Learning to code can be quite hard. Apart from the difficulties of learning a new *language*, following a book can be quite boring. From my point of view, one of the bests ways to become a good programmer is choosing *small and funny* experiments oriented to train specific techniques of programming. In this tutorial, we will learn to combine C++ with R to create efficient loops. We will also learn interesting things about ggplot2, a very important package for data visualization. The *excuse* to do all this is to create beautiful images using a mathematical concept called *Strange Attractors*, that we will discuss briefly also. Combining art and coding is often called *generative art*, so be ready because after following this tutorial you can discover a new use of R to fall for as I am. Here we go!

**Introduction**

Iteration is an indispensable technique in every programming language. Mastering it will help you to produce robust, fast, concise and easy to understand code. Iterating is repeating the same task over and over and the first way to do that efficiently is writing a *loop*, which is a way to resume many lines of code in just a few of them. In R, you can do this using the for function. For example, this loop calculates the mean of each column of a data frame called df:

library(tidyverse)

n <- 1000 # Rows of data frame

# df with 4 columns and n rows sampled from N(0,1)

df <- tibble(

a = rnorm(n),

b = rnorm(n),

c = rnorm(n),

d = rnorm(n)

)

output <- vector("double", ncol(df)) # Empty output

# For loop

for (i in 1:ncol(df)) {

output[[i]] <- mean(df[[i]])

}

output

## [1] -0.0069680221 0.0247380568 0.0014428622 -0.0002675645

In the previous example, we repeated 4 times the task (which is the *piece* of code inside the curly brackets). Another way to do iterations in R, sometimes more efficient than a for loop, is by using the *apply* family. This is how the previous loop can be rewritten using the apply function:

output <- apply(df, 2, mean)

output

These approaches have limitations. For example, if you want to use the result of an iteration as the input of the next one, you can not use *apply*. Another one is that sometimes R is not enough fast, which makes the for function quite inefficient. When you face this kind of situations, a good option is combinnig C++ and R. To illustrate this, let’s briefly introduce a stunning mathematical *creature*: the strange attractors.

**What are strange attractors?**

In mathematics, a dynamical system is a set of functions which describes the time dependence of a point in a geometrical space. An attractor is a set of numerical values towards a dynamical system tend to evolve in discrete time regardless of their starting conditions. Simple attractors can be **fixed points**, **sets of points** or **limit cycles**. More interesting attractors are *strange*, *chaotic* or *itinerant* attractors, which span an array of possible states in which a dynamical system can roam around without repeating itself.

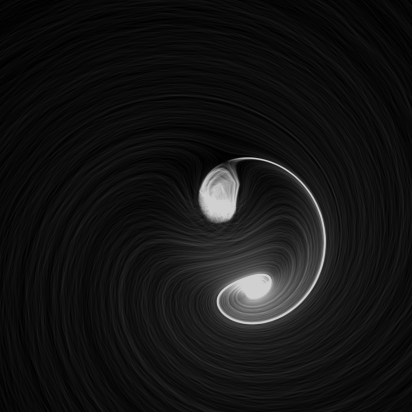
One of the best-known examples of strange attractors was discovered by Kiyohiro Ikeda [1](https://codingclubuc3m.rbind.io/post/2019-10-15/#fn1) and bears his name. It is defined by the next equations:

\[  
x\_{n+1}=1+u(x\_{n}\cos t\_{n}-y\_{n}\sin t\_{n}) \\  
y\_{n+1}=u(x\_{n}\sin t\_{n}+y\_{n}\cos t\_{n})  
\]

with \(u\ge0.6\) and

\[  
t\_{n}=0.4-{\frac {6}{1+x\_{n}^{2}+y\_{n}^{2}}}  
\]

In this example, our *imaginary* point moves in a 2D dimensional space, since we just need (x, y) coordinates to describe its movement. In the previous system, time is represented by n. As you can see, the Ikeda’s system only depends on one parameter, called u. The location of the point at n depends only on its location at n-1. For example, this image [2](https://codingclubuc3m.rbind.io/post/2019-10-15/#fn2) shows the trajectories of 2000 random points following this previous equation with \(u = 0.918\):



All points, regardless of their starting place, tend to end following the same path, represented in bright white color. Another interesting property of strange attractors is that points that are very close at some instant, can differ significantly in the next step. This is why strange attractors are also known as *chaotic* attractors.

Today we will experiment with Clifford attractors, which are defined by these equations:

\[  
x\_{n+1} = sin(a \: y\_{n}) + c \: cos(a \: x\_{n}) \\  
y\_{n+1} = sin(b \: x\_{n}) + d \: cos(b \: y\_{n})  
\]

These attractors were discovered by [Clifford Pickover](https://en.wikipedia.org/wiki/Clifford_A._Pickover), an american scientific divulgator and depend on four parameters; a, b, c and d.

**Populating attractors: an inefficient approach**

To represent the trajectory of a particular point, we need to store in a data frame its locations at every single (discrete) moment of time. For example, this is how we can do it using a for loop:

# Parameters setting

a <- -1.25

b <- -1.25

c <- -1.82

d <- -1.91

# Number of jumps

n <- 5

# Initialization of df to store locations of point

df <- data.frame(x = double(n+1), y = double(n+1))

# The point starts at (0,0)

df[1, "x"] <- 0

df[1, "y"] <- 0

# For loop to store locations

for (i in 1:n) {

df[i+1, "x"] <- sin(a\*df[i, "y"])+c\*cos(a\*df[i, "x"])

df[i+1, "y"] <- sin(b\*df[i, "x"])+d\*cos(b\*df[i, "y"])

}

df

## x y

## 1 0.0000000 0.0000000

## 2 -1.8200000 -1.9100000

## 3 1.8629450 2.1543131

## 4 0.8172482 0.9946399

## 5 -1.8969558 -1.4673201

## 6 2.2716156 1.1936322

As you can see, our point started at (0,0) and *jumped* 5 times (the number of jumps is represented by n). You can try to change n from 5 to 10000000 and see what happens (disclaimer: be patient).

A better approach to do it is using purrr, a functional programming package which provides useful tools for iterating through lists and vectors, generalizing code and removing programming redundancies. The purrr tools work in combination with functions, lists and vectors and results in code that is consistent and concise. This can be a purrr approach to our problem:

library(purrrlyr)

n <- 5

# Initialization of our data frame

df <- tibble(x = numeric(n+1),

y = numeric(n+1))

# Convert our data frame into a list

by\_row(df, function(v) list(v)[[1L]], .collate = "list")$.out -> df

# This function computes current location depending of previous one

f <- function(j, k, a, b, c, d) {

tibble(

x = sin(a\*j$y)+c\*cos(a\*j$x),

y = sin(b\*j$x)+d\*cos(b\*j$y)

)

}

# We apply accumulate on our list to store all steps and convert into a data frame

accumulate(df, f, a, b, c, d) %>% map\_df(~ .x) -> df

df

## # A tibble: 6 x 2

## x y

##

## 1 0 0

## 2 -1.82 -1.91

## 3 1.86 2.15

## 4 0.817 0.995

## 5 -1.90 -1.47

## 6 2.27 1.19

Even this approach is not efficient since the accumulate function must be applied on an existing object (the df in our case)

**Combining C++ and R: the good way**

Rcpp makes it very simple to connect C++ to R and provides a clean, approachable API that lets you write high-performance code, insulated from R’s complex C API. Typical bottlenecks that C++ can address include:

* Loops that can’t be easily vectorised because subsequent iterations depend on previous ones.
* Recursive functions, or problems which involve calling functions millions of times. The overhead of calling a function in C++ is much lower than in R.
* Problems that require advanced data structures and algorithms that R doesn’t provide. Through the standard template library (STL), C++ has efficient implementations of many important data structures, from ordered maps to double-ended queues.

This code populates our data frame using C++:

# Load in libaries

library(Rcpp)

library(tidyverse)

# Here comes the C++ code

cppFunction('DataFrame createTrajectory(int n, double x0, double y0,

double a, double b, double c, double d) {

// create the columns

NumericVector x(n);

NumericVector y(n);

x[0]=x0;

y[0]=y0;

for(int i = 1; i < n; ++i) {

x[i] = sin(a\*y[i-1])+c\*cos(a\*x[i-1]);

y[i] = sin(b\*x[i-1])+d\*cos(b\*y[i-1]);

}

// return a new data frame

return DataFrame::create(\_["x"]= x, \_["y"]= y);

}

')

a <- -1.25

b <- -1.25

c <- -1.82

d <- -1.91

# Call the function starting at (0,0) to store 10M jumps

df <- createTrajectory(10000000, 0, 0, a, b, c, d)

head(df)

## x y

## 1 0.0000000 0.0000000

## 2 -1.8200000 -1.9100000

## 3 1.8629450 2.1543131

## 4 0.8172482 0.9946399

## 5 -1.8969558 -1.4673201

## 6 2.2716156 1.1936322

Note that, apart from Rcpp, we also loaded the tidyverse package. This is because we will use some of its features, as dplyr or ggplot2, later.

The function createTrajectory is written in C++ and stores the itinerary of an initial point. You can try it to see that It’s extremely fast. In the previous example, I populated on the fly our data frame with 10 million points starting at (0,0) in less than 5 seconds in a conventional laptop.

**Rendering the plot with ggplot**

Once we have built our data frame, it’s time to represent it with ggplot.

Making graphics is a necessary skill and for this ggplot is an amazing tool, which by  
the way, I encourage you to learn. Usually, a good visualization is the best way to show your results. R is an extraordinary tool for visualizing data and one of the most important libraries to do it is ggplot2: it is powerful, flexible and is updated to include functionalities regularly. Although its syntax may seem a bit strange at first, here are four important concepts from my point of view to help you out with understanding it:

* It’s truly important to have clear the type of graph you want to plot: for each one there’s a different *geometry*: geom\_point to represent points, geom\_line to represent time series, geom\_bar to do bar chars … Have a look to ggplot2 package, there are many other types.
* Each of those geometries needs to define what data is to be used and how. For example, as you will see in our graph, we placed on the x-axis the x column of our data frame df and y on the y-axis, so we did aes(x = x, y = y).
* You can also combine geometries. We didn’t do that for our example but you may want, for example, to combine bars and lines on the same plot. To do it you can just use + operator. This operator is very important: it doesn’t only combine geometries, it’s also used to add features to the plot as we did to define its aspect ratio (with coord\_equal) or its appearance (with theme\_void).
* You can modify every detail of the look and feel of your graph but be patient. To begin with, I recommend the use of predefined themes as I did with theme\_void. There are many of them and it’s is a *quick and easy* way to change the appearance of your plot: just a way of making life easier and simpler.

The ggplot2 package is a new world by itself and takes part of the tidyverse, a collection of efficient libraries designed to analyze data in a consistent way.

This is how we can represent the points of our data frame:

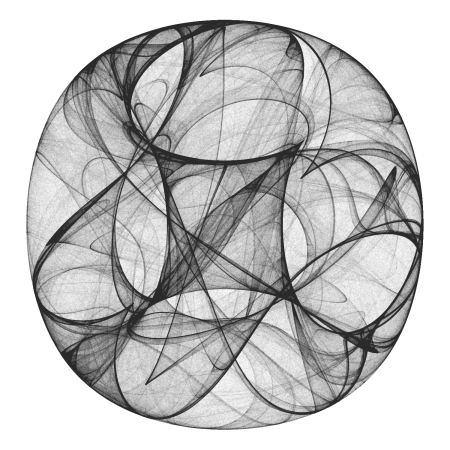
ggplot(df, aes(x = x, y = y)) +

geom\_point(shape = 46, alpha = .01) +

coord\_equal() +

theme\_void() -> plot

ggsave("strange1.png", plot, height = 5, width = 5, units = 'in')



The syntax of our plot is quite simple. Just two comments about it:

* Shape is set to 46, the smallest filled point.
* Since there are a lot of points, I make them transparent with alpha to take advantage on its overlapping and reveal patterns of shadows.

**Playing with parameters**

Strange attractors are extremely sensitive to their parameters. This feature makes them very entertaining as well as an infinite source of surprises. This is what happens to Clifford attractor when setting a, b, c and d parameters to -1.2, -1.9, 1.8 and -1.6 respectively:

a <- -1.2

b <- -1.9

c <- 1.8

d <- -1.6

df <- createTrajectory(10000000, 0, 0, a, b, c, d)

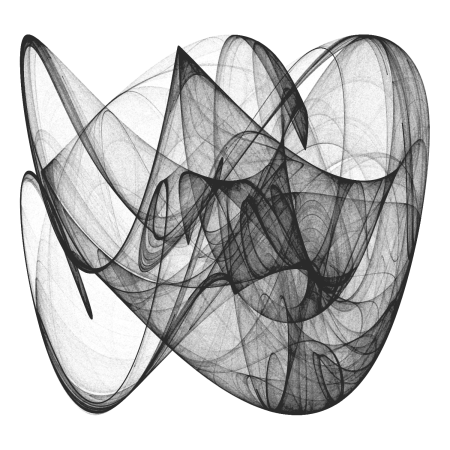
ggplot(df, aes(x = x, y = y)) +

geom\_point(shape = 46, alpha = .01) +

coord\_equal() +

theme\_void() -> plot

ggsave("strange2.png", plot, height = 5, width = 5, units = 'in')



Don’t you feel like playing with parameters? From my point of view, this is a *very classy* way of spending your time.

**The General 2D-Map**

Once we know what strange attractors *look like*, let’s try another family of them. This is the new formula we will experiment with:

\[  
x\_{n+1} = a\_{1} + a\_{2}x\_{n} + a\_{3}y\_{n} + a\_{4}|x\_{n}|^{a\_{5}} + a\_{6}|y\_{n}|^{a\_{7}} \\  
y\_{n+1} = a\_{8} + a\_{9}x\_{n} + a\_{10}y\_{n} + a\_{11}|x\_{n}|^{a\_{12}} + a\_{13}|y\_{n}|^{a\_{14}}  
\]

These attractors are localized mostly to a small region of the XY plane with *tentacles* that stretch out to large distances. If any of the exponents are negative and the attractor intersects the line along which the respective variable is zero, a point on the line maps to infinity. However, large values are visited infrequently by the orbit, so many iterations are required to determine that the orbit is unbounded. For this reason most of the attractors in the figures have holes in their interiors where their orbits are precluded from coming too close to their origins.

cppFunction('DataFrame createTrajectory(int n, double x0, double y0,

double a1, double a2, double a3, double a4, double a5,

double a6, double a7, double a8, double a9, double a10,

double a11, double a12, double a13, double a14) {

// create the columns

NumericVector x(n);

NumericVector y(n);

x[0]=x0;

y[0]=y0;

for(int i = 1; i < n; ++i) {

x[i] = a1+a2\*x[i-1]+ a3\*y[i-1]+ a4\*pow(fabs(x[i-1]), a5)+ a6\*pow(fabs(y[i-1]), a7);

y[i] = a8+a9\*x[i-1]+ a10\*y[i-1]+ a11\*pow(fabs(x[i-1]), a12)+ a13\*pow(fabs(y[i-1]), a14);

}

// return a new data frame

return DataFrame::create(\_["x"]= x, \_["y"]= y);

}

')

a1 <- -0.8

a2 <- 0.4

a3 <- -1.1

a4 <- 0.5

a5 <- -0.6

a6 <- -0.1

a7 <- -0.5

a8 <- 0.8

a9 <- 1.0

a10 <- -0.3

a11 <- -0.6

a12 <- -0.3

a13 <- -1.2

a14 <- -0.3

df <- createTrajectory(10000000, 1, 1, a1, a2, a3, a4, a5, a6,

a7, a8, a9, a10, a11, a12, a13, a14)

ggplot(df) +

geom\_point(aes(x = x, y = y), shape = 46, alpha = .01, color = "black") +

coord\_fixed() +

theme\_void() -> plot

ggsave("strange3.png", plot, height = 5, width = 5, units = 'in')



Ops! What happened? Why our image is so small now? The reason is because some few points are located quite far from the vast majority (you can think about them as *outliers*). Let’s remove 5% of extreme points from both tails independently:

mx <- quantile(df$x, probs = 0.05)

Mx <- quantile(df$x, probs = 0.95)

my <- quantile(df$y, probs = 0.05)

My <- quantile(df$y, probs = 0.95)

df %>% filter(x > mx, x < Mx, y > my, y < My) -> df

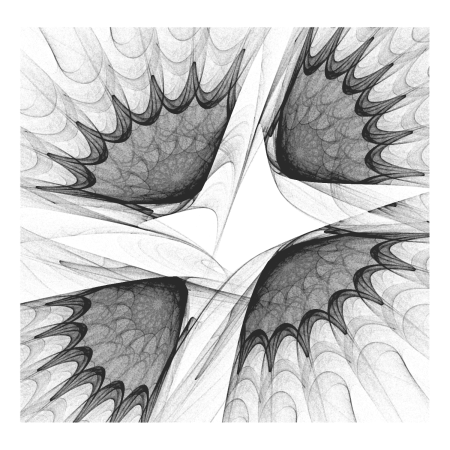
ggplot(df) +

geom\_point(aes(x = x, y = y), shape = 46, alpha = .01, color = "black") +

coord\_fixed() +

theme\_void() -> plot

ggsave("strange3.png", plot, height = 5, width = 5, units = 'in')



Now a stunning image appears.

Sometimes you will need to process the data frame after you generate it since:

* Extreme values can hide the pattern as happened before or even *scape* to infinite (solution: remove outliers).
* Some images can be quite *boring*: from my point of view, the more *distributed* ae points on the XY plane, the nicer is the resulting image (solution: try with a small data frame and keep it if some measure of dispersion over the plane or the individual axis is high enough; in this case, compute the big picture).

To conclude, let’s change the appearance of our plot. We will use the theme function to customize it setting the background color to black and adding a white line as frame.

opt <- theme(legend.position = "none",

panel.background = element\_rect(fill="black", color="white"),

plot.background = element\_rect(fill="black"),

axis.ticks = element\_blank(),

panel.grid = element\_blank(),

axis.title = element\_blank(),

axis.text = element\_blank())

plot <- ggplot(df) +

geom\_point(aes(x = x, y = y), shape = 46, alpha = .01, color = "white") +

coord\_fixed() +

opt

ggsave("strange4.png", plot, height = 5, width = 5, units = 'in')

