Chaos and fractals final project

Phileas Dazeley Gaist

2/28/2022

Contents

Introduction: A term's worth of code experiments with chaos and fractals

This winter term, I spent a long time in Rstudio putting together code experiments and visualisations of concepts from the Chaos and Fractals course. I could say I did this to challenge myself and to advance my coding literacy, but the truth is I was just having a lot of fun, and I found as a side effect that my coding examples helped me conceptualise and learn the class topics. Some of the programs I wrote this term felt consequential enough that I decided to share them on GitHub and social media. I hope others will try and play around with them in creative ways. In this document, I will explore a few of the programs I wrote throughout this term, how I experimented with them, what prompted me to write them, and how I think they might be useful for others.

Graphical iteration plotter

Description and contextualisation

Graphical iteration is (as the name suggests), a graphical way to visualise the orbits of a function for different initial conditions. It was one of the first topics we discussed in class, and most importantly, it sometimes resulted in squiggly lines. As we explored the topic by hand-iterating on different functions, I began to wonder whether I could find a program to do the task for me. Although I easily found several online graphical iteration tools for the logistic equation, I could not find one that was general enough that the use could specify their own function as well as its parameters and initial conditions; a shame, because I was really curious to try it on functions including a $\sin(x)$ term, which I could have plotted by hand, but which I felt rather too unconfident to try. I decided to try writing my own general graphical iterator program in R (although I'm usually more of a python and C# programmer, R has a considerable advantage when it comes to plotting thanks to the tidyverse library, which includes the excellent ggplot).

While this is a pretty simple program, I did not find an equivalent to it online. This program has the advantages that it is rather simple to set up, quite fast in my testing experience, and easy to run, as it depends only on the tidyverse library, which is free and open source. I imagine that this tool could be useful in education settings for demonstration purposes, or for analysis, as the program also produces data frames of the plotting data. I am hopeful that this tool might be useful to students in university contexts. I have seen graphical iteration used in economics courses at the university level, so perhaps this tool could help economists in training, for instance, to model poverty traps.

Setup

This is the part of the program in which the user can specify initial conditions, number of iterations, plot axis limits, and the function to plot and iterate on.

```
# set your initial condition and desired number of iterations:
x_0s <- c(7, 3.6, 4.43)
N <- 100

# set the iteration plot x axis range (lower and upper bounds):
x_min <- 0; x_max <- 8
y_min <- -2; y_max <- 8

use_custom_range_x <- FALSE
use_custom_range_y <- FALSE

# declare your function here:
func <- function(x){
  return(-2 * sin(x) + x) # function goes here
}
```

The nitty-gritty

This is where the bulk of the program lives.

- `get_function_data()` gets the data required to plot the user-specified function.
- `graphical_iterator()` gets the segment data required to plot the user-specified orbits on top of the user-specified function plot.
- `get_time_series_data()` gets the data required to plot a time series of the orbits (optional)

```
get_function_data <- function(range = c(-1, 1), steps = 100){

  steps_multiplier <- (range[2]-range[1])/10
  if(steps_multiplier < 1){steps_multiplier <- 1}
  # adds steps to get data for depending on the number of 10s
  # in the specified plot x range

  x <- seq(from = range[1], to = range[2], length.out = steps * steps_multiplier)

  y <- array(dim = steps * steps_multiplier)
  for(i in 1:length(x)){
    y[i] <- func(x[i])
  }

  return(data.frame(x = x, y = y))
}
```

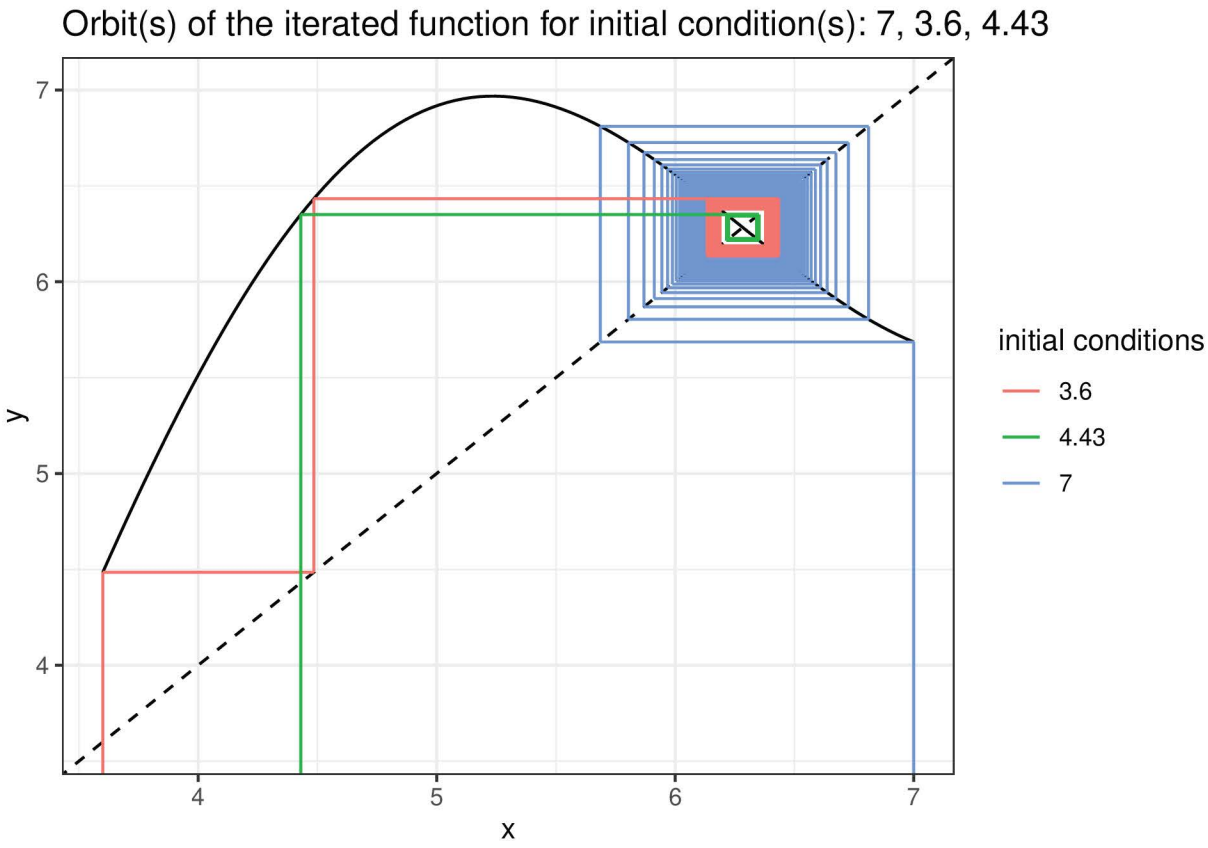
```r
graphical_iterator <- function(x_0s, N = 100){

  segments <- data.frame()
  for(i in x_0s){

    start <- i
    vert <- FALSE

    x_0 <- rep(i,times=1+(N*2))
    xstarts <- c(start)
    ystarts <- c(y_min)
    xends <- c(start)
    yends <- c(func(start))

    # iteratively get the coordinates of the next segment points
    for(i in 1:(2 * N))
      # range = 2 * N because every step will be described by two segments
    {
      # if the last segment was vertical, the next must be horizontal
      if(vert){
        xstarts <- c(xstarts, start)
        ystarts <- c(ystarts, start)
        xends <- c(xends, start)
        yends <- c(yends, func(start))
        vert <- FALSE
      }
      else{
        xstarts <- c(xstarts, start)
        ystarts <- c(ystarts, func(start))
        xends <- c(xends, func(start))
        yends <- c(yends, func(start))
        vert <- TRUE
        start <- func(start) # update start value
      }
    }
    segments <- rbind(segments, data.frame(x_0s = x_0, xstarts, ystarts,
                                           xends, yends))
  }
  return(segments)
}

get_time_series_data <- function(x_0s, N = 100){
  trajectories <- data.frame()

  for(i in x_0s){
    x_t <- i
```

```
    x_0 <- rep(i,times=N+1)
    n <- 0:N

    trajectory <- c(x_t)

    for(t in 0:(N-1)){
      x_t <- func(x_t)
      trajectory <- c(trajectory, x_t) # add x_t_1's value to the trajectory vector
    }
    trajectories <- rbind(trajectories,
                          data.frame(x_0s = x_0, ns = n, trajectories = trajectory))
  }
  return(trajectories)
}

cobweb_trajects <- graphical_iterator(x_0s = x_0s, N = N)

if(use_custom_range_x == FALSE){
  x_min <- min(cobweb_trajects$xstarts); x_max <- max(cobweb_trajects$xends)
}
if(use_custom_range_y == FALSE){
  y_min <- min(cobweb_trajects$xstarts); y_max <- max(cobweb_trajects$xends)
}

plot_data <- get_function_data(range = c(x_min,x_max)) # gets the plotting data

trajectories <- get_time_series_data(x_0s = x_0s, N = N) # gets the time-series data
```

Plots

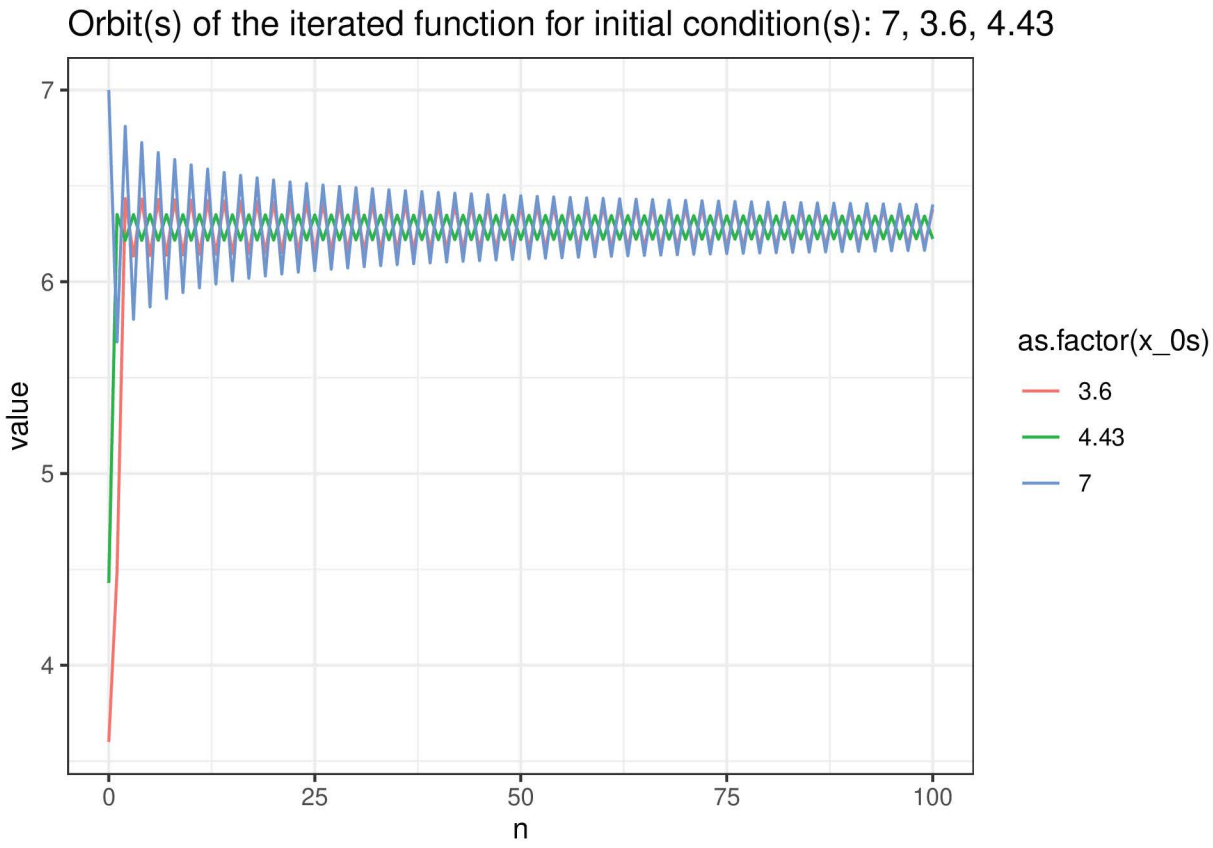
Graphical iteration plot:

```
plot_data %>%
  ggplot(aes(x, y)) +
  geom_line(colour = "black") +
  geom_abline(linetype = "dashed") +
  geom_segment(data = cobweb_trajects, aes(x = xstarts, y = ystarts, xend = xends,
                                           yend = yends, colour=as.factor(x_0s))) +
  coord_cartesian(xlim = c(x_min, x_max), ylim = c(y_min, y_max)) + theme_bw() +
  labs(title=paste0("Orbit(s) of the iterated function for initial condition(s): ",
                    toString(x_0s)),
       colour="initial conditions")
```



Bonus time-series plot:

```
trajectories <- get_time_series_data(x_0s = x_0s, N = N) # gets the time-series data

# trajectory plot
trajectories %>%
  ggplot(aes(ns, trajectories, colour=as.factor(x_0s))) +
  geom_line() + labs(x="n") +
  labs(y="value",
       title=paste0("Orbit(s) of the iterated function for initial condition(s): ",
                    toString(x_0s))) + theme_bw()
```



Logistic equation example

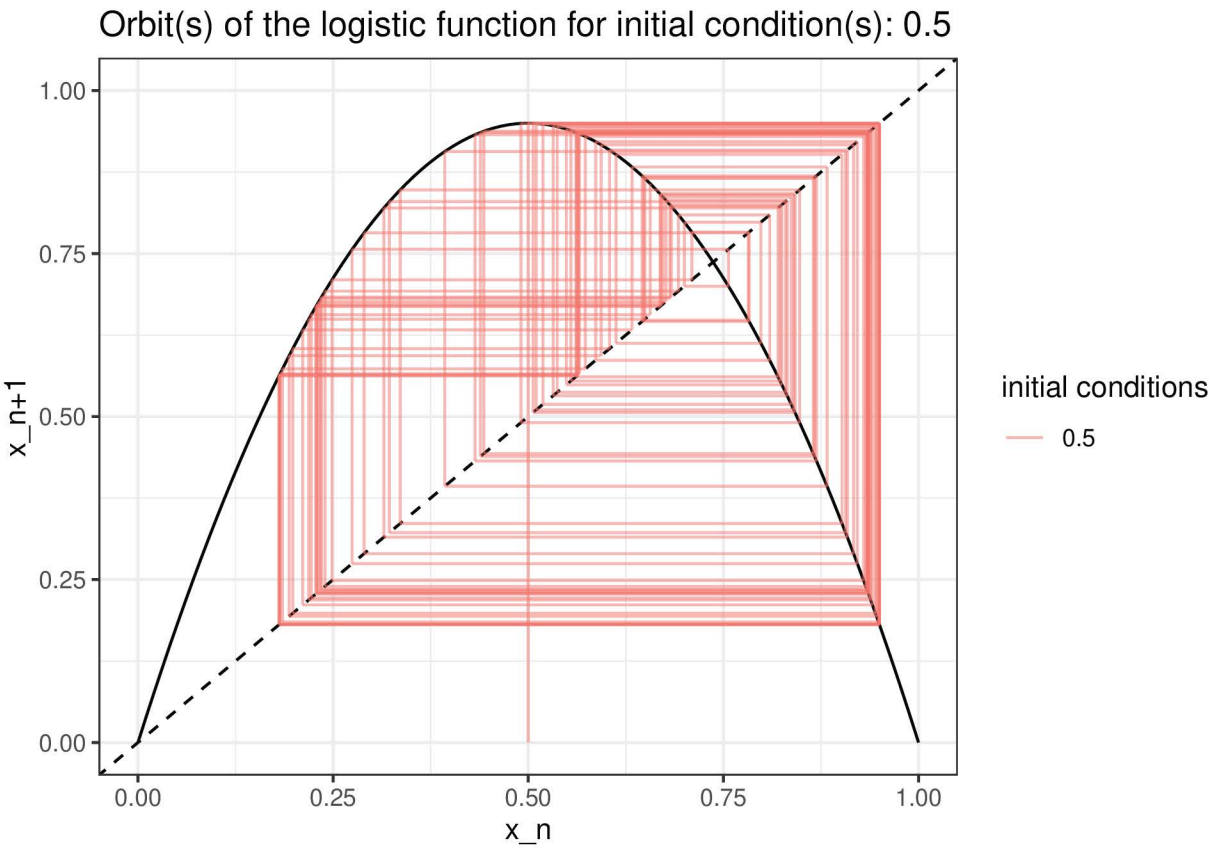
Set-up:

```
x_0s <- c(0.5)
N <- 100

# declare the logistic equation as our function
func <- function(x, r=3.8){
  return(r*x*(1-x))
}

# set graphical iteration plot limits
x_min <- y_min <- 0; x_max <- y_max <- 1
```

Graphical iteration plot:

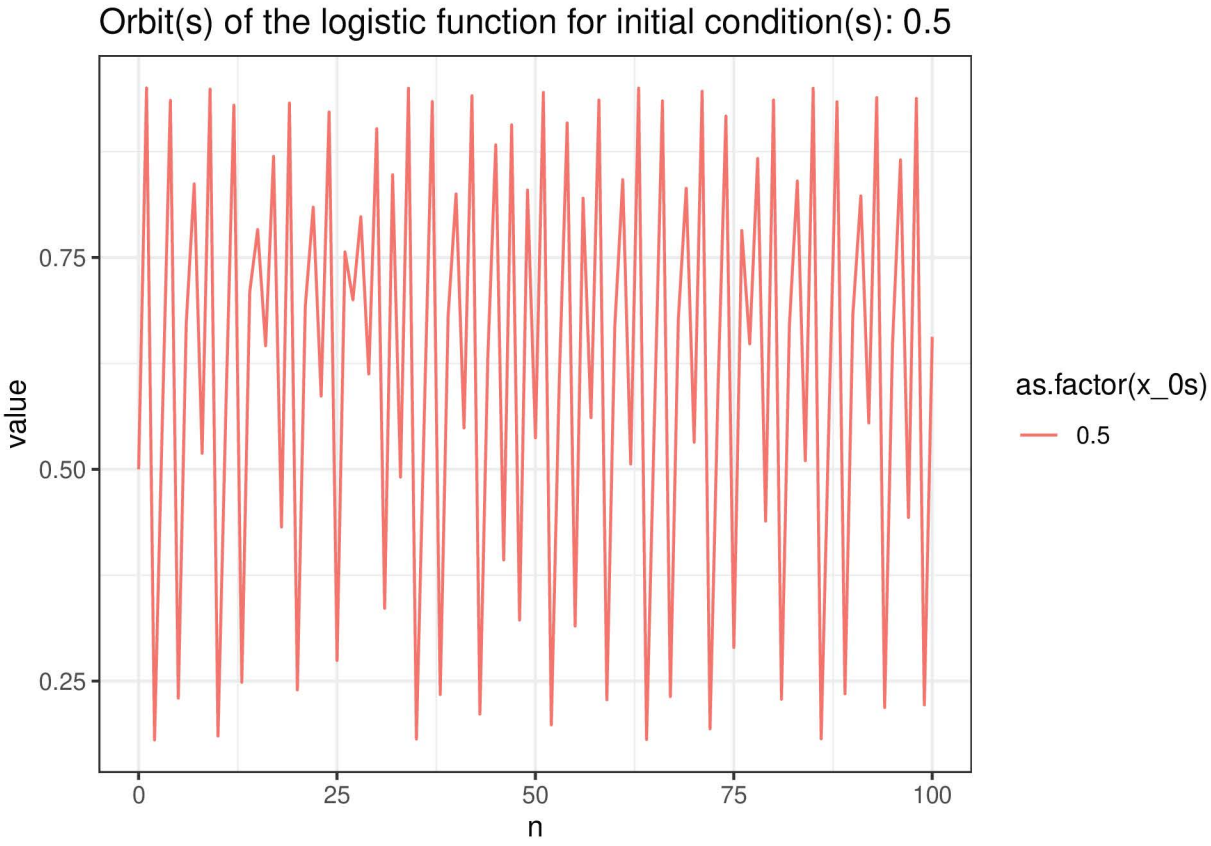
```
cobweb_trajects <- graphical_iterator(x_0s = x_0s, N = N)

plot_data <- get_function_data(range = c(x_min,x_max)) # gets the plotting data

plot_data %>%
```

```
  ggplot(aes(x, y)) +
  geom_line(colour = "black") +
  geom_abline(linetype = "dashed") +
  geom_segment(data = cobweb_trajects, aes(x = xstarts, y = ystarts, xend = xends,
                                           yend = yends, colour=as.factor(x_0s)),
               alpha=0.5) +
  coord_cartesian(xlim = c(x_min, x_max), ylim = c(y_min, y_max)) + theme_bw() +
  labs(title=paste0("Orbit(s) of the logistic function for initial condition(s): ",
                    toString(x_0s)),
       colour="initial conditions", x="x_n", y="x_n+1")
```



Bonus time-series plot:

```
trajectories <- get_time_series_data(x_0s = x_0s, N = N) # gets the time-series data

# trajectory plot
trajectories %>%
  ggplot(aes(ns, trajectories, colour=as.factor(x_0s))) +
  geom_line() + labs(x="n") +
  labs(y="value",
       title=paste0("Orbit(s) of the logistic function for initial condition(s): ",
                    toString(x_0s))) + theme_bw()
```



Final state (bifurcation) diagram of the logistic equation plotter

Description and contextualisation

The final state diagram of the logistic map shows the set of points reached on an orbit of the discrete logistic equation $x_{n+1} = rx_n(1 - x_n)$ for any growth rate value r after n iterations. I imagine to qualify as a perfect final state diagram, n should approach infinity, but since I only have limited computation time, I have opted for finite n values in the examples below.

Although there already exist many examples of bifurcation diagram plotting tools in many programming languages (including R), I found many of them tedious to use, or limited for my purposes. Many bifurcation diagram plotting tools obtain the bifurcation diagram data through processes that do not directly involve the logistic equation, and result in plots with x and y axes that do not line up with those one would expect from the bifurcation diagram of the logistic map. I could not find a version of this tool which plotted the bifurcation diagram in R using iteration of the logistic equation directly, or which plotted the results using ggplot (useful, because I was curious to generate image sequences from plots with varying parameter values to visualise how changes affected the final state plot).

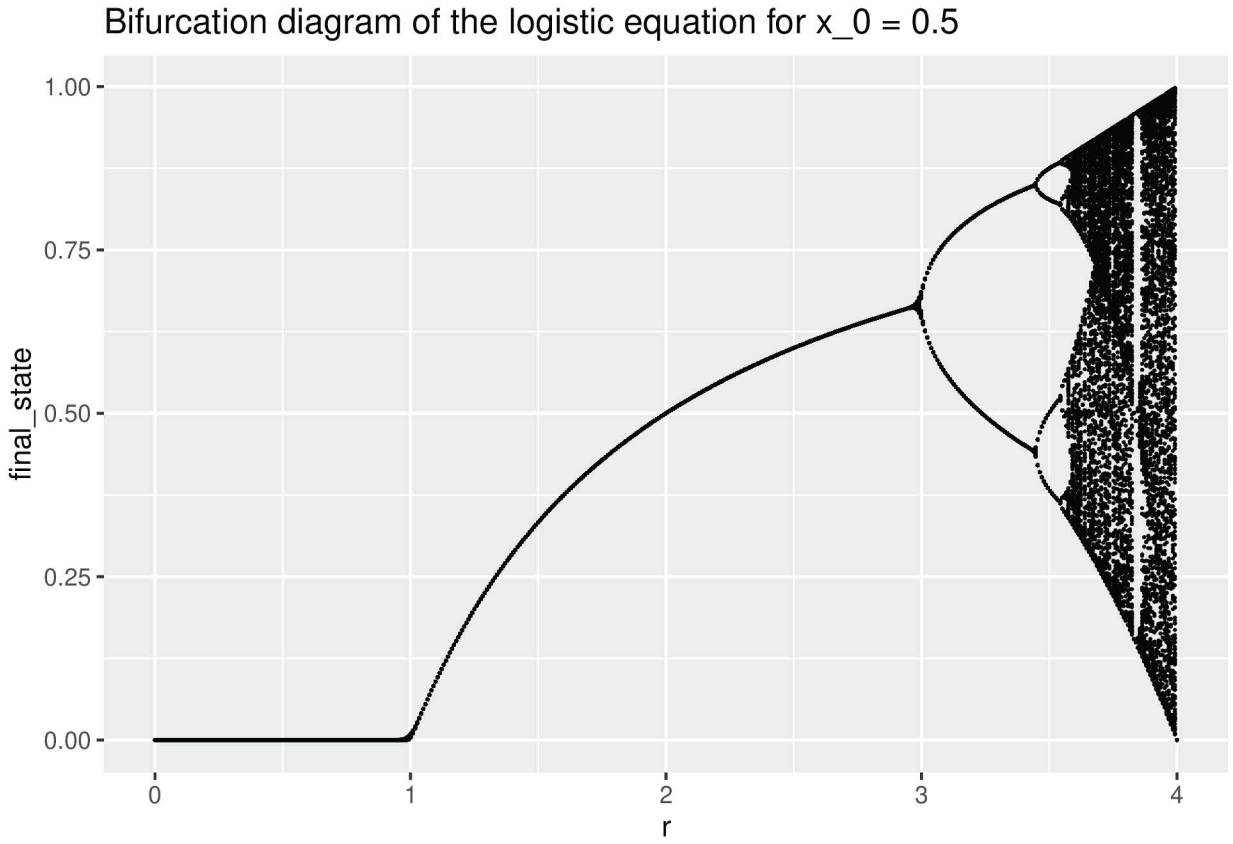
My version of the program is not particularly fast, but it isn't slow either, and it has the advantage of being easy to set up, and in my opinion, very easy to read. It could could be useful in education contexts as a demonstration tool, or as an exploratory tool. Additionally, it could be useful for analytical purposes, as it generates and stored the plotting data in a convenient `tibble` object,

similar to a `dataframe`. In fact, I am using this program in the Data Science II course I am currently taking, to write predictive models of the initial conditions which generate the final state diagrams.

Final state diagrams

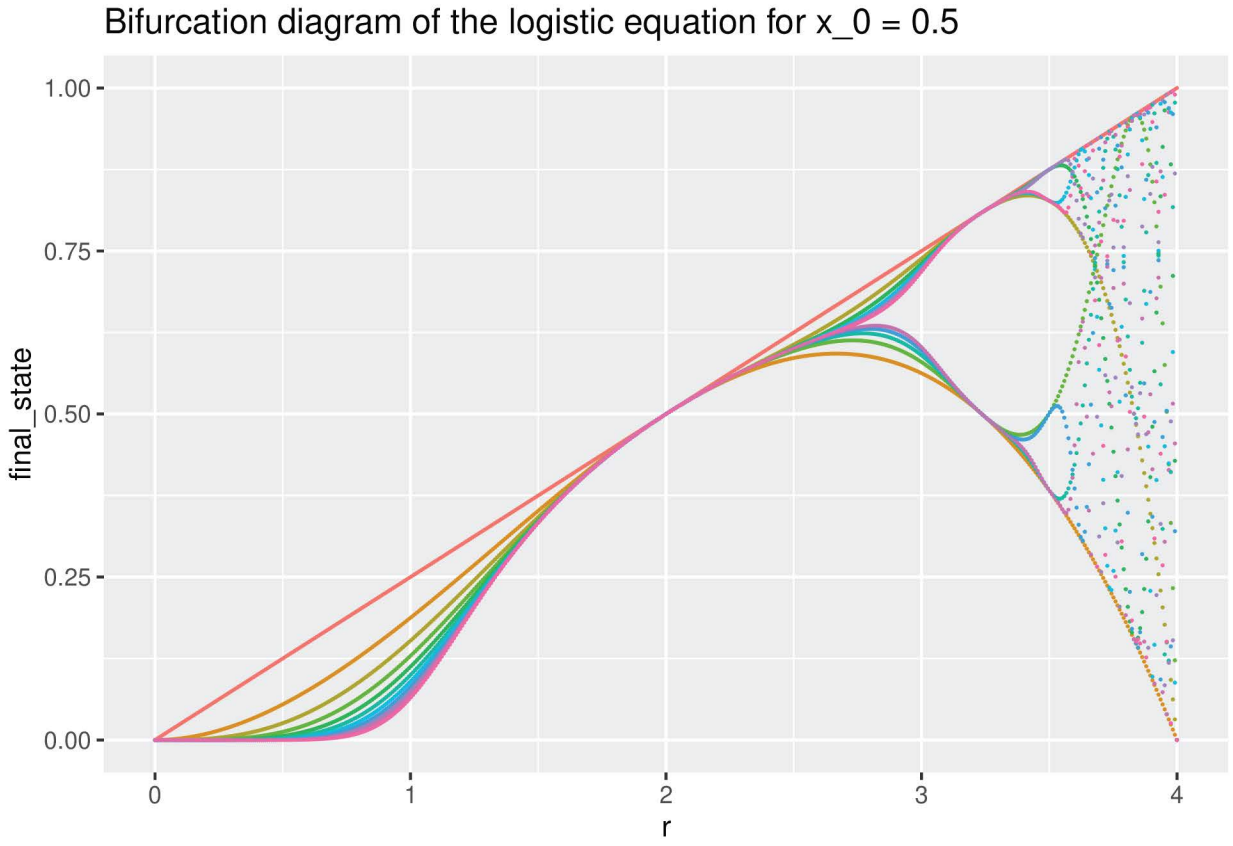
In the following example, I chose to compute 300 iterates, skipping the first 100, with the initial condition x_0 = 0.5:

```
logistic_eq <- function(x, r){return(r*x*(1-x))}

x <- seq(0, 4, length=500) # r scale
y <- c()

bifurcation_data <- function(r_values, x_0, min_iter=100, max_iter=300){
  for(r in r_values){
    new_y <- x_0
    orbit <- c()
    for(i in 0:max_iter){
      new_y <- logistic_eq(new_y, r)
      if(i <  min_iter){next}
      orbit <- c(orbit, new_y)
    }
    y <- c(y, tibble(orbit))
  }
  return(tibble(r=x, final_state=y, iter_num=rep(tibble(min_iter:max_iter),
                                                 length(r_values))))
}

x_0 <- 0.5

bifurcation_data(x, x_0) %>% unnest(everything()) %>% ggplot(aes(r, final_state)) +
  geom_point(size=0.01) +
  labs(title=paste0("Bifurcation diagram of the logistic equation for x_0 = ", x_0))
```



This is quite nice! But what if we did the opposite, and only plotted the first few iterates instead? Let's also colour the points according to their corresponding iterate number (n):

```
logistic_eq <- function(x, r){return(r*x*(1-x))}

x <- seq(0, 4, length=500) # r scale
y <- c()

x_0 <- 0.5
bifurcation_data(x, x_0, 0, 10) %>% unnest(everything()) %>%
  ggplot(aes(r, final_state, colour=as.factor(iter_num))) +
  geom_point(size=0.01) +
  labs(title=paste0("Bifurcation diagram of the logistic equation for x_0 = ",
                    x_0)) +
  theme(legend.position="none")
```



Animation for initial conditions between 0 and 1

If we are curious to quickly compare initial conditions, one way might be to animate an image sequence of plots for a sequence of initial conditions between 0 and 1. The code to generate such an image sequence might go like this:

```
# x_0s <- seq(0, 1, by=0.01)
#
# for(x_0_val in x_0s){
#   bifurcation_data(x, x_0_val, 100, 300) %>% unnest(everything()) %>%
#     ggplot(aes(r, final_state, colour=(iter_num))) +
#     geom_point(size=0.01) +
#     labs(title=paste0("Bifurcation diagram of the logistic equation for x_0 = ",
#                       x_0_val)) +
#     theme(legend.position = "none") +
#     ylim(c(0, 1))
#   ggsave(paste0("bifurcation_diagram_x_0_", x_0_val, ".png"),
#          width=30, height=20, units="cm")
# }
```

We can then flip through the images, or make them into a video like this one: *link to YouTube*

(this is doable entirely in R, and in fact, in really any programming language if you have the time to figure it out, but I just used Adobe After effects in this case, out of convenience)

Complex number plots

Description and contextualisation

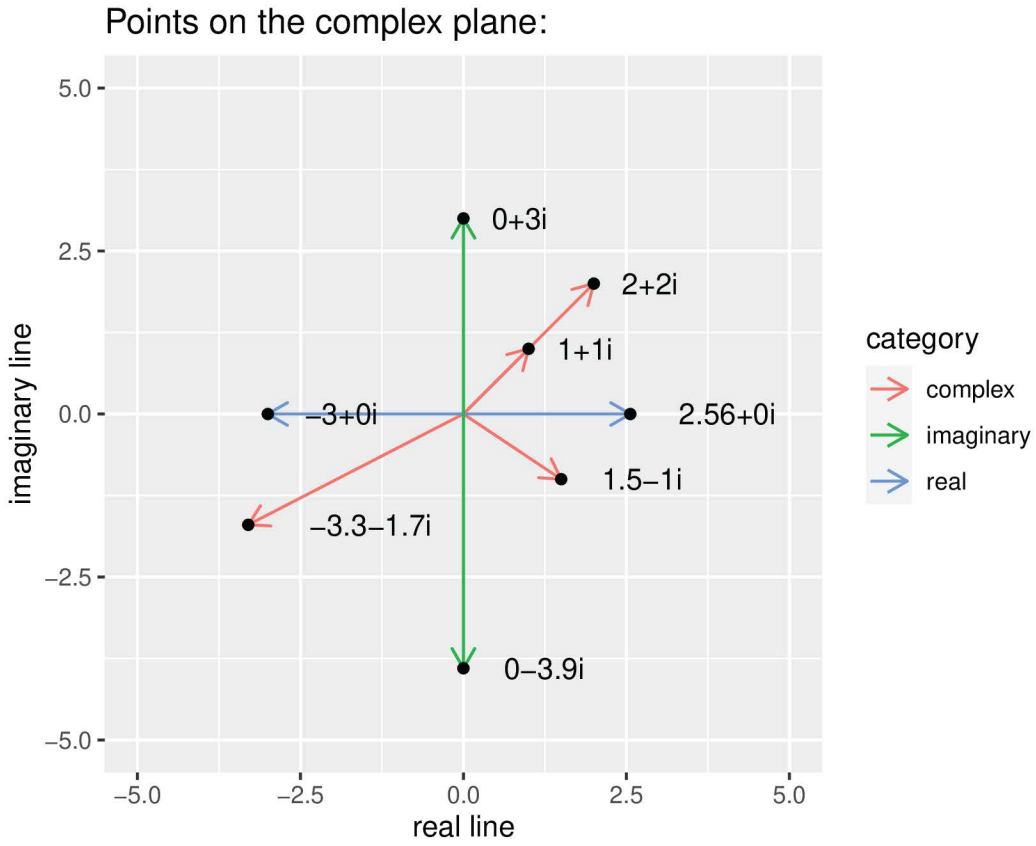
Out of the box, R can compute complex numbers, and plot them on the complex plane, but plotting numbers on the complex plane in ggplot (which results in much nicer plots) required jumping through a few loops and hoops. Plotting in ggplot unlocks a whole bunch of interesting and powerful plotting possibilities in R, including plotting fractals on the complex plane as heat maps, or rasters, which was my primary motivation to learn to do so.

Example

The following code snippet lays out one way of plotting on the complex plane using ggplot.

```
numbers <- c(1+1i, 2+2i, -3.3-1.7i, 1.5-1i, -3+0i, 2.56+0i, 3i, -3.9i)

# let's split the complex numbers into real and imaginary parts to feed to ggplot
real <- Re(numbers)
imaginary <- Im(numbers)
category <- as.factor(c("complex", "complex", "complex", "complex",
                        "real", "real", "imaginary", "imaginary"))

name <- numbers

tibble(real, imaginary, name, category) %>% ggplot(aes(real, imaginary)) +
  geom_segment(lineend = "round", linejoin = "mitre",
               arrow = arrow(length = unit(0.3, "cm")), x=0, y=0,
               xend=real, yend=imaginary, aes(colour=category)) +
  geom_point() +
  geom_text(aes(label=name), hjust=-.5, vjust = .5) +
  labs(title= "Points on the complex plane:", x="real line", y = "imaginary line") +
  lims(x=c(-5, 5), y=c(-5, 5)) +
  coord_equal()
```

```
## Don't know how to automatically pick scale for object of type complex. Defaulting to continu
```



Julia and Mandelbrot set plotters

Description and contextualisation

Julia sets are a type of prisoner set: the set of all values of Z_0, a complex number initial condition for which the orbit of the function $Z_{n+1} = Z_n^2 + C$, where C is a complex number constant. The Mandelbrot set is another prisoner set: the set of all values of C for which the function $Z_{n_1} = Z_n^2 + C$ does not go to infinity when $Z_0 = 0$.

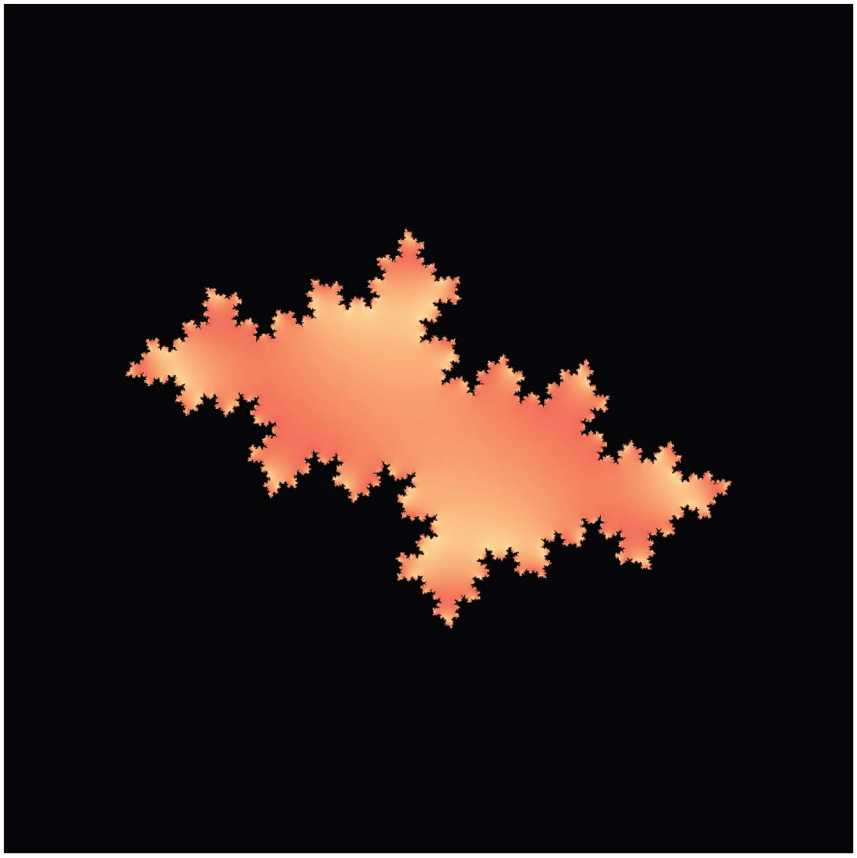
So Julia and Mandelbrot sets are sets of complex numbers, which can be plotted on the complex plane. There are already a lot of examples of Julia and Mandelbrot set plotting tools available online which are much faster, more versatile, and easier to use than my R versions, but although I knew that my code would be slow and a little clunky from the start, I still felt like giving it a go, simply because that would allow me to make such plots offline.

I doubt this code would be very useful for anybody else, but I enjoyed writing it, and I enjoy reading and running it. The code does store the Julia and Mandelbrot plotting data in a convenient `tibble`, so it could be useful for analytical purposes if someone were motivated to put in a little extra work to clean the data.

Julia sets

Julia set plot

```
# let's define a sample value of c
c <-  -0.555+0.455i

# let's define our Julia formula
julia <- function(z, c){
  return(z^2+c)
}

# let's define a maximum number of steps to decide if we think an orbit blows up
# to infinity or not
max_iter <- 100
# now let's define the limits and detail of our plot

# x and y limits (corresponding to real and imaginary)
x <- c(-2, 2); y <- c(-2, 2)
# horizontal and vertical resolution (the number of data points between limits)
h_res <- 1000; v_res <- 1000

real_line <- seq(x[1], x[2], length=h_res)
imaginary_line <- seq(y[1], y[2], length=v_res)*1i

space_grid <- outer(real_line,imaginary_line,"+") %>% c()

z <- space_grid
for(i in 1:max_iter){
  # apply the Julia function to every z_0 value
  z <- julia(z, c)
}

julia_data <- tibble(r=Re(space_grid),
                     i=Im(space_grid),
                     z=as.vector(exp(-Mod(z)))) %>% na.omit()


# plot

julia_data %>% ggplot(aes(r, i, fill=z)) +
  geom_raster(interpolate = T) +
  scale_x_continuous(expand=c(0,0)) +
  scale_y_continuous(expand=c(0,0)) +
  scale_fill_viridis(option="magma", direction = 1) +
  clear_theme +
  coord_equal()
```

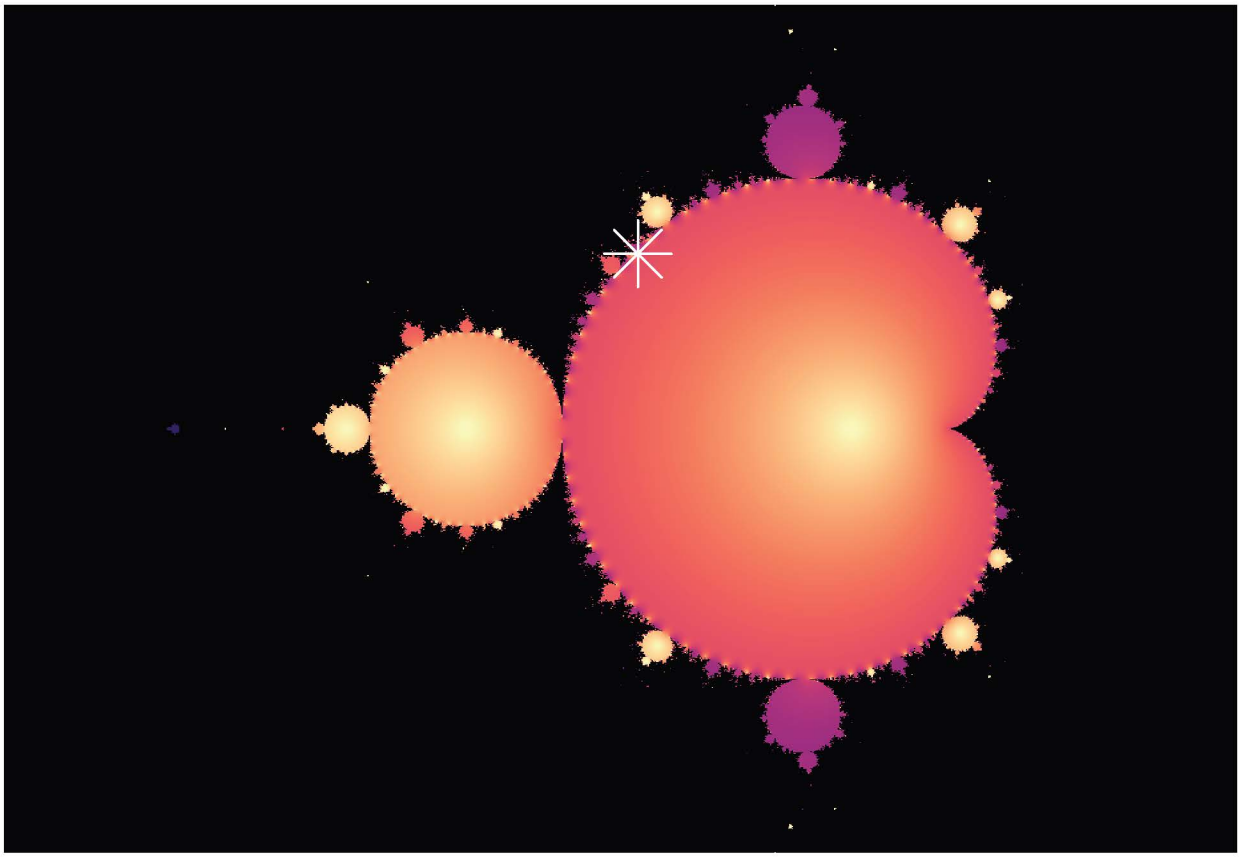


Mandelbrot set

Mandelbrot set plot

```
x <- c(-2.2, 1); y <- c(-1.1, 1.1)
h_res <- 1000; v_res <- 1000

max_iter <- 100

real_line <- seq(x[1], x[2], length=h_res)
imaginary_line <- seq(y[1], y[2], length=v_res)*1i

space_grid <- outer(real_line,imaginary_line,"+") %>% c()

z <- 0
cs <- space_grid # all c values
for(i in 1:max_iter){
  z <- z^2 + cs
}

mandelbrot_data <- tibble(r=Re(space_grid),
                          i=Im(space_grid),
```

```
                        z=as.vector(exp(-Mod(z)))) %>% na.omit()


# plot
mandelbrot_data %>% ggplot(aes(r, i, fill=z)) +
  geom_raster(interpolate = F) +
  scale_x_continuous(expand=c(0,0)) +
  scale_y_continuous(expand=c(0,0)) +
  scale_fill_viridis(option="magma") +
  clear_theme +
  geom_point(aes(Re(c),Im(c)), shape=8, colour="white", size=8) +
  # ^^ add a point for the julia set value of c plotted previously
  coord_equal()
```



Elementary Cellular Automaton

Description and contextualisation

"A cellular automaton is a collection of"colored" cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules based on the states of neighboring cells. The rules are then applied iteratively for as many time steps as desired." (Weisstein, Eric W. "Cellular Automaton." From MathWorld–A Wolfram Web Resource.)

In the Chaos and Fractals course this winter term, our coverage of cellular automata was somewhat covert. As it turned out, we played with cellular automata long before we discussed them explicitly in the class. Nonetheless, the notion of a "cellular automaton" appealed to me intrinsically, perhaps influenced in part by my love for Conway's Game of Life, and the many hours I have spent playing with it, or watching Numberphile video about it.

Therefore, to me, writing a cellular automaton program in R was an obvious goal. I decided to limit my first attempt to programming an elementary cellular automaton, which is to say an automaton that evolved in a single spacial dimension, in a 2d system, where time is the second dimension. I may attempt to recreate Conway's Game of Life at a later date, and creating my own cellular automatons with custom rules and n-dimensional space-times.

This particular program is more fun than it is immediately useful. It could be used for demonstration purposes in an education setting, but I can't really imagine it being useful for too much else out of the box. However, with some work, maybe the layouts produced by the program might serve as a seed generator for some other program later on. Perhaps a 3d (spacial dimension) cellular automaton program, or a 3d model boolean greebling tool. Who knows?

Anyway, let's see it working!

Example

Take this cellular automaton rule:

Figure 1: A Cellular automaton rule

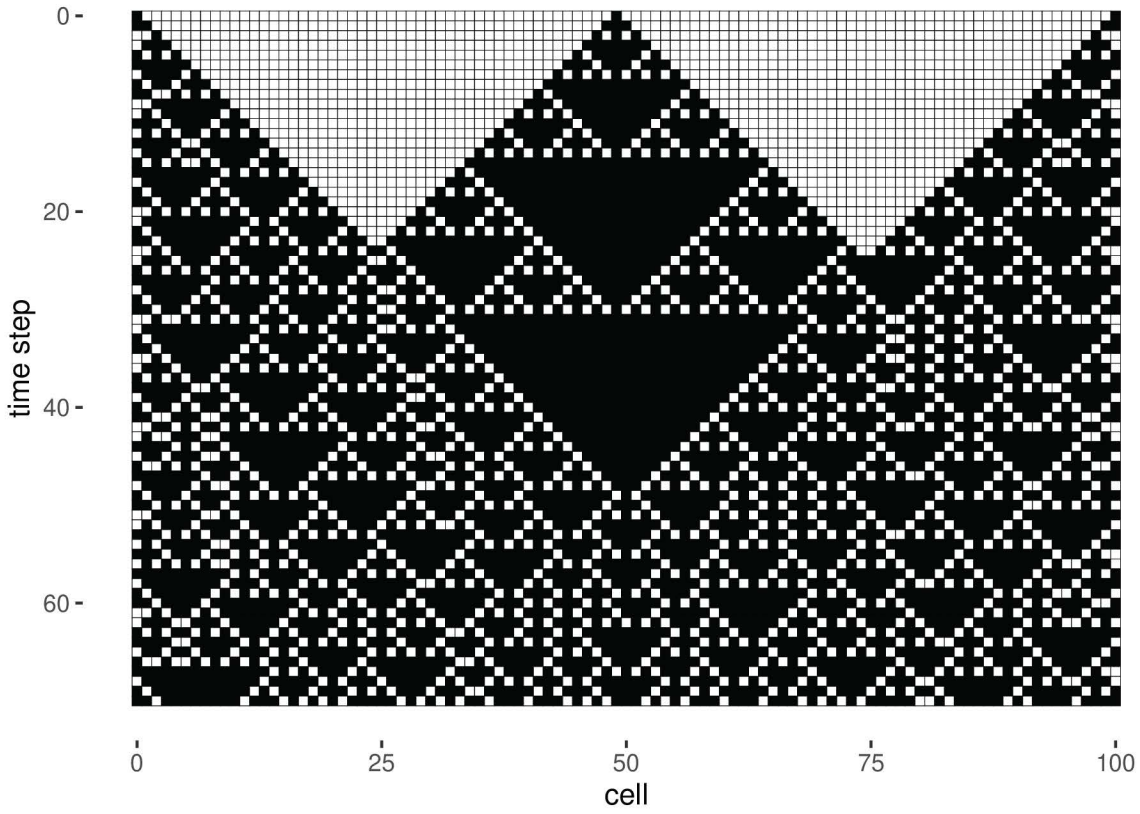
```
clear_theme <- theme(legend.position="none",
                     panel.background = element_rect(fill="white"))
```

```
# initial conditions
# let's define the row of cells at time step n = 0
initial_conditions <- rep(0, 101)
initial_conditions[(length(initial_conditions) %/% 2)] <- 1
initial_conditions[1] <- initial_conditions[length(initial_conditions)] <- 1

# and let's decide how many time steps to apply the rule
n <-  70

last_conditions <- new_conditions <- data <- initial_conditions
x <- rep(0:(length(initial_conditions)-1), n+1)
y <- rep(0:n, each=length(initial_conditions))

for(i in 1:n){
  for(j in 1:length(new_conditions)){
```

```
    # check if the cell has left or right neighbours
    left_neighbour <- F; right_neighbour <- F
    if(j>1){left_neighbour <- T}
    if(j<length(new_conditions)){right_neighbour <- T}

    # update each cell
    values <- c()
    if(left_neighbour){values[1] <- last_conditions[j-1]} else {values[1] <- 0}
    if(right_neighbour){values[3] <- last_conditions[j+1]} else {values[3] <- 0}
    values[2] <- last_conditions[j]
    values <- paste(values, collapse = "")

    # rules
    if(values == "000" | values == "011" | values == "110"){new_conditions[j] <- 0}
    else{new_conditions[j] <- 1}
  }

  data <- c(data, new_conditions)
  last_conditions <- new_conditions
}

image <- tibble(x, y, value=data)
head(image)
```

```
## # A tibble: 6 x 3
##       x     y value
##   <int> <int> <dbl>
## 1     0     0     1
## 2     1     0     0
## 3     2     0     0
## 4     3     0     0
## 5     4     0     0
## 6     5     0     0
```

```
image %>% ggplot(aes(x, y, fill=as.factor(value))) + geom_tile(colour="black") +
  scale_y_reverse() + scale_fill_manual(values=c("white", "black")) +
  coord_equal() + labs(x="cell", y="time step") + clear_theme
```



Here are some initial conditions suggestions:

```
# initial_conditions <- rep(c(0, 1), 52)

# initial_conditions <- sample(c(0, 1), 100, replace = T)

# initial_conditions[length(initial_conditions) %/% 2] <- 1

# Fibonacci initial conditions
# width <- 100 # approximate width
# fibb <- c(1)
# while(sum(fibb)<width){
#   fibb <- c(fibb, sum(tail(fibb, 2)))
# }
# initial_conditions <- c()
# for(i in fibb){
#   initial_conditions <- c(initial_conditions, rep(sample(c(0, 1), 1), length=i))
# }

# inverting copier initial conditions / flipper machine initial conditions
# ^^ (start with F, next is T, next is TF, ...)
# iterations <- 7
# initial_conditions <- c(F)
# for(i in 1:iterations){
#   initial_conditions <- c(initial_conditions, !initial_conditions)
```

```
# }
```

And here are some rule suggestions that all result in different effects:

For a compilation of interesting elementary cellular automaton rules, see $\href *https* : *//en.wikipedia.org/wiki/Elementary_cellular_automaton*

```
# if((values == "101" | values == "001" | values == "100")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if((values == "101" | values == "001" | values == "110")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if((values == "101" | values == "001" | values == "011")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if((values == "111" | values == "000" | values == "011")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if((values == "111" | values == "000")){new_conditions[j] <- 0}else{new_conditions[j] <- 1}

# if((values == "011" | values == "000" | values == "100")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if((values == "010" | values == "001" | values == "100")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if((values == "010" | values == "000" | values == "111")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if((values == "010" | values == "111")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if((values == "101" | values == "001" |
#     values == "100" | values == "010")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if(!(values == "101" | values == "001" |
#     values == "100" | values == "010")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if(!(values == "111" | values == "001" |
#     values == "100" | values == "010")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if(!(values == "111" | values == "000" |
#     values == "100" | values == "010")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}
```

```
# if(!(values == "111" | values == "000" | values == "010")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}

# if(!(values == "011" | values == "110" | values == "000")){new_conditions[j] <- 0}
# else{new_conditions[j] <- 1}
```

Note that the rules and initial conditions interact such that the same initial conditions usually result in different outcomes for different rules, and that the same rule will usually produce different outcomes for different initial conditions.

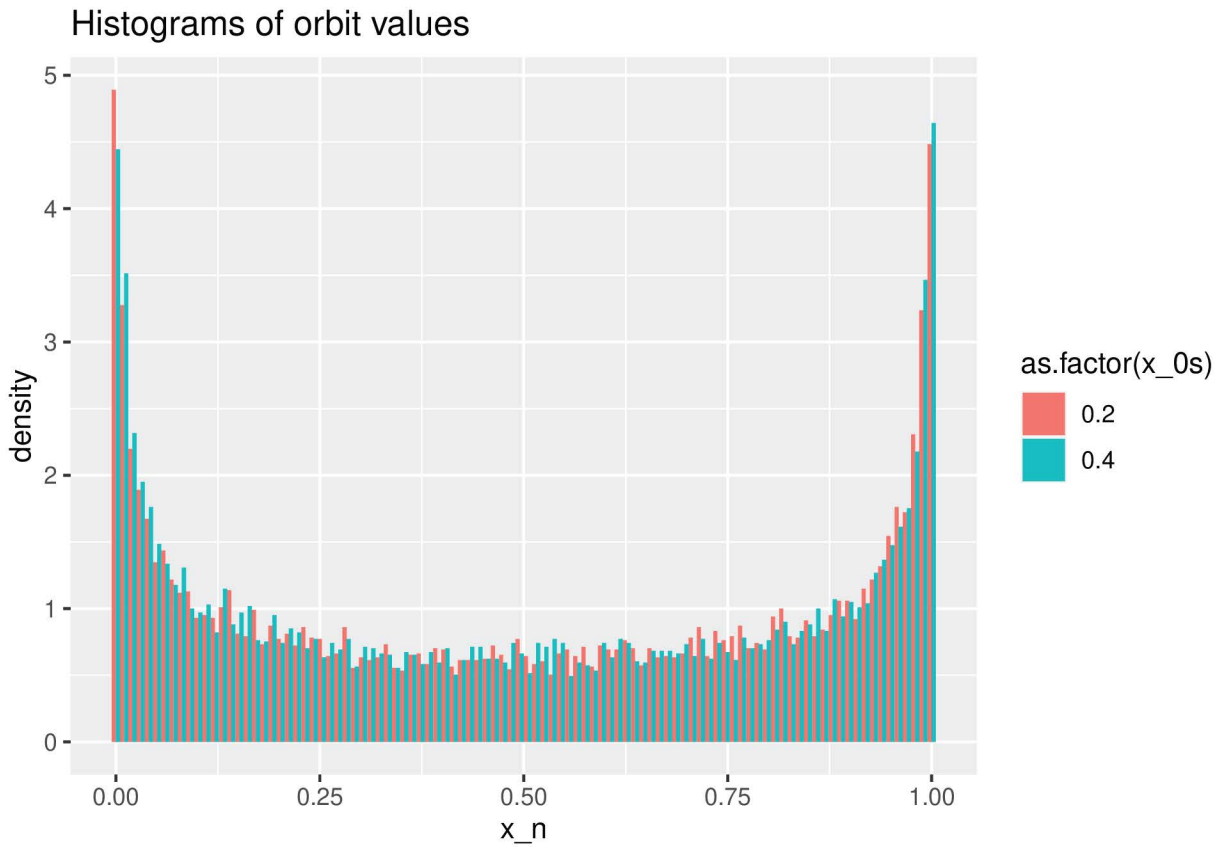
Bonus projects:

The following projects are snippets of code I wrote this term which I felt were too small, too specific, or too inflexible to write about in this final, but which I still felt deserved to be included here some form, partially because I'm pleased with the results they produce (even though the code is a little clunkier in these than in my other examples), and because I am likely to come back to this final in search of my chaos and fractals code in the future, so will sleep better if I know the important ones are are there.

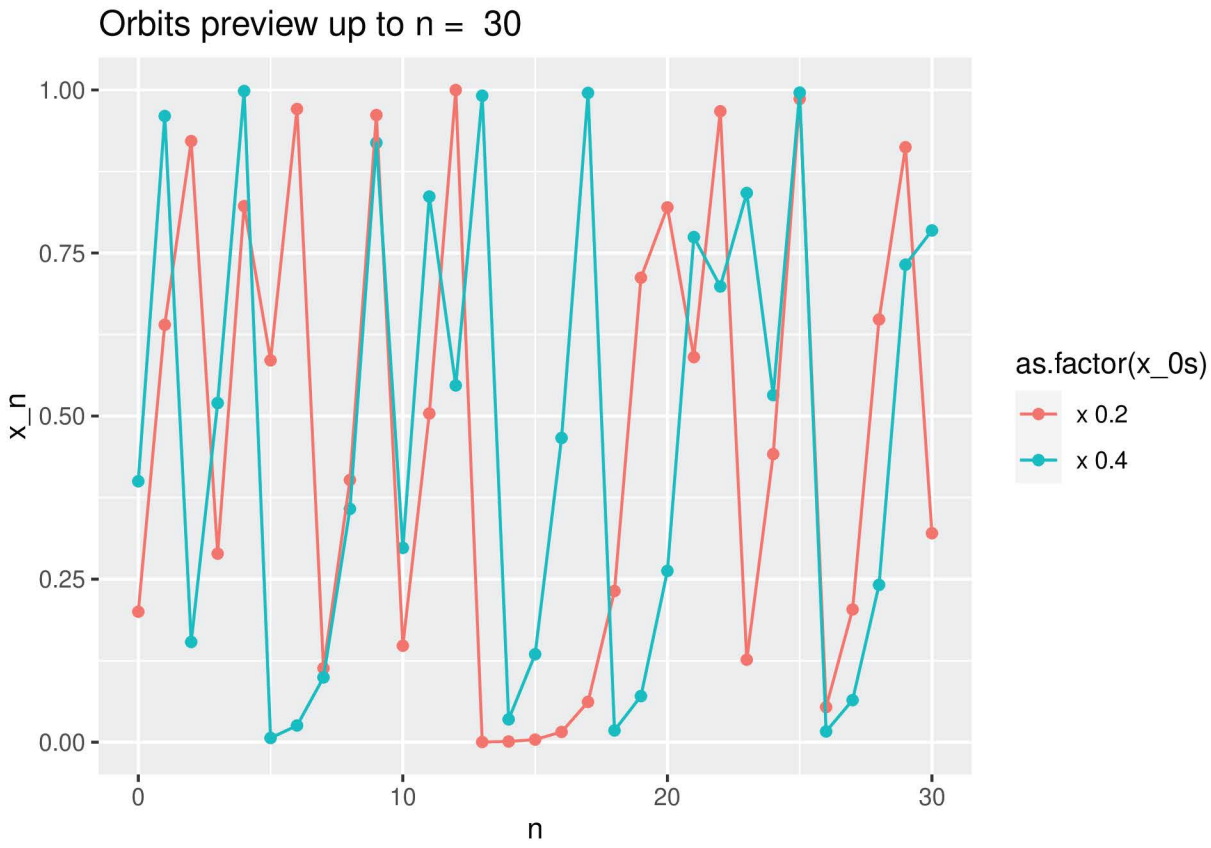
Histogram of orbits of the logistic map

```
r <- 4
x_0s <- c(0.2, 0.4)
N <- 10000

bins <- 100
orbit_preview_length <- 30

# function declaration
func <- function(x){
  return(r*x*(1-x))
}

get_function_iteration_trajectories <- function(x_0s, N = 100){
  trajectories <- data.frame()

  for(i in x_0s){
    x_t <- i
    x_0 <- rep(i,times=N+1)
    n <- 0:N

    trajectory <- c(x_t)

    for(t in 0:(N-1)){
```

```
      x_t <- func(x_t)
      trajectory <- c(trajectory, x_t) # add x_t_1's value to the trajectory vector
    }
    trajectories <- rbind(trajectories, data.frame(x_0s = x_0, n = n, x_n = trajectory))
  }
  return(trajectories)
}

trajectories <- get_function_iteration_trajectories(x_0s = x_0s, N = N)

trajectories %>% ggplot(aes(x_n, y=..density..)) +
  geom_histogram(aes(fill=as.factor(x_0s)), position="dodge", bins=bins) +
  labs(title = paste("Histograms of orbit values"))
```




```
trajectories$x_0s <- paste("x", trajectories$x_0s)
trajectories %>% pivot_wider(names_from = x_0s, values_from = x_n) %>%
  filter(n <= orbit_preview_length) %>%
  pivot_longer(-n, names_to = "x_0s", values_to = "x_n") %>%
  ggplot(aes(n, x_n, colour = as.factor(x_0s))) +
  geom_point() + geom_line() +
  labs(title = paste("Orbits preview up to n = ", orbit_preview_length))
```



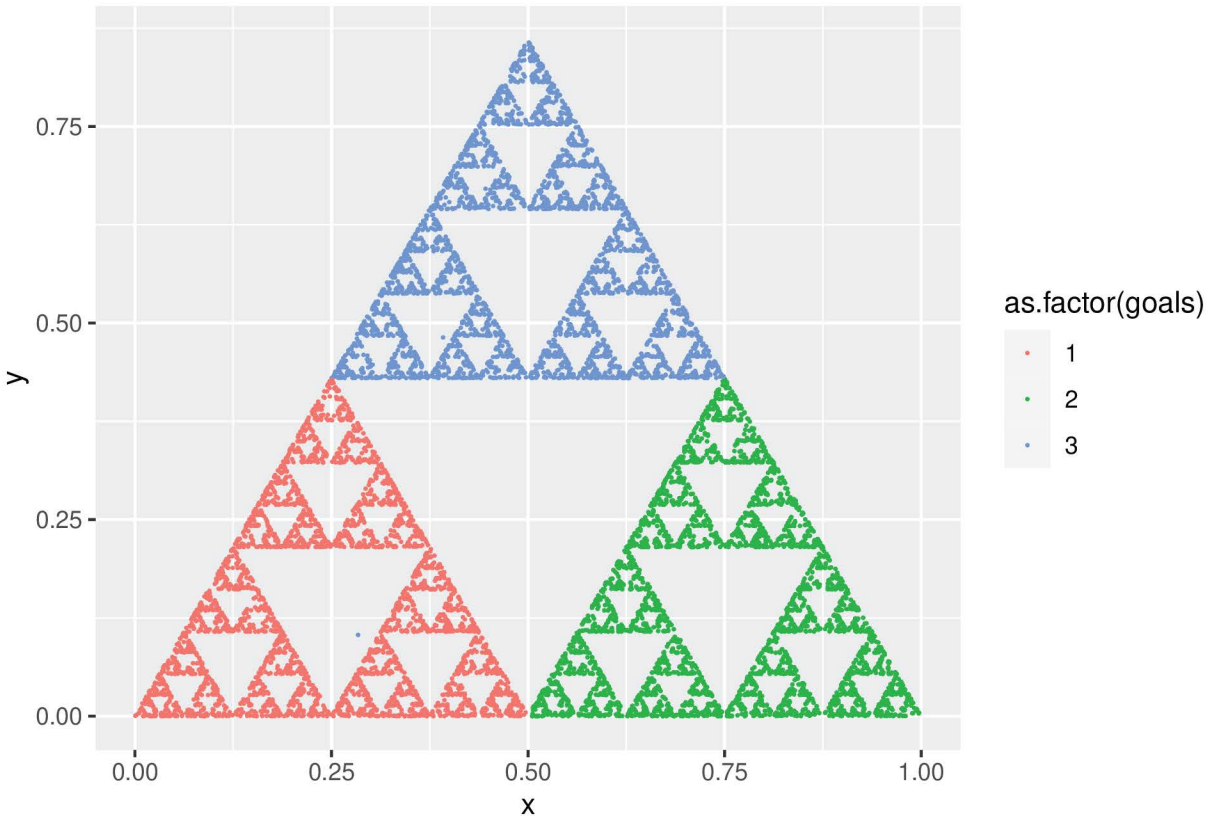
Sierpinski Gasket

```
N = 10000
movement_ratio <- 0.5
point_size <- 0.1

# let's set up a list of triangle point coordinates to use in our chaos game
# function (I'm sure I could write this out better, but I'm feeling lazy today)
triangle_points <- list(c(0, 0), c(1, 0), c(0.5, 0.86))

# choose a random point inside the triangle
y <- runif(1, 0, 0.86)
x <- runif(1, ((0.25/0.43)*y), (1-(0.25/0.43)*y)) # this is not strictly needed,
# we could have the first point start anywhere.

goal_index <- sample(c(1:3), 1)
goal_vertex <- triangle_points[[goal_index]] # first goal vertex

i <- 0

points <- data.frame(x, y, i, goals=goal_index)
```

```
for(i in 1:N){
  # set the last point
  last_point <- tail(points, 1)[1:2]
  # roll a three-sided die
  points[nrow(points) + 1,] = c(((goal_vertex[1]+last_point[1])*(1-movement_ratio)),
                                ((goal_vertex[2]+last_point[2])*(1-movement_ratio)),
                                i, goal_index)
  goal_index <- sample(c(1:3), 1)
  goal_vertex <- triangle_points[[goal_index]]
}

head(points)
```

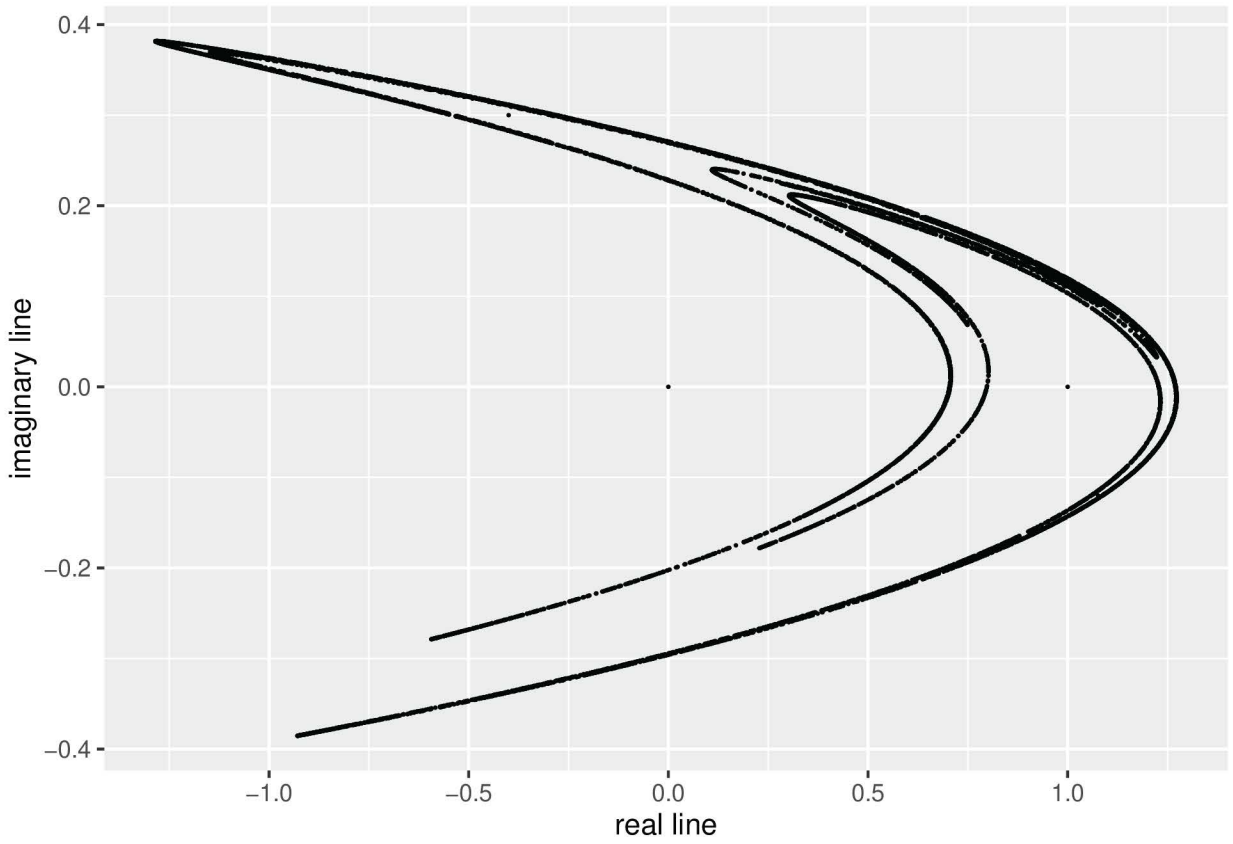
```
##           x         y i goals
## 1 0.2837934 0.1033356 0     3
## 2 0.3918967 0.4816678 1     3
## 3 0.4459484 0.6708339 2     3
## 4 0.4729742 0.7654169 3     3
## 5 0.2364871 0.3827085 4     1
## 6 0.6182435 0.1913542 5     2
```

```
points %>% ggplot(aes(x, y, colour=as.factor(goals))) + geom_point(size=point_size) +
  coord_equal() + lims(x=c(0,1), y=c(0, 0.86))
```

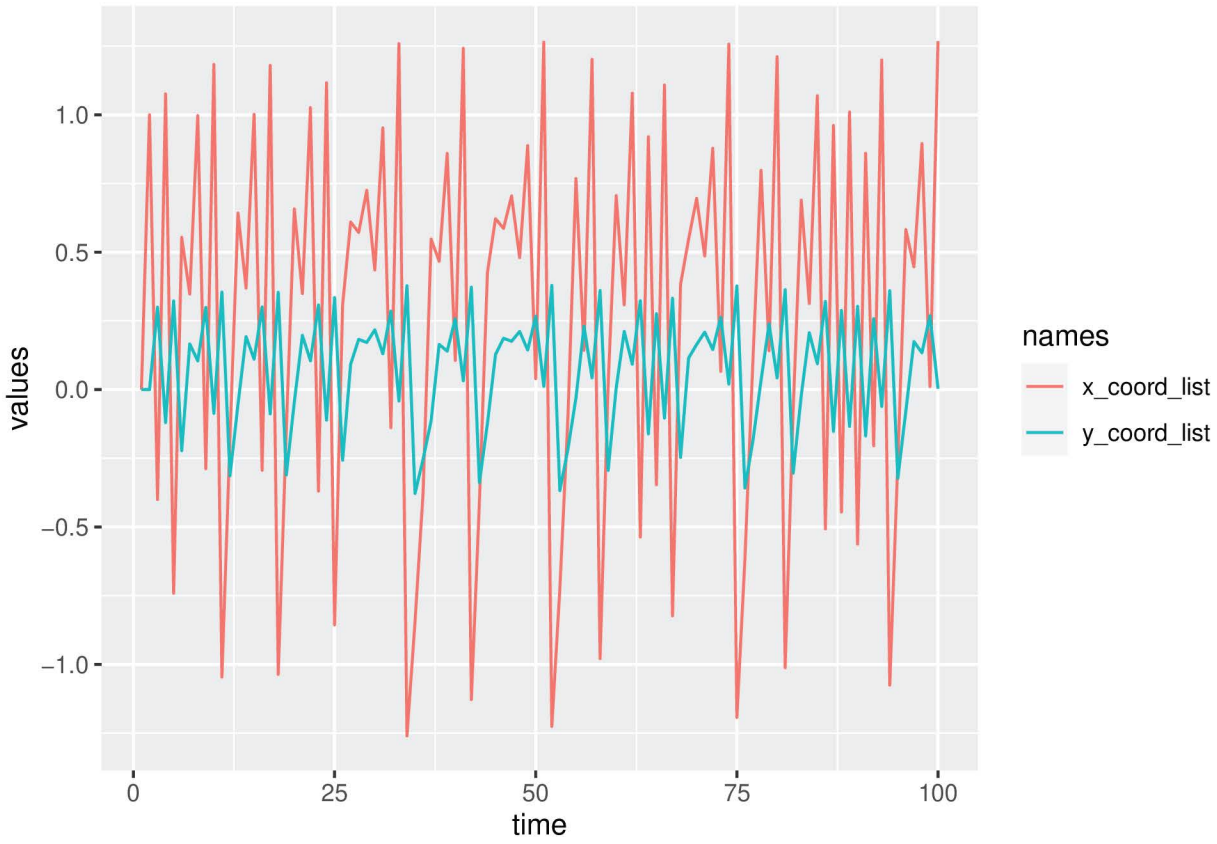


Henon map

```
n <- 10000
x_0 <- 0
y_0 <- 0

a <- 1.4; b <- 0.3

henon_next_point <- function(x, y, a, b){
  new_x <- y+1-a*x^2
  new_y <- b*x
  return(list(new_x, new_y))
}

find_coordinates_for_n_iterations <- function(x_0, y_0, n){
  x_coord_list = c(x_0)
  y_coord_list = c(y_0)
  x <- NA; y <- NA
  for(i in 1:n){
    c(x,y) %<-% henon_next_point(tail(x_coord_list, 1),
                                 tail(y_coord_list, 1), a, b)

    x_coord_list = c(x_coord_list,x)
    y_coord_list = c(y_coord_list, y)
  }
  return(tibble(x_coord_list, y_coord_list))
}

henon_coordinates <- find_coordinates_for_n_iterations(x_0, y_0, n)
henon_coordinates <- henon_coordinates %>% mutate(time=row_number())

# spacial plot
henon_coordinates %>% ggplot(aes(x_coord_list, y_coord_list)) + geom_point(size=0.1) +
  labs(x="real line", y="imaginary line")
```



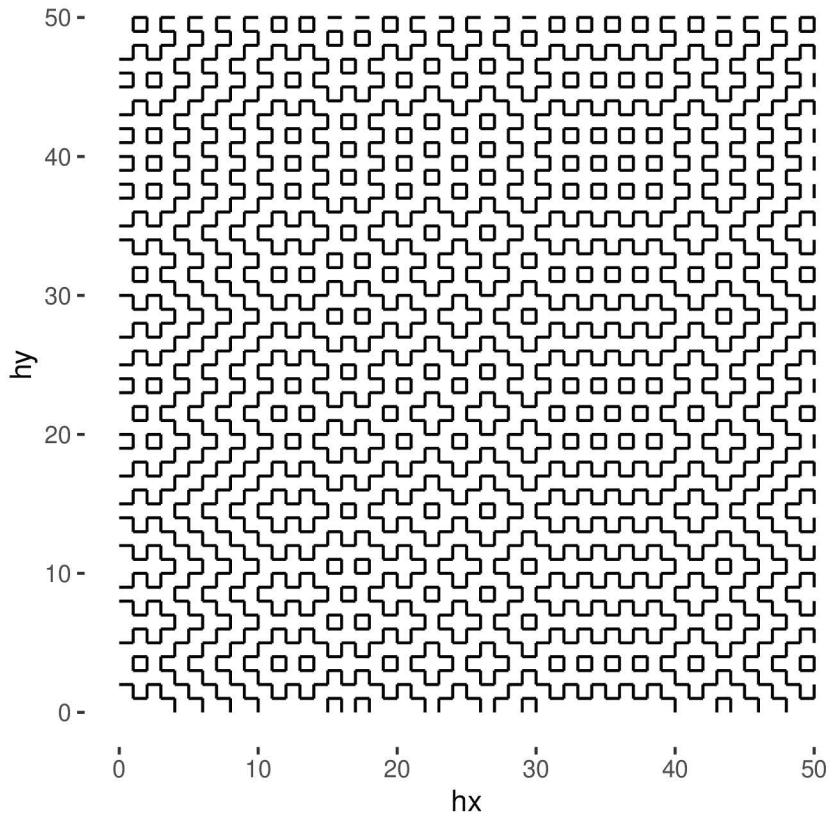

```
# time series plot of the first 100 iterations
head(henon_coordinates, 100) %>%
  pivot_longer(-time, names_to = "names", values_to = "values") %>%
  rowwise() %>% ggplot(aes(time, values, colour=names)) + geom_line()
```



Knitting

```
# canvas settings
len <- 50
number <- 2 # desired number of unique patterns

row_values <- sample(c(0, 1), len, replace=T) # 0 or 1 values
col_values <- sample(c(0, 1), len, replace=T) # 0 or 1 values

# paste(c("row values: ", row_values), collapse=" ")
# paste(c("col values: ", col_values), collapse=" ")

zero <- seq(1, by=2, length=(len %/% 2))
one <- seq(0, by=2, length=(len %/% 2))

# paste(c("zero sequence: ", zero), collapse=" ")
# paste(c("one sequence: ", one), collapse=" ")

dash_line_coords <- function(values){
  ticks <- c()
  for(v in values){
    if(v == 0){ticks <- c(ticks, zero)}
```

```
    else if(v == 1){ticks <- c(ticks, one)}
  }
  return(ticks)
}

x <- c(); y <-c() # ticks
x <- dash_line_coords(row_values) # x ticks
y <- dash_line_coords(col_values) # y ticks

# x y xend yend format for geom_segment:
hx <- x; hy <- rep(1:(len), each=(len %/% 2))
hxend <- x+1; hyend <- hy

vx <- rep(1:(len), each=(len %/% 2)); vy <- y
vxend <- vx; vyend <- y+1

tibble(hx, hy, hxend, hyend,
               vx, vy, vxend, vyend) %>%
  ggplot() +
  geom_segment(aes(x=hx, y=hy, xend=hxend, yend=hyend)) +
  geom_segment(aes(x=vx, y=vy, xend=vxend, yend=vyend)) + clear_theme +
  coord_equal()
```



Reflection

This term was a roller-coaster ride. A lot of work, a lot of ups and downs, high hopes and some pretty big disappointments. Through it all, funnily enough, the Chaos and Fractals course was an anchor and a space of stability which prevented me from drifting too far off course from my goals and responsibilities to myself and to others. In an otherwise at times distractingly relaxed, at times distractingly difficult term, I have been on a journey of mathematical discovery and learning. My interest in mathematics has grown since my arrival at College of the Atlantic, but this term my relationship with maths has changed. I never had expected that I would want to pursue mathematics for the sake of mathematics, or as an academic concentration and potentially significant part of my career to come. I feel that I am developing an intuitive sense of some truly meaningful relationships that transcend maths into physics, chemistry, ecology, and the social sciences, just to name a few examples, and I am excited to dive deeper and learn more.