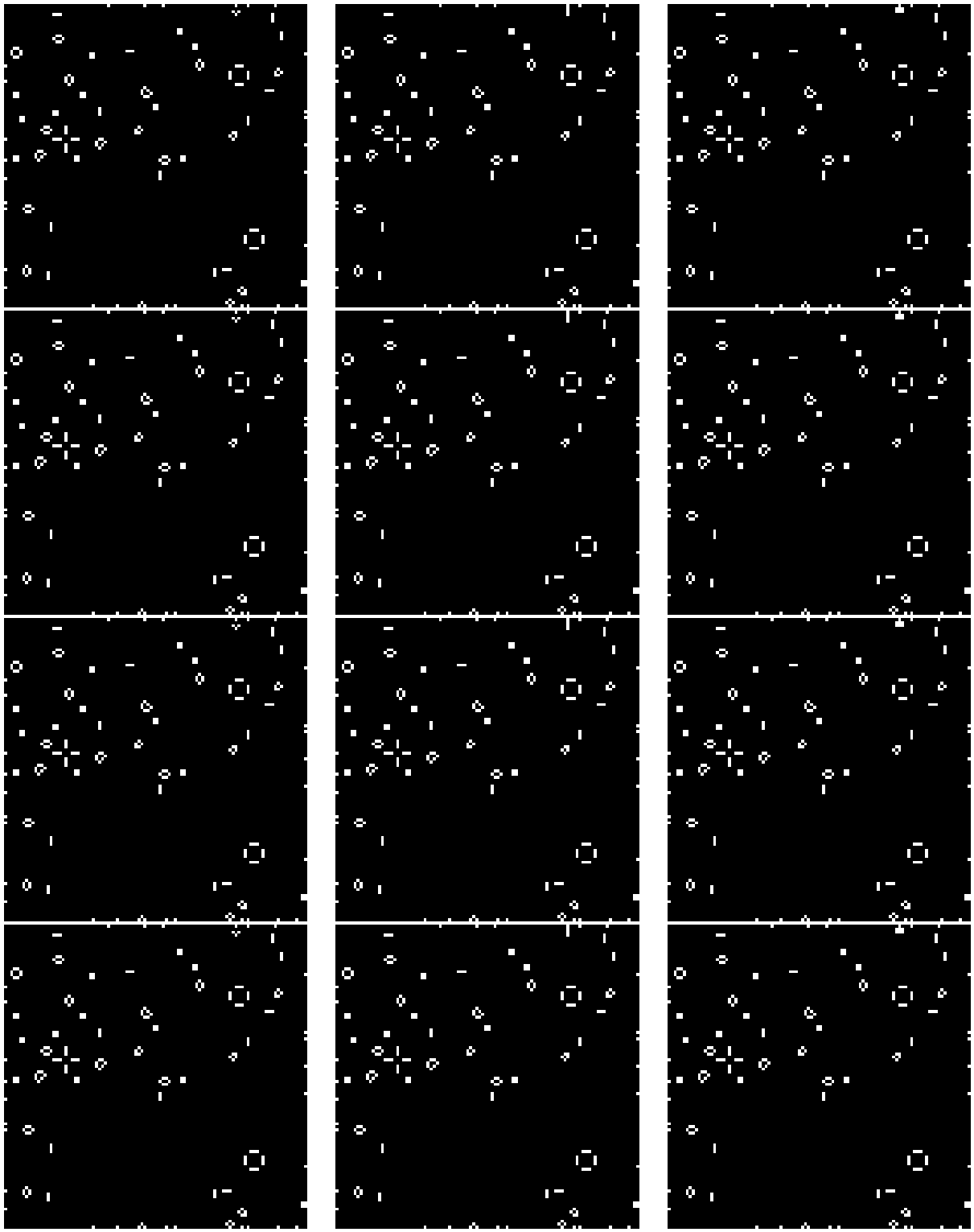


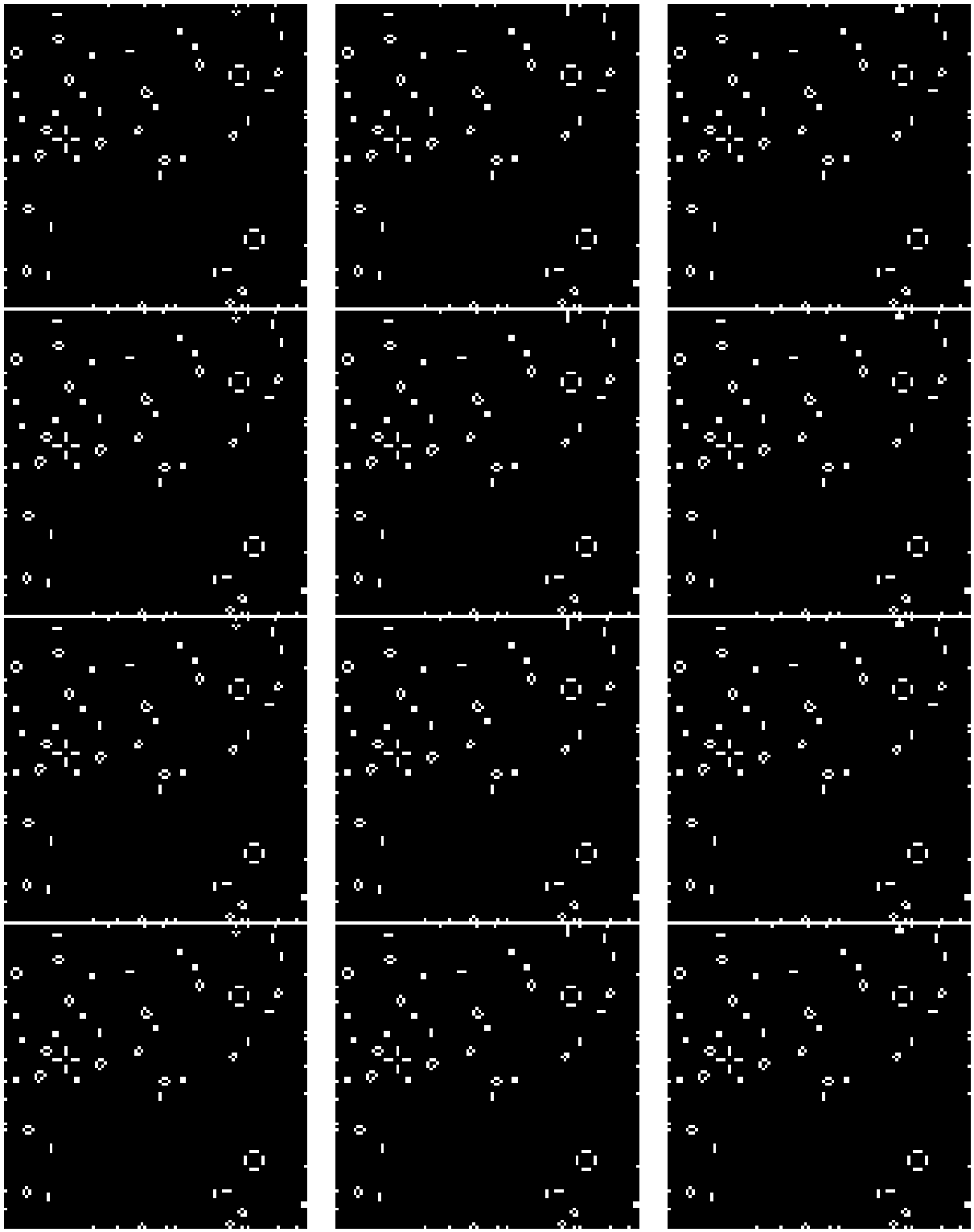


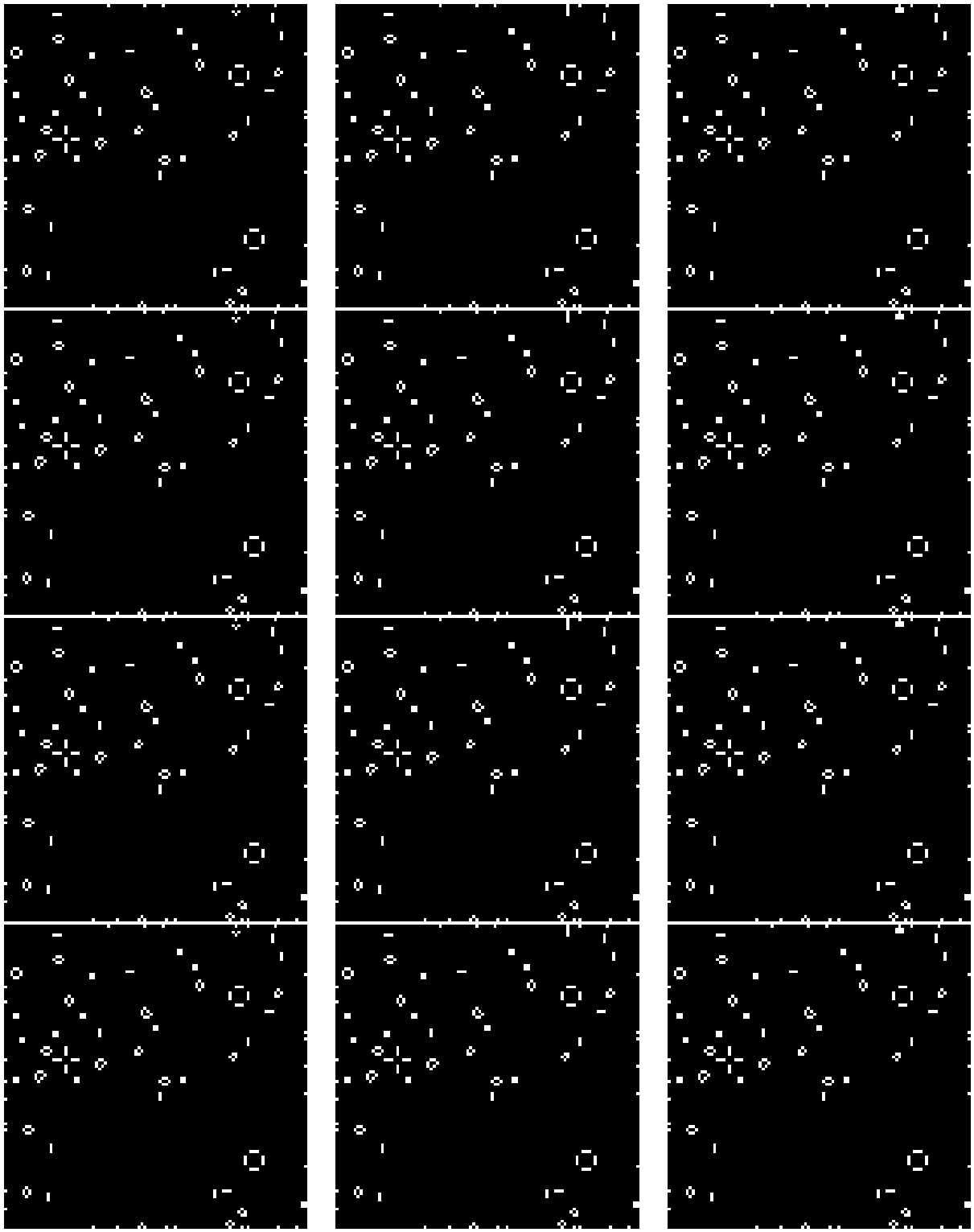


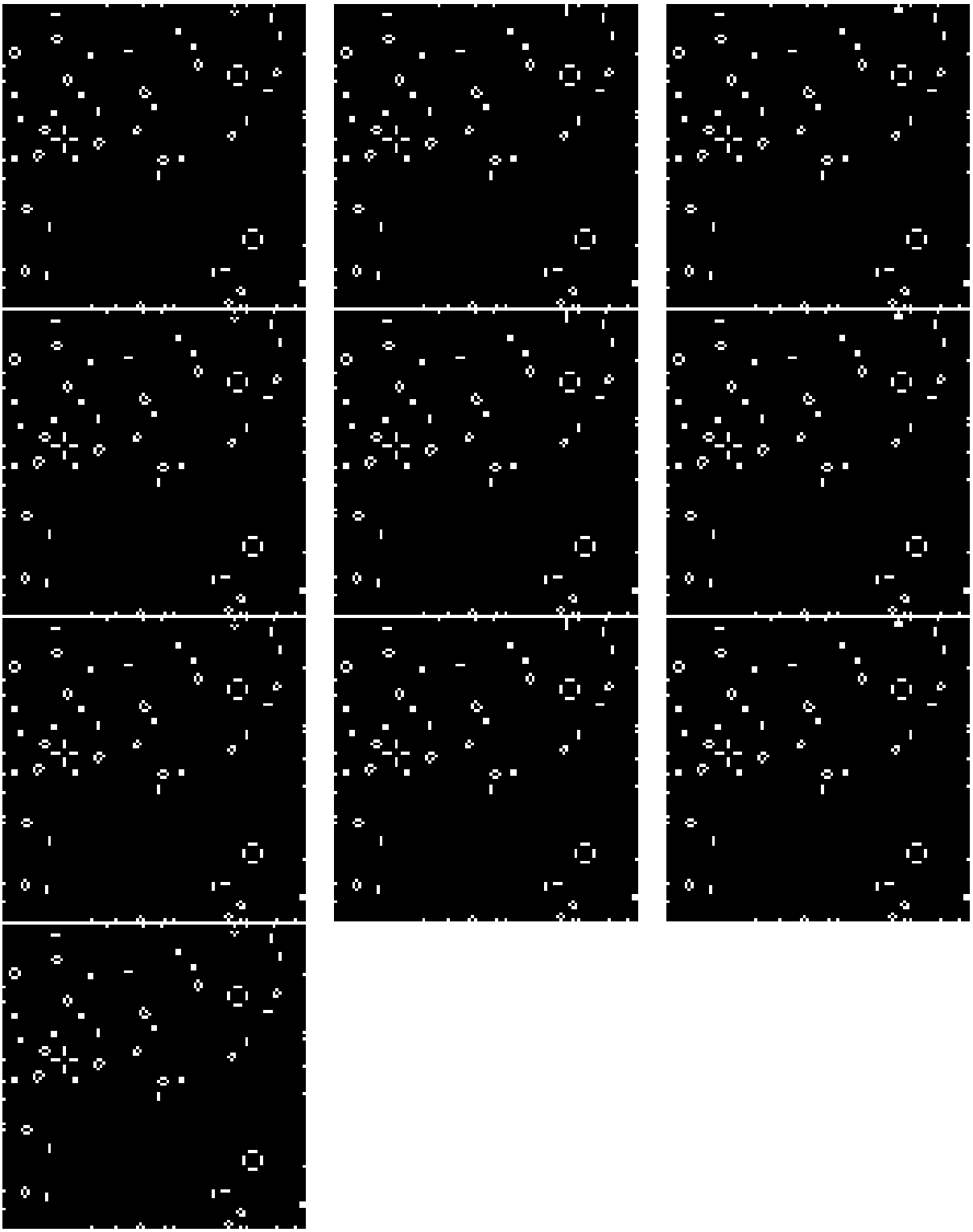






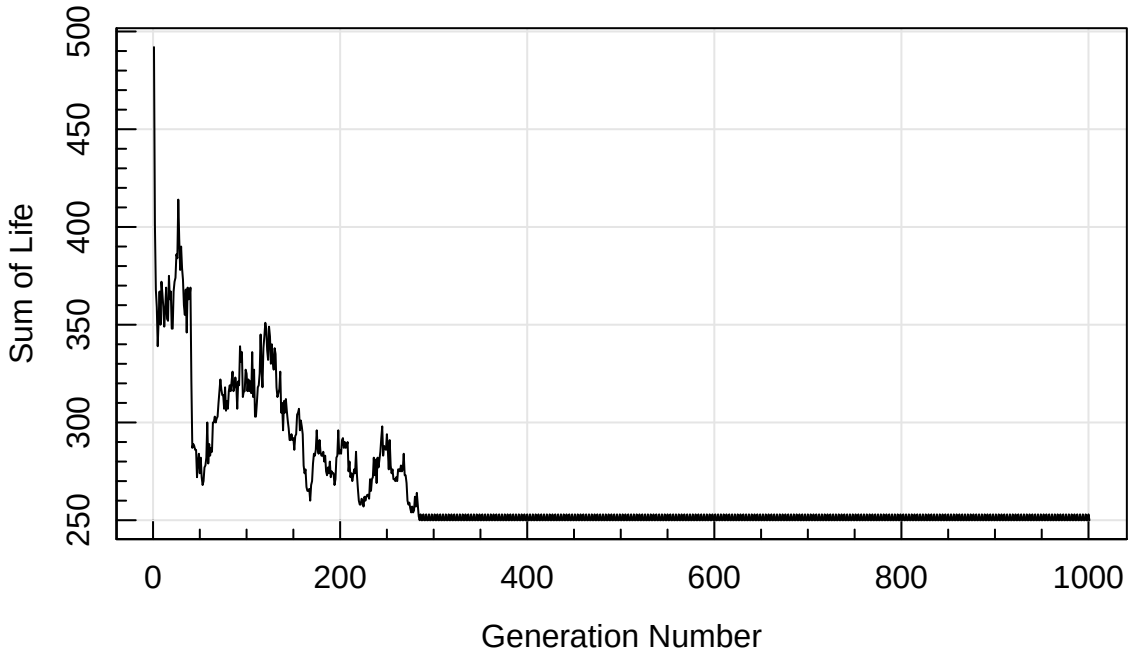






An interesting thing to note is that the simulation becomes quasi-stable, but it can create "life" (see the noise in the summation plot).

```
magplot(sum_game, type='l', xlab='Generation Number', ylab='Sum of Life')
```



With this random seed we find the population actually stabilises completely around generation 280.

Our R code is pretty slow and memory hungry however. We can easily speed this up with some Rcpp:

```
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::IntegerMatrix evolve_n_cpp(Rcpp::IntegerMatrix game_mat, int steps = 1){
  int nx = game_mat.nrow();
  int ny = game_mat.ncol();

  Rcpp::IntegerMatrix state_mat(nx-2,ny-2);

  for(int step = 0; step < steps; step++){
    // loop around all of our cells
    for (int i = 1; i < nx - 1; i++){
      for (int j = 1; j < ny - 1; j++){
        // loop around the 8 cells surrounding each cell
        for(int m = -1; m < 2; m++){
          for(int n = -1; n < 2; n++){
            if(m != 0 | n != 0){ // So we don't count the square we are in!
              // increment the surrounding cell count
              state_mat(i-1,j-1) = state_mat(i-1,j-1) + game_mat(i+m,j+n);
            }
          }
        }
      }
    }

    for (int i = 1; i < nx - 1; i++) {
      for (int j = 1; j < ny - 1; j++) {
        if(state_mat(i-1,j-1) < 2){
          game_mat(i,j) = 0;
        }
        if(state_mat(i-1,j-1) > 3){
          game_mat(i,j) = 0;
        }
        if(state_mat(i-1,j-1) == 3){
```

```
          game_mat(i,j) = 1;
        }
        state_mat(i-1,j-1) = 0;
      }
    }
  }

  return game_mat;
}
```

And we can so our usual profiling:

```
set.seed(666)
game_mat_R = matrix(0L, 1e2, 1e2)
game_mat_R[sample(prod(dim(game_mat_R)),1e3)] = 1L

set.seed(666)
game_mat_cpp = matrix(0L, 1e2, 1e2)
game_mat_cpp[sample(prod(dim(game_mat_cpp)),1e3)] = 1L

microbenchmark(
  evolve_n_R(game_mat_R, 10),
  evolve_n_cpp(game_mat_cpp, 10)
)
```

```
## Unit: microseconds
##                            expr      min       lq      mean    median        uq
##      evolve_n_R(game_mat_R, 10) 2540.647 2875.699 3648.8643 2987.3625 3822.4915
##  evolve_n_cpp(game_mat_cpp, 10)  432.222  435.912  465.1266  442.3285  463.9765
##         max neval
##  26835.853   100
##   1164.769   100
```

Here we find it is about 9 times faster in Rcpp. It would be more, but we wrote more explicit loops for our C++ solution which uses a lot less RAM. The R version has to keep ~10 versions of the matrix in memory for some amount of time- the input, the 8 offsets, and the sum of the 8 offsets. The C++ version just has the input and the summed offsets in memory.

It is sometimes easier to understand what is happening on the "infinite" grid by only creating life in the centre and seeing how it evolves out. Here is a random example that creates some interesting features after a short period. Evolve it to step 100 and then see what happens step by step.

```
set.seed(667)
game_mat_cpp = matrix(0L, 1e2, 1e2)
game_mat_cpp[cbind(sample(41:60,100,replace=TRUE),sample(41:60,100,replace=TRUE))] = 1L
```

See if you can find some other interesting random creations that do unexpected things (or if you have a really good intuition try making some of your own by hand).

Sword < Pen < Computer

Years ago (1976 in fact) biologist Robert May was interested in seeing if he could explain complex behaviour in population variation with a simple model. A simple model that has characteristics of breeding looks like the following:

$$x_{n+1} = \lambda x_n(1 - x_n),$$

where $0 > x_0 > 1$. x represents the fraction of the maximum population we currently have, and λ is the growth rate when the population is very small $x \sim 0$. The $(1 - x_n)$ represents population pressure that

starts to occur once we have grown too large, suppressing the population for the next generation.

In practice we usually set $x_0 = 0.5$ to start the iterations. The question is what population value of x will we stabilise to for a given λ? This equation is simple and seemingly pretty deterministic with sensible solutions only between $0 < \lambda < 4$ (prove this to yourself), but let's write a function to check values of x:

```
pop_converge = function(lambda, Nconv=100, Nkeep=10){
  x = 0.5
  for(i in 1:Nconv){x = lambda*x*(1-x)} # run Nconv times to get convergence
  output = rep(Nkeep)
  for(i in 1:Nkeep){output[i] = x; x = lambda*x*(1-x)} # keep the next Nkeep to check
  return(output)
}
```

Now let us check some values of λ:

```
pop_converge(0)
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
pop_converge(0.5)
```

```
##  [1] 1.549744e-31 7.748721e-32 3.874361e-32 1.937180e-32 9.685902e-33
##  [6] 4.842951e-33 2.421475e-33 1.210738e-33 6.053689e-34 3.026844e-34
```

```
pop_converge(1)
```

```
##  [1] 0.009395780 0.009307499 0.009220870 0.009135845 0.009052382 0.008970436
##  [7] 0.008889967 0.008810936 0.008733303 0.008657033
```

```
pop_converge(1.5)
```

```
##  [1] 0.3333333 0.3333333 0.3333333 0.3333333 0.3333333 0.3333333 0.3333333
##  [8] 0.3333333 0.3333333 0.3333333
```

```
pop_converge(2)
```

```
##  [1] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
```

```
pop_converge(2.5)
```

```
##  [1] 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6
```

That looks pretty nice- $\lambda < 1$ converges to 0, and above 1 we converge to a stable non-zero value. Let us keep going:

```
pop_converge(3)
```

```
##  [1] 0.6433012 0.6883943 0.6435228 0.6882036 0.6437382 0.6880180 0.6439477
##  [8] 0.6878372 0.6441516 0.6876610
```

This is odd, it seems we are bouncing around between two values, but in a quasi stable manner. In terms of describing populations this makes some sense: if you breed too efficiently (larger λ) you might overpopulate your environment one generation, see population pressure due to a lack of food etc and have a collapse, and then repeat etc. This transition to a stable population with a period of 2 is known as bifurcation.

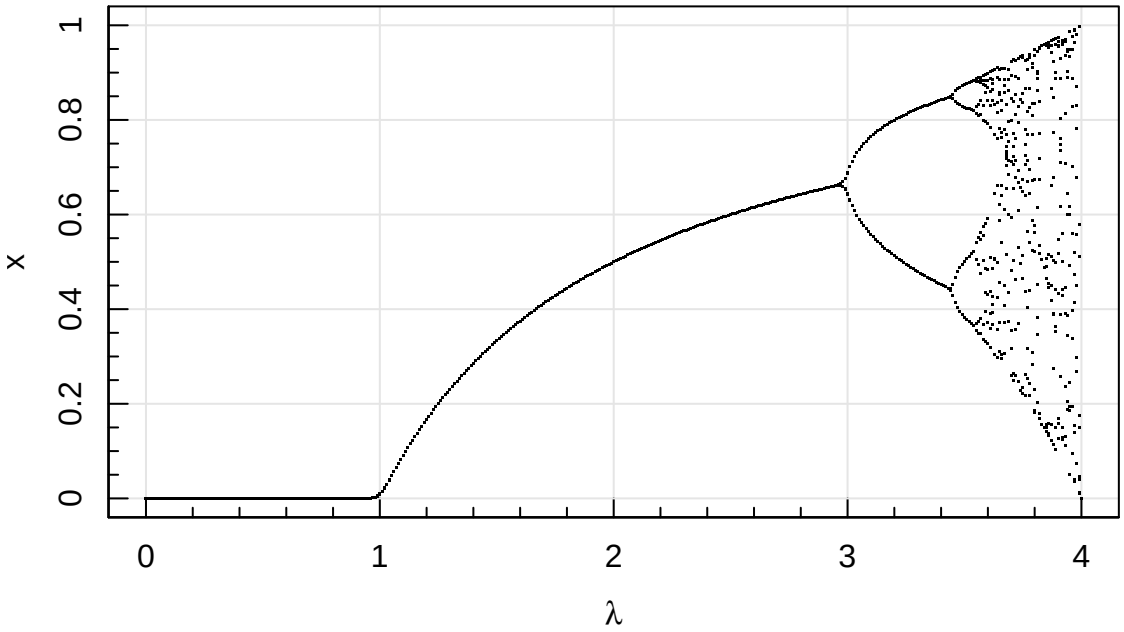
We can calculate the next increment:

```
pop_converge(3.5)
```

```
##  [1] 0.5008842 0.8749973 0.3828197 0.8269407 0.5008842 0.8749973 0.3828197
##  [8] 0.8269407 0.5008842 0.8749973
```

This is also repeating, but with period 4! This is another solution bifurcation. Did you expect that? Since we are seeing interesting population behaviour, let us plot the last 10 iterations as a function of λ:

```
magplot(NA, NA, xlim=c(0,4), ylim=c(0,1), grid=TRUE, xlab=expression(lambda), ylab='x')
for(i in seq(0,4, by=0.01)){
  points(rep(i,10), pop_converge(i), pch='.')
}
```

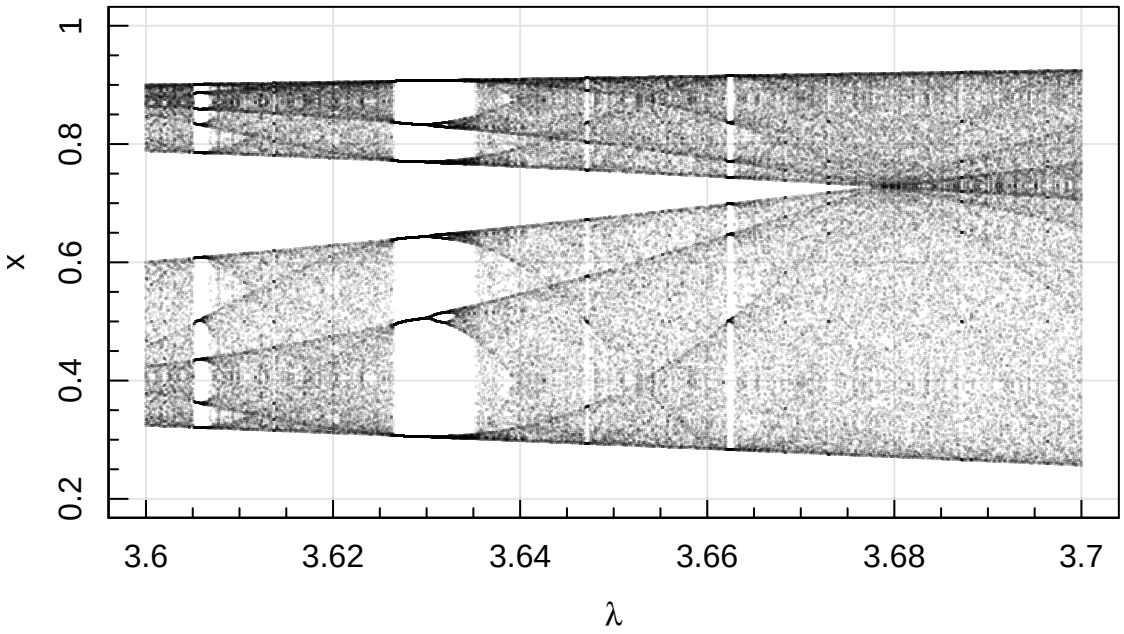


Nobody would expect this plot from the original equation. Almost every computer scientist who has created the above plot is surprised the first time they see this, and intrigued as to where it comes from. It is known as the 'Logistic Map', and it is one the simplest and earliest examples of chaotic systems being created by deterministic equations (arguably Conway's Game of Life got there first, but it is not nearly as chaotic in practice).

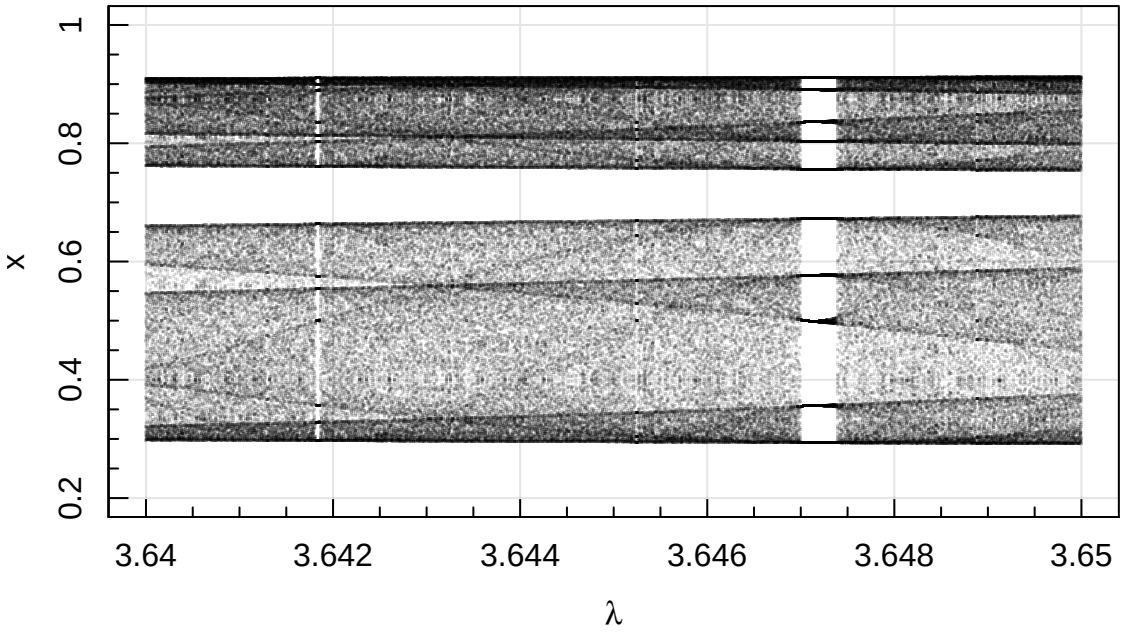
In your own time try increasing the density of points on the logistic map. What you will see will take you into the eye of infinity and chaos theory! The logistic map contains within it pretty much every fundamental constant in maths (π and e in the ratios between features; prime numbers; the golden ratio etc). It also led to the discovery of a new possible transcendental number (like π and e, so cannot be expressed using polynomial terms with real numbers) called Feigenbaum's constant.

You will find, as you ramp up the resolution of this plot, huge amounts of complex structure and new quasi-stable regions amidst the apparent chaos beyond $\lambda \sim 3.6$. It is very hard to reason why it has this form, but it is fun to explore it a bit.

Here is a nice zoom between $3.6 < \lambda < 3.7$ to try:



And now we will zoom again to between $3.64 < \lambda < 3.65$:



The stunning thing about this map is that it exists without regard to human knowledge or the physics of the Universe (it would be self-evident even if the Universe did not exist). It is trivial to describe the method to form it and to code this method, and yet the solution itself is not closed form, and can only be reached numerically with a computer. It is also fractal in nature with infinite structure at infinite levels of inspection (try making a high resolution image of a sub-region).

Get to the Point!

It is easy to see how some iterative series converge and some diverge. Consider this one:

$$x_{n+1} = x_n^2$$

You should be able to see easily that if out initial $|x_0| < 1$ then the series will converge, and do so increasingly rapidly for larger initial values. The convergent value that meet the criterion stated will eventually tend to 0, e.g.:

```
x = 0.9
for(i in 1:100){x = x^2}
x
```

```
## [1] 0
```

In fact the above iteration series can be easily solved with $x_0^{2^n}$ (check this yourself), and it is this double exponent that means it will converge or diverge so quickly.

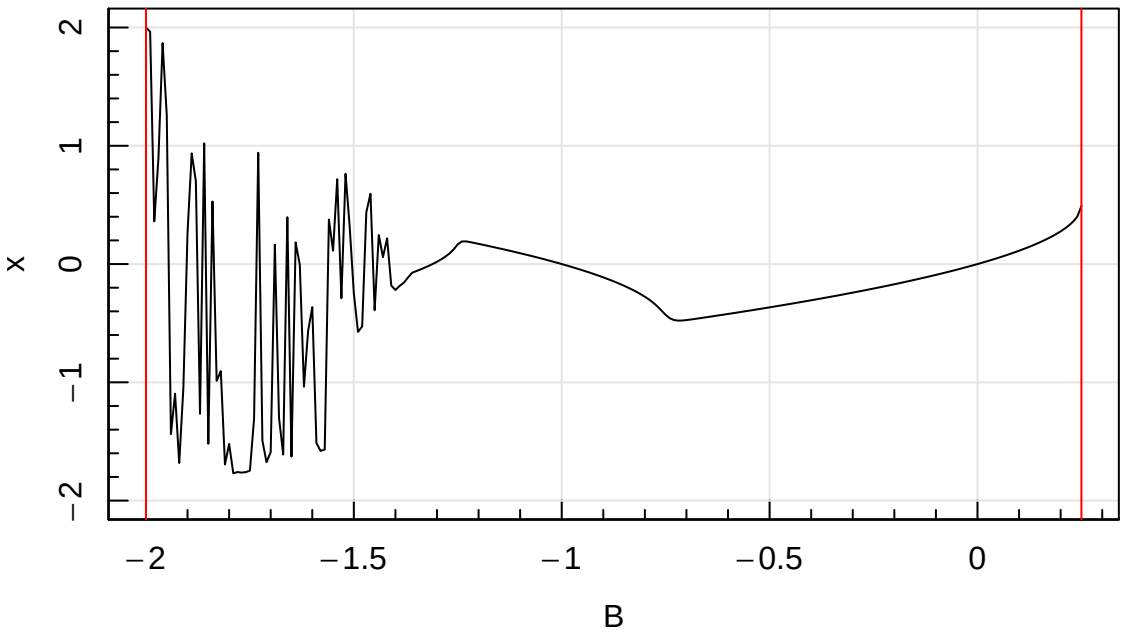
Making things a bit more interesting, we can check the following:

$$x_{n+1} = x_n^2 + B,$$

where B is a real number. Now, if we start at $x_0 = 0$, for what values of B does this series converge?

```
Bout = rep(NA,801)
j = 1
for(B in seq(-4,4,by=0.01)){
  x = 0
  for(i in 1:100){
    x = x^2 + B
  }
  Bout[j] = x
  j = j + 1
}
```

```
magplot(seq(-4,4,by=0.01), Bout, type='l', grid=TRUE, xlab='B', ylab='x', xlim=c(-2,0.25), ylim=c(-2
abline(v=c(-2, 1/4), col='red')
```



The limits before we diverge to infinity turn out to be $-2 < B < 1/4$ (it is easy to show this yourself).

We can ask a similar question for complex numbers too, where $i = \sqrt{-1}$ and any complex number is then $C = A + Bi$. We start at $x_0 = 0 + 0i$,

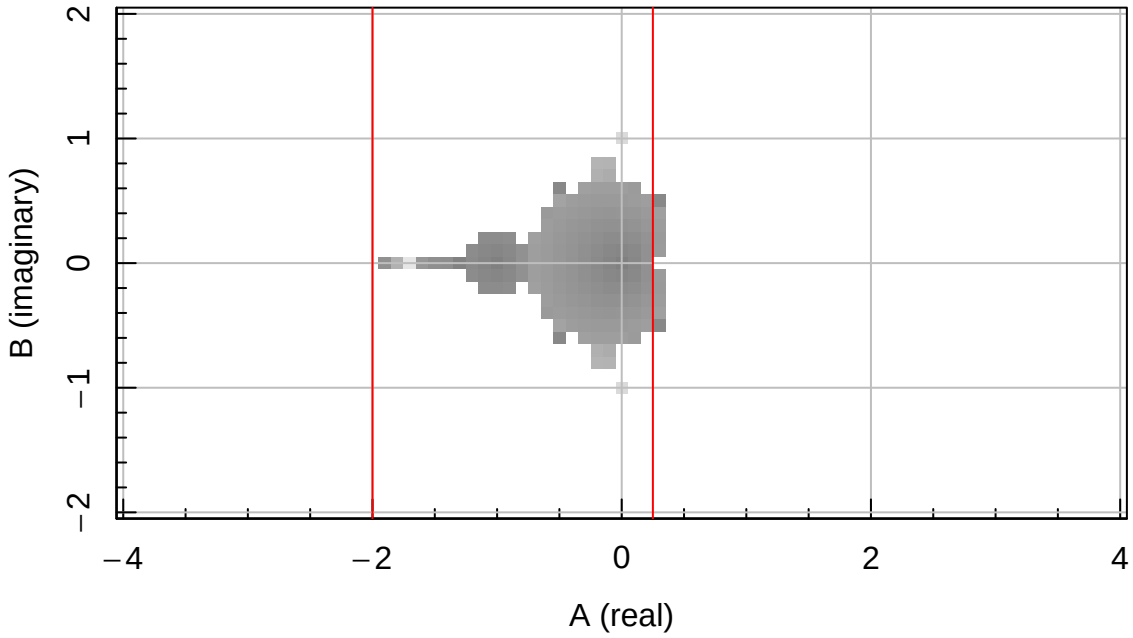
```
IterC_N = function(C=1/4 + 0i, N=1e2){
  x = 0
  for(i in 1:N){
    x = x^2 + C
  }
  return(x)
}
```

We can explore this on a grid between [-2,2] and [-2i,2i]:

```
checkgrid = expand.grid(seq(-2,2,by=0.1),seq(-2,2,by=0.1))
result = rep(NA,dim(checkgrid)[1])
for(i in 1:dim(checkgrid)[1]){
  result[i] = Mod(IterC_N(C=checkgrid[i,1] + checkgrid[i,2]*1i))
}
```

If we make a plot of this, we will find that every cell that is white will be diverging to infinity, and all cells that are grey are converging to some value (or sequence of values).

```
magimage(x=seq(-2,2,by=0.1), y=seq(-2,2,by=0.1),
        z=matrix(result,41), zlim=c(-2,2), magmap=FALSE, grid=TRUE,
        xlab='A (real)', ylab='B (imaginary)')
abline(v=c(-2, 1/4), col='red')
```



That looks pretty odd! It was discovered by a number of people in the late 1970s and is usually referred to as the **Mandelbrot Set** (giving all the credit to Benoit Mandelbrot). Notice that on the real line we get the result we expect (that $-2 < B < 1/4$ is converged), but once we enter the complex plane we see some very *complex* behaviour. Part of assignment 1 will be to re-write this code that produces the above plot in **Rcpp**, but it should be obvious by now why **R** will be slow at tackling a problem of this type: nested *for* loops are never fast in any interpreted language, including **R**.

Big Data and Parallel Processing

R has a lot of support for 'big data' including sparse matrices and lazy loading. It can also interface with traditional databases (as we saw last lecture with **SQL** and **Arrow** Datasets.

A useful aspect for astronomers is how easily you can do 'embarrassingly' parallel problems, like calculating the similar properties (e.g. velocity dispersions) or similar types of subsets of data (e.g. groups of galaxies). For this we need the extra packages **foreach** and **doParallel**).

First we tell our parallel back-end to use 6 processors since the machine I am using happens to have 8, and it is a good idea to leave one or two free for other OS tasks. You can check how many you have with the *detectCores* function in the **parallel** package.

```
registerDoParallel(cores=6)
```

Without getting into the detail, here we are doing something like measuring the velocity dispersions of a galaxy group/clusters (where the velocity has a standard deviation (σ) or 1,000 km/s:

```
system.time(for(i in 1:100){sd(rnorm(100000,sd=1e3))})
```

```
##    user  system elapsed
##   0.226   0.005   0.230
```

```
system.time(foreach(i=1:100)%dopar%{sd(rnorm(100000,sd=1e3))})
```

```
##    user  system elapsed
##   0.253   0.038   0.069
```

On my machine the parallel code runs about 4 times faster (you will never quite see the speed-up approach the number or cores due to parallel overheads).

Big Data and Parallel Processing

A fairly common data processing problem is one where you want to process a large amount of data (that requires, say, half of your system memory), but you have a multicore machine and would like to make use of the many free threads. Without careful handling the standard way this is done in high-level languages is to copy all of this data into each thread, which is this case obviously means you could only run on two threads.

There are a few packages that offer a solution to this, but the most popular is the aptly named **bigmemory** package that is designed to work well with **foreach**. For reasons that are quite technical, we have to use a different parallel back-end to process our **bigmemory** object called **SNOW**.

The other steps are to create a **bigmemory** class object, describe it (basically the output of this is a pointer to the object) and then within our parallel task we have to attach our virtual matrix via this pointer.

```
n = 10000
m = 10000
c = matrix(runif(n*m),n,m)
# convert it to bigmatrix
x = as.big.matrix(matrix(runif(n*m),n,m))
# get a description of the matrix
mdesc = describe(x)
# create the required connections
cl = makeCluster(6)
registerDoSNOW(cl)
## 1) No referencing
out = foreach(i = 1:1e3, .combine=c, .packages='bigmemory') %dopar% {
  # attach the pointer reference
  t = attach.big.matrix(mdesc)
  #execute our correlation test operation on the matrix
  return(cor.test(t[,i], t[,i+1])$p.value)
}
```

If this work correctly then the following should be a uniform distribution between 0 and 1 (do not worry about why this is true yet, we will get to that):

```
maghist(out)
```

```
## Summary of used sample:
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.001071 0.252252 0.516805 0.508433 0.757661 0.999882
```

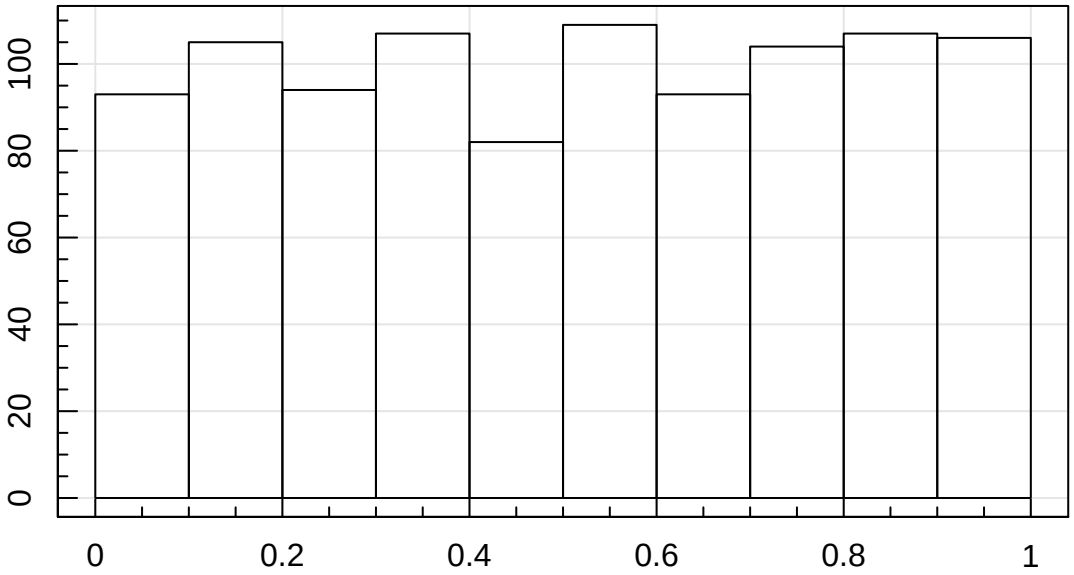
```
## Pop Std Dev: 0.28947
```

```
## MAD: 0.36813
```

```
## Half 16-84 Quan (1s): 0.33947
```

```
## Half 02-98 Quan (2s): 0.47756
```

```
## Using 1000 out of 1000
```



This looks good to me!

So above we shared our 800 MB matrix across multiple cores rather than making a copy each time we spawned a process (where we ran 6 in this case). Coding this up is always a bit fiddly since you have to remember to use a different back-end and run the *describe* and *attach* functions, so I only do this when really necessary and suffer a few minutes of Googling to remind myself how to do it (but this example should act as a good template too).

If you check the options within *as.big.matrix* you will find it is possible to create the object on disk rather than in memory. This serves as a mechanism to create larger than memory objects that you are still able to process across multiple threads, should you need to do this.