The Emergence of Patterns in Nature and Chaos Theory

Ryan Greenup & James Guerra

August 28, 2020

Contents

§ 1 Submission Outline

¶ 1.1 Introduction

This project at the outset was very broadly concerned with using *Python* for computer algebra, much to the the reluctance of our supervisor we have however resolved to look at a broad variety of tools, including *Maxima* and *GNUPlot*, in particular a language we wanted an opportunity to explore was *Julia* [4] [1].

In order to give the project a more focused direction we have decided to look into: [2]

- The Emergence of patterns in Nature
- Chaos Theory & Dynamical Systems
- Fractals

These three topics are very tightly connected and so it is difficult to look at any one in a vacuum, they also almost necessitate the use of software packages due to the fact that these phenomena appear to occur in recursive systems, more over such software needs to perform very well under recursion and iteration (making this a very good focus for this topic generally, and an excuse to work with Julia as well).

¶ 1.2 Preliminary Problems

We did a tonne of these but this one really illustrates the sort of things that we're looking at.

1.2.1 Iteration and Recursion

To illustrate an example of different ways of thinking about a problem, consider the series shown in (1)[3] :

$$g(k) = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{3}}}{3} \frac{\sqrt{2+\sqrt{3+\sqrt{4}}}}{4} \cdot \ldots \frac{\sqrt{2+\sqrt{3+\ldots+\sqrt{k}}}}{k} \tag{1}$$

let's modify this for the sake of discussion:

$$h(k) = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{3+\sqrt{2}}}{3} \cdot \frac{\sqrt{4+\sqrt{3+\sqrt{2}}}}{4} \cdot \ldots \cdot \frac{\sqrt{k+\sqrt{k-1+\ldots\sqrt{3+\sqrt{2}}}}}{k} \tag{2}$$

[1] See section

[2] The amount of independence that our supervisor afforded us to investigate other languages is something that we are both extremely grateful for.

[3] This problem is taken from Project A (44) of Dr. Hazrat's *Mathematica: A Problem Centred Approach* [13]

The function h can be expressed by the series:

$$h\left(k\right)=\prod_{i=2}^{k}\left(\frac{f_i}{i}\right)\quad:\quad f_i=\sqrt{i+f_{i-1}},\ \ f_1=1$$

Within *Python*, it isn't difficult to express h, the series can be expressed with recursion as shown in listing 1, this is a very natural way to define series and sequences and is consistent with familiar mathematical thought and notation. Individuals more familiar with programming than analysis may find it more comfortable to use an iterator as shown in listing 2.

```
##############################################################################
###
from sympy import *
def h(k):
    if k > 2:
        return f(k) * f(k-1)
    else:
        return 1

def f(i):
    expr = 0
    if i > 2:
        return sqrt(i + f(i -1))
    else:
        return 1
```

Listing 1: Solving (2) using recursion.

```
  from sympy import *
  def h(k):
      k = k + 1 # OBOB
      l = [f(i) for i in range(1,k)]
      return prod(l)

  def f(k):
      expr = 0
      for i in range(2, k+2):
          expr = sqrt(i + expr, evaluate=False)
      return expr/(k+1)
```

Listing 2: Solving (2) by using a `for` loop.

Any function that can be defined by using iteration, can always be defined via recursion and vice versa, [6, 5] see also [29, 14]

there is, however, evidence to suggest that recursive functions are easier for people to understand [2] . Although independent research has shown that the specific language chosen can have a bigger effect on how well recursive as opposed to iterative code is understood [28].

The relevant question is which method is often more appropriate, generally the process for determining which is more appropriate is to the effect of:

1. Write the problem in a way that is easier to write or is more appropriate for demonstration
2. If performance is a concern then consider restructuring in favour of iteration
    - For interpreted languages such ***R*** and *Python*, loops are usually faster, because of the overheads involved in creating functions [29] although there may be exceptions to this and I'm not sure if this would be true for compiled languages such as *Julia*, *Java*, ***C*** etc.

Some Functions are more difficult to express with Recursion in

Attacking a problem recursively isn't always the best approach, consider the function $g(k)$ from (1):

$$\begin{aligned} g(k) &= \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{3}}}{3} \frac{\sqrt{2+\sqrt{3+\sqrt{4}}}}{4} \cdot \ldots \frac{\sqrt{2+\sqrt{3+\ldots+\sqrt{k}}}}{k} \\ &= \prod_{i=2}^{k} \left(\frac{f_i}{i} \right) \quad : \quad f_i = \sqrt{i + f_{i+1}} \end{aligned}$$

Observe that the difference between (1) and (2) is that the sequence essentially *looks* forward, not back. To solve using a `for` loop, this distinction is a non-concern because the list can be reversed using a built-in such as `rev`, `reversed` or `reverse` in *Python*, ***R*** and *Julia* respectively, which means the same expression can be implemented.

To implement recursion however, the series needs to be restructured and this can become a little clumsy, see (3):

$$g(k) = \prod_{i=2}^{k} \left(\frac{f_i}{i} \right) \quad : \quad f_i = \sqrt{(k-i) + f_{k-i-1}} \tag{3}$$

Now the function could be performed recursively in *Python* in a similar way as shown in listing 3, but it's also significantly more confusing because the f function now has k as a parameter and this is only made significantly more complicated by the variable scope of functions across common languages used in Mathematics and Data science such as `bash`, *Python*, ***R*** and *Julia* (see section 1.2.1).

If however, the `for` loop approach was implemented, as shown in listing 4, the function would not significantly change, because the `reversed()` function can be used to flip the list around.

What this demonstrates is that taking a different approach to simply describing this function can lead to big differences in the complexity involved in solving this problem.

Variable Scope of Nested Functions

¶ 1.3 Fibonacci Sequence

The Fibonacci Sequence occurs in patterns observed in nature very frequently (see [27, 3, 21, 23, 17, 26]), an example of such an occurence is discussed in 1.3.3 .

```python
from sympy import *
def h(k):
    if k > 2:
        return f(k, k) * f(k, k-1)
    else:
        return 1

def f(k, i):
    if k > i:
        return 1
    if i > 2:
        return sqrt((k-i) + f(k, k - i -1))
    else:
        return 1
```

Listing 3: Using Recursion to Solve (1)

```python
from sympy import *
def h(k):
    k = k + 1 # OBOB
    l = [f(i) for i in range(1,k)]
    return prod(l)

def f(k):
    expr = 0
    for i in reversed(range(2, k+2)):
        expr = sqrt(i + expr, evaluate=False)
    return expr/(k+1)
```

Listing 4: Using Iteration to Solve (1)

This pops up all the time in natural sequences and fractals so we'll deal with ways to solve it The *Fibonacci Sequence* The *Fibonacci Sequence* and the *Golden Ratio* both occur in patterns observed in nature and are fundamentally related (see section 1.3.3), in this section we lay out a strategy to find an analytic solution to the *Fibonacci Sequence* by relating it to a continuous series and generalise this approach to any homogenous linear recurrence relation.

The hope is that by identifying relationships between discrete and continuos systems insights can be drawn with regard to the occurrence of patterns related to the *Fibonacci Sequence* and *Golden Ratio* in nature.

1.3.1 Computational Approach

The *Fibonacci* Numbers are given by:

$$F_n = F_{n-1} + F_{n-2} \tag{4}$$

This type of recursive relation can be expressed in *Python* by using recursion, as shown in listing 5, however using this function will reveal that it is extraordinarily slow, as shown in listing 6, this is because the results of the function are not cached and every time the function is called every value is recalculated[4], meaning that the workload scales in exponential as opposed to polynomial time.

The `functools` library for python includes the `@functools.lru_cache` decorator which will modify a defined function to cache results in memory [11], this means that the recursive function will only need to calculate each result once and it will hence scale in polynomial time, this is implemented in listing 7.

```
    def rec_fib(k):
        if type(k) is not int:
            print("Error: Require integer values")
            return 0
        elif k == 0:
            return 0
        elif k <= 2:
            return 1
        return rec_fib(k-1) + rec_fib(k-2)
```

Listing 5: Defining the *Fibonacci Sequence* (4) using Recursion

```
    start = time.time()
    rec_fib(35)
    print(str(round(time.time() - start, 3)) + "seconds")

## 2.245seconds
```

Listing 6: Using the function from listing 5 is quite slow.

[4] Dr. Hazrat mentions something similar in his book with respect to *Mathematica*® [13, Ch. 13]

```python
from functools import lru_cache

@lru_cache(maxsize=9999)
def rec_fib(k):
    if type(k) is not int:
        print("Error: Require Integer Values")
        return 0
    elif k == 0:
        return 0
    elif k <= 2:
        return 1
    return rec_fib(k-1) + rec_fib(k-2)


start = time.time()
rec_fib(35)
print(str(round(time.time() - start, 3)) + "seconds")
## 0.0seconds
```

Listing 7: Caching the results of the function previously defined 6

```python
start = time.time()
rec_fib(6000)
print(str(round(time.time() - start, 9)) + "seconds")

## 8.3923e-05seconds
```

Restructuring the problem to use iteration will allow for even greater performance as demonstrated by finding F_{10^6} in listing 8. Using a compiled language such as *Julia* however would be thousands of times faster still, as demonstrated in listing 9.

In this case however an analytic solution can be found by relating discrete mathematical problems to continuous ones as discussed below at section .

1.3.2 Exponential Generating Functions

Motivation

Consider the *Fibonacci Sequence* from (4):

$$\begin{aligned} a_n &= a_{n-1} + a_{n-2} \\ \iff a_{n+2} &= a_{n+1} + a_n \end{aligned} \tag{5}$$

from observation, this appears similar in structure to the following *ordinary differential equation*, which would be fairly easy to deal with:

$$f''(x) - f'(x) - f(x) = 0$$

```
    def my_it_fib(k):
        if k == 0:
            return k
        elif type(k) is not int:
            print("ERROR: Integer Required")
            return 0
        # Hence k must be a positive integer

        i  = 1
        n1 = 1
        n2 = 1

        # if k <=2:
        #     return 1

        while i < k:
           no = n1
           n1 = n2
           n2 = no + n2
           i = i + 1
        return (n1)

    start = time.time()
    my_it_fib(10**6)
    print(str(round(time.time() - start, 9)) + "seconds")

   ## 6.975890398seconds
```

Listing 8: Using Iteration to Solve the Fibonacci Sequence

```
function my_it_fib(k)
    if k == 0
        return k
    elseif typeof(k) != Int
        print("ERROR: Integer Required")
        return 0
    end
    # Hence k must be a positive integer

    i  = 1
    n1 = 1
    n2 = 1

    # if k <=2:
    #     return 1
    while i < k
       no = n1
       n1 = n2
       n2 = no + n2
       i = i + 1
    end
    return (n1)
end

@time my_it_fib(10^6)

##  my_it_fib (generic function with 1 method)
##    0.000450 seconds
```

Listing 9: Using Julia with an iterative approach to solve the 1 millionth fibonacci number

By ODE Theory we have $y \propto e^{m_i x},\ \ i = 1, 2$:

$$f(x) = e^{mx} = \sum_{n=0}^{\infty} \left[r^m \frac{x^n}{n!} \right]$$

So using some sort of a transformation involving a power series may help to relate the discrete problem back to a continuous one.

Example

Consider using the following generating function, (the derivative of the generating function as in (7) and (8) is provided in section 1.3.2)

$$f(x) = \sum_{n=0}^{\infty} \left[a_n \cdot \frac{x^n}{n!} \right] \tag{6}$$

$$\implies f'(x) = \sum_{n=0}^{\infty} \left[a_{n+1} \cdot \frac{x^n}{n!} \right] \tag{7}$$

$$\implies f''(x) = \sum_{n=0}^{\infty} \left[a_{n+2} \cdot \frac{x^n}{n!} \right] \tag{8}$$

So the recursive relation from (5) could be expressed :

$$\begin{aligned} a_{n+2} &= a_{n+1} + a_n \\ \frac{x^n}{n!} a_{n+2} &= \frac{x^n}{n!} (a_{n+1} + a_n) \\ \sum_{n=0}^{\infty} \left[\frac{x^n}{n!} a_{n+2} \right] &= \sum_{n=0}^{\infty} \left[\frac{x^n}{n!} a_{n+1} \right] + \sum_{n=0}^{\infty} \left[\frac{x^n}{n!} a_n \right] \end{aligned}$$

And hence by applying (6):

$$f''(x) = f'(x) + f(x) \tag{9}$$

Using the theory of higher order linear differential equations with constant coefficients it can be shown:

$$f(x) = c_1 \cdot \exp\left[\left(\frac{1 - \sqrt{5}}{2} \right) x \right] + c_2 \cdot \exp\left[\left(\frac{1 + \sqrt{5}}{2} \right) \right]$$

By equating this to the power series:

$$f(x) = \sum_{n=0}^{\infty} \left[\left(c_1 \left(\frac{1 - \sqrt{5}}{2} \right)^n + c_2 \cdot \left(\frac{1 + \sqrt{5}}{2} \right)^n \right) \cdot \frac{x^n}{n} \right]$$

Now given that:

$$f(x) = \sum_{n=0}^{\infty} \left[a_n \frac{x^n}{n!} \right]$$

We can conclude that:

$$a_n = c_1 \cdot \left(\frac{1-\sqrt{5}}{2} \right)^n + c_2 \cdot \left(\frac{1+\sqrt{5}}{2} \right)$$

By applying the initial conditions:

$$\begin{aligned} a_0 = c_1 + c_2 &\implies c_1 = -c_2 \\ a_1 = c_1 \left(\frac{1+\sqrt{5}}{2} \right) - c_1 \frac{1-\sqrt{5}}{2} &\implies c_1 = \frac{1}{\sqrt{5}} \end{aligned}$$

And so finally we have the solution to the *Fibonacci Sequence* 5:

$$\begin{aligned} a_n &= \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \\ &= \frac{\varphi^n - \psi^n}{\sqrt{5}} \\ &= \frac{\varphi^n - \psi^n}{\varphi - \psi} \end{aligned} \tag{10}$$

where:

- $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.61\ldots$
- $\psi = 1 - \varphi = \frac{1-\sqrt{5}}{2} \approx 0.61\ldots$

Derivative of the Exponential Generating Function

Differentiating the exponential generating function has the effect of shifting the sequence to the backward: [18]

$$f(x) = \sum_{n=0}^{\infty} \left[a_n \frac{x^n}{n!} \right] \tag{11}$$

$$\begin{aligned} f'(x)) &= \frac{\mathrm{d}}{\mathrm{d}x} \left(\sum_{n=0}^{\infty} \left[a_n \frac{x^n}{n!} \right] \right) \\ &= \frac{\mathrm{d}}{\mathrm{d}x} \left(a_0 \frac{x^0}{0!} + a_1 \frac{x^1}{1!} + a_2 \frac{x^2}{2!} + a_3 \frac{x^3}{3!} + \ldots \frac{x^k}{k!} \right) \\ &= \sum_{n=0}^{\infty} \left[\frac{\mathrm{d}}{\mathrm{d}x} \left(a_n \frac{x^n}{n!} \right) \right] \\ &= \sum_{n=0}^{\infty} \left[\frac{a_n}{(n-1)!} x^{n-1} \right] \\ \implies f'(x) &= \sum_{n=0}^{\infty} \left[\frac{x^n}{n!} a_{n+1} \right] \end{aligned} \tag{12}$$

Inductive Proof **James** This can be shown for all derivatives by way of induction, for

$$f^{(k)}(x) = \sum_{n=0}^{\infty} \frac{a_{n+k} \cdot x^n}{n!} \quad \text{for } k \geq 0 \tag{13}$$

Assume that. $f^{(k)}(x) = \sum_{n=0}^{\infty} \frac{a_{n+k} \cdot x^n}{n!}$
Using this assumption, prove for the next element $k+1$
We need $f^{(k+1)}(x) = \sum_{n=0}^{\infty} \frac{a_{n+k+1} \cdot x^n}{n!}$

$$\begin{aligned}
\text{LHS} &= f^{(k+1)}(x) \\
&= \frac{\mathrm{d}}{\mathrm{d}x}\left(f^{(k)}(x)\right) \\
&= \frac{\mathrm{d}}{\mathrm{d}x}\left(\sum_{n=0}^{\infty} \frac{a_{n+k} \cdot x^n}{n!}\right) \quad \text{by assumption} \\
&= \sum_{n=0}^{\infty} \frac{a_{n+k} \cdot n \cdot x^{n-1}}{n!} \\
&= \sum_{n=1}^{\infty} \frac{a_{n+k} \cdot x^{n-1}}{(n-1)!} \\
&= \sum_{n=0}^{\infty} \frac{a_{n+k+1} \cdot x^n}{n!} \\
&= \text{RHS}
\end{aligned}$$

Thus, if the derivative of the series shown in (6) shifts the sequence across, then every derivative thereafter does so as well, because the first derivative has been shown to express this property (12), all derivates will.

Homogeneous Proof

An equation of the form:

$$\sum_{i=0}^{n} \left[c_i \cdot f^{(i)}(x)\right] = 0 \tag{14}$$

is said to be a homogenous linear ODE: [31, Ch. 2]

Linear because the equation is linear with respect to $f(x)$

Ordinary because there are no partial derivatives (e.g. $\frac{\partial}{\partial x}(f(x))$)

Differential because the derivates of the function are concerned

Homogenous because the ***RHS*** is 0

- A non-homogeous equation would have a non-zero RHS

There will be k solutions to a k^{th} order linear ODE, each may be summed to produce a superposition which will also be a solution to the equation, [31, Ch. 4] this will be considered as the desired complete solution (and this will be shown to be the only solution for the recurrence relation (15)). These k solutions will be in one of two forms:

1. $f(x) = c_i \cdot e^{m_i x}$
2. $f(x) = c_i \cdot x^j \cdot e^{m_i x}$

where:

- $\sum_{i=0}^{k} \left[c_i m^{k-i} \right] = 0$
  - This is referred to the characteristic equation of the recurrence relation or ODE [19]
- $\exists i, j \in \mathbb{Z}^+ \cap [0, k]$
  - These is often referred to as repeated roots [19, 32] with a multiplicity corresponding to the number of repetitions of that root [24, §3.2]

Unique Roots of Characteristic Equation

1. Example An example of a recurrence relation with all unique roots is the fibonacci sequence, as described in section 1.3.2 .
2. Proof Consider the linear recurrence relation (15):

$$\sum_{i=0}^{n} [c_i \cdot a_i] = 0, \quad \exists c \in \mathbb{R}, \ \forall i < k \in \mathbb{Z}^+$$

This implies:

$$\sum_{n=0}^{\infty} \left[\sum_{i=0}^{k} \left[\frac{x^n}{n!} c_i a_n \right] \right] = 0 \tag{15}$$

$$\sum_{n=0}^{\infty} \sum_{i=0}^{k} \frac{x^n}{n!} c_i a_n = 0 \tag{16}$$

$$\sum_{i=0}^{k} c_i \sum_{n=0}^{\infty} \frac{x^n}{n!} a_n = 0 \tag{17}$$

By implementing the exponential generating function as shown in (6), this provides:

$$\sum_{i=0}^{k} \left[c_i f^{(i)}(x) \right] \tag{18}$$

Now assume that the solution exists and all roots of the characteristic polynomial are unique (i.e. the solution is of the form $f(x) \propto e^{m_i x}$: $\quad m_i \neq m_j \forall i \neq j$), this implies that [31, Ch. 4] :

$$f(x) = \sum_{i=0}^{k} [k_i e^{m_i x}], \quad \exists m, k \in \mathbb{C}$$

This can be re-expressed in terms of the exponential power series, in order to relate the solution of the function $f(x)$ back to a solution of the sequence a_n, (see section for a derivation of the exponential power series):

$$\begin{aligned}\sum_{i=0}^{k}[k_i e^{m_i x}] &= \sum_{i=0}^{k}\left[k_i \sum_{n=0}^{\infty}\frac{(m_i x)^n}{n!}\right] \\ &= \sum_{i=0}^{k}\sum_{n=0}^{\infty} k_i m_i^n \frac{x^n}{n!} \\ &= \sum_{n=0}^{\infty}\sum_{i=0}^{k} k_i m_i^n \frac{x^n}{n!} \\ &= \sum_{n=0}^{\infty}\left[\frac{x^n}{n!}\sum_{i=0}^{k}[k_i m_i^n]\right], \quad \exists k_i \in \mathbb{C},\ \ \forall i \in \mathbb{Z}^+ \cap [1,k] \end{aligned} \tag{19}$$

Recall the definition of the generating function from **??**, by relating this to (19):

$$\begin{aligned} f(x) &= \sum_{n=0}^{\infty}\left[\frac{x^n}{n!}a_n\right] \\ &= \sum_{n=0}^{\infty}\left[\frac{x^n}{n!}\sum_{i=0}^{k}[k_i m_i^n]\right] \\ \implies a_n &= \sum_{n=0}^{k}[k_i m_i^n] \end{aligned}$$

□

This can be verified by the fibonacci sequence as shown in section 1.3.2 , the solution to the characteristic equation is $m_1 = \varphi, m_2 = (1-\varphi)$ and the corresponding solution to the linear ODE and recursive relation are:

$$\begin{aligned} f(x) =& c_1 e^{\varphi x} + c_2 e^{(1-\varphi)x}, \quad \exists c_1, c_2 \in \mathbb{R} \subset \mathbb{C} \\ \iff a_n =& k_1 n^{\varphi} + k_2 n^{1-\varphi}, \quad \exists k_1, k_2 \in \mathbb{R} \subset \mathbb{C} \end{aligned}$$

Repeated Roots of Characteristic Equation

1. Example Consider the following recurrence relation:

$$\begin{aligned} a_n - 10a_{n+1} + 25a_{n+2} &= 0 \\ \implies \sum_{n=0}^{\infty}\left[a_n\frac{x^n}{n!}\right] - 10\sum_{n=0}^{\infty}\left[\frac{x^n}{n!}+\right] + 25\sum_{n=0}^{\infty}\left[a_{n+2}\frac{x^n}{n!}\right] &= 0 \end{aligned} \tag{20}$$

By applying the definition of the exponential generating function at (6) :

$$f''(x) - 10f'(x) + 25f(x) = 0$$

By implementing the already well-established theory of linear ODE's, the characteristic equation for (**??**) can be expressed as:

$$\begin{aligned} m^2 - 10m + 25 &= 0 \\ (m-5)^2 &= 0 \\ m &= 5 \end{aligned} \tag{21}$$

Herein lies a complexity, in order to solve this, the solution produced from (21) can be used with the *Reduction of Order* technique to produce a solution that will be of the form [32, §4.3].

$$f(x) = c_1 e^{5x} + c_2 x e^{5x} \tag{22}$$

(22) can be expressed in terms of the exponential power series in order to try and relate the solution for the function back to the generating function, observe however the following power series identity (TODO Prove this in section):

$$x^k e^x = \sum_{n=0}^{\infty} \left[\frac{x^n}{(n-k)!} \right], \quad \exists k \in \mathbb{Z}^+ \tag{23}$$

by applying identity (23) to equation (22)

$$\begin{aligned} \implies f(x) &= \sum_{n=0}^{\infty} \left[c_1 \frac{(5x)^n}{n!} \right] + \sum_{n=0}^{\infty} \left[c_2 n \frac{(5x^n)}{n(n-1)!} \right] \\ &= \sum_{n=0}^{\infty} \left[\frac{x^n}{n!} (c_1 5^n + c_2 n 5^n) \right] \end{aligned}$$

Given the defenition of the exponential generating function from (6)

$$\begin{aligned} f(x) &= \sum_{n=0}^{\infty} \left[a_n \frac{x^n}{n!} \right] \\ \iff a_n &= c_{15}^n + c_2 n_5^n \end{aligned}$$

□

2. Proof In order to prove the the solution for a k^{th} order recurrence relation with k repeated
   Consider a recurrence relation of the form:

$$\begin{aligned} \sum_{n=0}^{k} [c_i a_n] &= 0 \\ \implies \sum_{n=0}^{\infty} \sum_{i=0}^{k} c_i a_n \frac{x^n}{n!} &= 0 \\ \sum_{i=0}^{k} \sum_{n=0}^{\infty} c_i a_n \frac{x^n}{n!} & \end{aligned}$$

By substituting for the value of the generating function (from (6)):

$$\sum_{i=0}^{k}\Big[c_i f^{(k)}(x)\Big] \tag{24}$$

Assume that (24) corresponds to a charecteristic polynomial with only 1 root of multiplicity k, the solution would hence be of the form:

$$\sum_{i=0}^{k}\Big[c_i m^i\Big] = 0 \wedge m = B,\ \exists! B \in \mathbb{C}$$

$$\implies f(x) = \sum_{i=0}^{k}\Big[x^i A_i e^{mx}\Big],\quad \exists A \in \mathbb{C}^+,\ \forall i \in [1,k] \cap \mathbb{N} \tag{25}$$

$$\tag{26}$$

If we assume that (see section 1):

$$k \in \mathbb{Z} \implies x^k e^x = \sum_{n=0}^{\infty}\left[\frac{x^n}{(n-k)!}\right] \tag{27}$$

By applying this to (25) :

$$\begin{aligned} f(x) &= \sum_{i=0}^{k}\left[A_i \sum_{n=0}^{\infty}\left[\frac{(xm)^n}{(n-i)!}\right]\right] \\ &= \sum_{n=0}^{\infty}\left[\sum_{i=0}^{k}\left[\frac{x^n}{n!}\frac{n!}{(n-i)}A_i m^n\right]\right] \qquad (28)\\ &= \sum_{n=0}^{\infty}\left[\frac{x^n}{n!}\sum_{i=0}^{k}\left[\frac{n!}{(n-i)}A_i m^n\right]\right] \qquad (29)\end{aligned}$$

Recall the generating function that was used to get 24:

$$\begin{aligned} f(x) &= \sum_{n=0}^{\infty}\left[a_n \frac{x^n}{n!}\right] \\ \implies a_n &= \sum_{i=0}^{k}\left[A_i \frac{n!}{(n-i)!}m^n\right] \\ &= \sum_{i=0}^{k}\left[m^n A_i \prod_{0}^{k}[n-(i-1)]\right] \qquad (30)\end{aligned}$$

$\because \ \ i \leq k$

$$= \sum_{i=0}^{k} \left[A_i^* m^n n^i \right], \quad \exists A_i \in \mathbb{C}, \ \ \forall i \in \mathbb{Z}^+$$

□

General Proof In sections 1.3.2 and 1.3.2 it was shown that a recurrence relation can be related to an ODE and then that solution can be transformed to provide a solution for the recurrence relation, when the charecteristic polynomial has either complex roots or 1 repeated root. Generally the solution to a linear ODE will be a superposition of solutions for each root, repeated or unique and so a goal of our research will be to put this together to find a general solution for homogenous linear recurrence relations.

Sketching out an approach for this:

- Use the Generating function to get an ODE
- The ODE will have a solution that is a combination of the above two forms
- The solution will translate back to a combination of both above forms

1. Power Series Combination JAMES In this section a proof for identity 27 is provided.

   (a) Motivation

   Consider the function $f(x) = xe^x$. Using the taylor series formula we get the following:

$$\begin{aligned} xe^x &= 0 + \frac{1}{1!}x + \frac{2}{2!}x^2 + \frac{3}{3!}x^3 + \frac{4}{4!}x^4 + \frac{5}{5!}x^5 + \dots \\ &= \sum_{n=0}^{\infty} \frac{nx^n}{n!} \\ &= \sum_{n=1}^{\infty} \frac{x^n}{(n-1)!} \end{aligned}$$

   Similarly, $f(x) = x^2e^x$ will give:

$$\begin{aligned} x^2e^x &= \frac{0}{0!} + \frac{0x}{1!} + \frac{2x^2}{2!} + \frac{6x^3}{3!} + \frac{12x^4}{4!} + \frac{20x^5}{5!} + \dots \\ &= \frac{2 \cdot 1x^2}{2!} + \frac{3 \cdot 2x^3}{3!} + \frac{4 \cdot 3x^4}{4!} + \frac{5 \cdot 4x^5}{5!} + \dots \\ &= \sum_{n=2}^{\infty} \frac{n(n-1)x^n}{n!} \\ &= \sum_{n=2}^{\infty} \frac{x^n}{(n-2)!} \end{aligned}$$

   We conjecture thatIf we continue this on, we get:

$$x^k e^x = \sum_{n=k}^{\infty} \frac{x^n}{(n-k)!} \quad \text{for } k \in \mathbb{Z}^+$$

(b) Proof by Induction To verify, let's prove this by induction.

i. Base Test $k = 0$

$$LHS = x^0 e^x = e^x$$
$$RHS = \sum_{n=0}^{\infty} \frac{x^n}{n!} = e^x$$

Therefore LHS = RHS, so $k = 0$ is true

ii. Bridge Assume $x^k e^x = \sum_{n=k}^{\infty} \frac{x^n}{(n-k)!}$
Using this assumption, prove for the next element $k+1$

We need $x^{k+1}e^x = \sum_{n=k+1}^{\infty} x^n \frac{}{(n-(k+1))!\$Sobymathematicalinductionf(x)=x^k e^x=\sum_{n=k}^{\infty} \frac{x^n}{(n-k)!}}$ for $k \geq 0$
Moving on, by applying identity (23) to equation (22)

1.3.3 Fibonacci Sequence and the Golden Ratio

The *Fibonacci Sequence* is actually very interesting, observe that the ratios of the terms converge to the *Golden Ratio*:

$$\begin{aligned}
F_n &= \frac{\varphi^n - \psi^n}{\varphi - \psi} = \frac{\varphi^n - \psi^n}{\sqrt{5}} \\
\iff \frac{F_{n+1}}{F_n} &= \frac{\varphi^{n+1} - \psi^{n+1}}{\varphi^n - \psi^n} \\
\iff \lim_{n\to\infty}\left[\frac{F_{n+1}}{F_n}\right] &= \lim_{n\to\infty}\left[\frac{\varphi^{n+1} - \psi^{n+1}}{\varphi^n - \psi^n}\right] \\
&= \frac{\varphi^{n+1} - \lim_{n\to\infty}\left[\psi^{n+1}\right]}{\varphi^n - \lim_{n\to\infty}\left[\psi^n\right]}
\end{aligned}$$

because $|\ \psi\ | < 0\ n \to \infty \implies \psi^n \to 0$:

$$\begin{aligned}
&= \frac{\varphi^{n+1} - 0}{\varphi^n - 0} \\
&= \varphi
\end{aligned}$$

We'll come back to this later on when looking at spirals and fractals.

We hope to demonstrate this relationship between the ratio of successive terms of the fibonacci sequence without relying on ODEs and generating functions and by instead using limits and the *Monotone Convergence Theorem*, the hope being that this will reveal deeper underlying relationships between the *Fibonacci Sequence*, the *Golden Ratio* and there occurrences in nature (such as the example in section 1.3.3 given that the both appear to occur in patterns observed in nature.

Fibonacci Sequence in Nature — RYAN

The distribution of sunflower seeds is an example of the *Fibonacci Sequence* occuring in a pattern observed in nature (see Figure 3).

Imagine that the process a sunflower follows when placing seeds is as follows: [5]

1. Place a seed
2. Move some small unit away from the origin
3. Rotate some constant angle θ (or) from the previous seed (with respect to the origin).
4. Repeat this process until a seed hits some outer boundary.

This process can be simulated in Julia [4] as shown in listing 10, which combined with *ImageMagick* (see e.g. 8), produces output as shown in figure 1 and 2.

A distribution of seeds undder this process would be optimal if the amount of empty space was minimised, spirals, stars and swirls contain patterns compromise this.

To minimize this, the proportion of the circle traversed in step 3 must be an irrational number, however this alone is not sufficent, the decimal values must also be not to approximated by a rational number, for example [23]:

- $\pi \mod 1 \approx \frac{1}{7} = 0.7142857142857143$
- $e \mod 1 \approx \frac{5}{7} = 0.14285714285714285$

It can be seen by simulation that ϕ and ψ (because $\phi \mod 1 = \psi$) are solutions to this optimisation problem as shown in figure 2, this solution is unstable, a very minor change to the value will result in patterns re-emerging in the distribution.

Another interesting property is that the number of spirals that appear to rotate clockwise and anti-clockwise appear to be fibonacci numbers. Connecting this occure with the relationship between the *Fibonacci Sequence* as discussed in section 1.3.3 is something we hope to look at in this project. Illustrating this phenomena with *Julia* by finding the mathematics to colour the correct spirals is also something we intend to look at in this project.

The bottom right spiral in figure 1 has a ratio of rotation of $\frac{1}{\pi}$, the spirals look similar to one direction of the spirals occuring in figure 2, it is not clear if there is any significance to this similarity.

¶ 1.4 Persian Recursion

Although some recursive problems are a good fit for mathematical thinking such as the *Fibonacci Sequence* discussed in section 1.3.2 other problems can be be easily interpreted computationally but they don't really carry over to any mathematical perspective, one good example of this is *the persian recursion*, which is a simple procedure developed by Anne Burns in the 90s [8] that produces fantastic patterns upon feedback and iteration

The procedure is illustrated in figure 4 begins with an empty or zero square matrix with sides $2^n + 1$, $\exists n \in \mathbb{Z}^+$ and some value given to the edges:

1. Decide on some four variable function with a finite domain and range of size m, for the example shown at listing 11 and in figure 5 the function $f(w, x, y, z) = (w + x + y + z) \mod m$ was chosen.
2. Assign this value to the centre row and centre column of the matrix

[5] This process is simply conjecture, other than seeing a very nice example at *MathIsFun.com* [23], we have no evidence to suggest that this is the way that sunflowers distribute there seeds.

However the simulations performed within *Julia* are very encouraging and suggest that this process isn't too far off.

```
 = 1.61803398875
 = ^-1
 = 0.61803398875
function sfSeeds(ratio)
 = Turtle()
    for  in [(ratio*2*)*i for i in 1:3000]
        gsave()
        scale(0.05)
        rotate()
#        Pencolor(, rand(1)[1], rand(1)[1], rand(1)[1])
        Forward(, 1)
        Rectangle(, 50, 50)
        grestore()
    end
    label = string("Ratio = ", round(ratio, digits = 8))
    textcentered(label, 100, 200)
end
@svg begin
    sfSeeds()
end 600 600
```

Listing 10: Simulation of the distribution of sunflowers as described in section 1.3.3

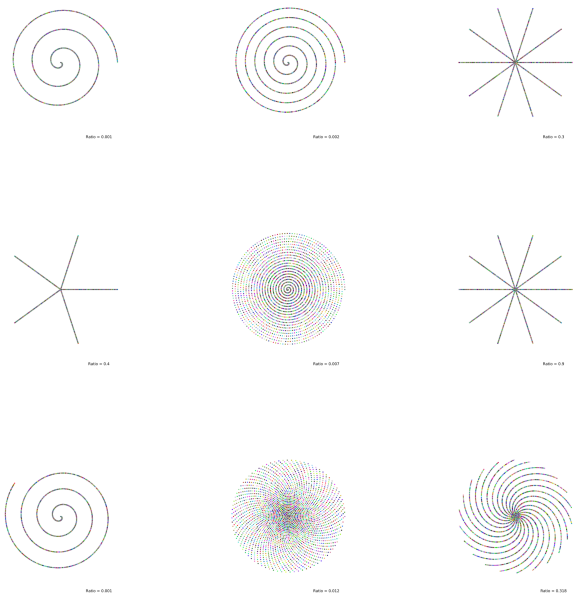


Figure 1: Simulated Distribution of Sunflower seeds as described in section 1.3.3 and listing 10

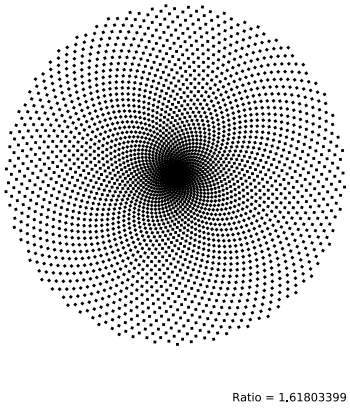


Figure 2: Optimisation of simulated distribution of Sunflower seeds occurs for $\theta = 2\varphi\pi$ as described in section 1.3.3 and listing 10

Figure 3: Distribution of the seeds of a sunflower (see [7] licenced under CC)

3. Repeat this for each newly enclosed subsmatrix.

This can be implemented computationally by defining a function that:

- takes the index of four corners enclosing a square sub-matrix of some matrix as input,
- proceeds only if that square is some positive real value.
- colours the centre column and row corresponding to a function of those four values
- then calls itself on the corners of the four new sub-matrices enclosed by the coloured row and column

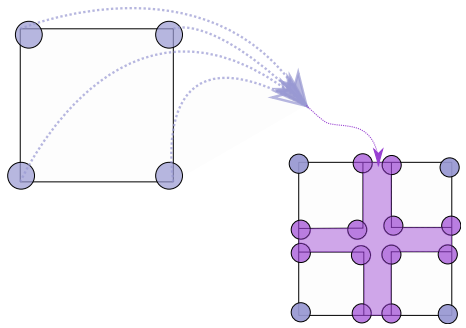


Figure 4: Diagram of the Persian Recursion, implemented with *Python* in listing 11

This is demonstrated in listing 11 with python and produces the output shown in figures 5, various interesting examples are provided in the appendix at section ¶ 8.1 .

By mapping the values to colours, patterns emerge, this emergence of complex patterns from simple rules is a well known and general phenomena that occurs in nature [9, 16], as a matter of fact:

> One of the suprising impacts of fractal geometry is that in the presence of complex patterns there is a good chance that a very simple process is responsible for it.

Many patterns that occur in nature can be explained by relatively simple rules that are exposed to feedback and iteration [25, p. 16], this is a centreal theme of Alan Turing's *The Chemical Basis For Morphogenesis* [30] which we hope to look in the course of this research.

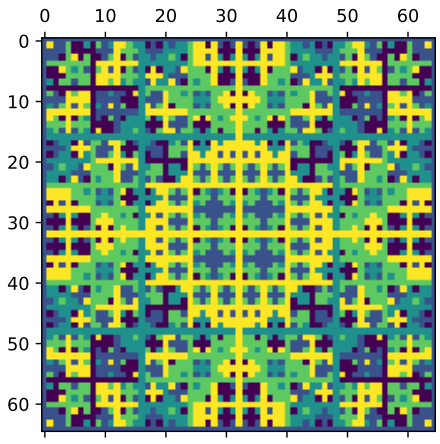


Figure 5: Output produced by listing 11 with 6 folds

```
%matplotlib inline
# m is colours
# n is number of folds
# Z is number for border
# cx is a function to transform the variables
def main(m, n, z, cx):
    import numpy as np
    import matplotlib.pyplot as plt

    # Make the Empty Matrix
    mat = np.empty([2**n+1, 2**n+1])
    main.mat = mat

    # Fill the Borders
    mat[:,0] = mat[:,-1] = mat[0,:] = mat[-1,:] = z

    # Colour the Grid
    colorgrid(0, mat.shape[0]-1, 0, mat.shape[0]-1, m)

    # Plot the Matrix
    plt.matshow(mat)

# Define Helper Functions
def colorgrid(l, r, t, b, m):
    # print(l, r, t, b)
    if (l < r -1):
        ## define the centre column and row
        mc = int((l+r)/2); mr = int((t+b)/2)

        ## Assign the colour
        main.mat[(t+1):b,mc] = cx(l, r, t, b, m)
        main.mat[mr,(l+1):r] = cx(l, r, t, b, m)

        ## Now Recall this function on the four new squares
                #l r   t   b
        colorgrid(l, mc, t, mr, m)    # NW
        colorgrid(mc, r, t, mr, m)    # NE
        colorgrid(l, mc, mr, b, m)    # SW
        colorgrid(mc, r, mr, b, m)    # SE

def cx(l, r, t, b, m):
    new_col = (main.mat[t,l] + main.mat[t,r] +  main.mat[b,l] +
    ↪  main.mat[b,r]) % m
    return new_col.astype(int)

main(5,6, 1, cx)
```

Listing 11: Implementation of the persian recursion scheme in *Python*

¶ 1.5 Julia

1.5.1 Motivation

Consider the iterative process $x \to x^2,\ \ x \in \mathbb{R}$, for values of $x > 1$ this process will diverge and for $x < 1$ it will converge.

Now Consider the iterative process $z \to z^2,\ \ z \in \mathbb{C}$, for values of $|z| > 1$ this process will diverge and for $|z| < 1$ it will converge.

Although this seems trivial this can be generalised.

Consider:

- The complex plane for $|z| \leq 1$
- Some function $f_c(z) = z^2 + c, \quad c \leq 1 \in \mathbb{C}$ that can be used to iterate with

Every value on that plane will belong to one of the two following sets

- P_c
  - The set of values on the plane that converge to zero (prisoners)
  - Define $Q_c^{(k)}$ to be the the set of values confirmed as prisoners after k iterations of f_c
    - this implies $\lim_{k\to\infty}\left[Q_c^{(k)}\right] = P_c$
- E_c
  - The set of values on the plane that tend to ∞ (escapees)

In the case of $f_0(z) = z^2$ all values $|z| \leq 1$ are bounded with $|z| = 1$ being an unstable stationary circle, but let's investigate what happens for different iterative functions like $f_1(z) = z^2 - 1$, despite how trivial this seems at first glance.

1.5.2 Plotting the Sets **ATTACH**

Although the convergence of values may appear simple at first, we'll implement a strategy to plot the prisoner and escape sets on the complex plane.

Because this involves iteration and *Python* is a little slow, We'll denote complex values as a vector[6] and define the operations as described in listing 12.[7]

To implement this test we'll consider a function called `escape_test` that applies an iteration (in this case $f_0 : z \to z^2$) until that value diverges or converges.

While iterating with f_c once $|z| > \max(\{c, 2\})$, the value must diverge because $|c| \leq 1$, so rather than record whether or not the value converges or diverges, the `escape_test` can instead record the number of iterations (k) until the value has crossed that boundary and this will provide a measurement of the rate of divergence.

Then the `escape_test` function can be mapped over a matrix, where each element of that matrix is in turn mapped to a point on the cartesian plane, the resulting matrix can be visualised as an image [8], this is implemented in listing 13 and the corresponding output shown in .

with respect to listing 13:

[6] See figure for the obligatory *XKCD* Comic

[7] This technique was adapted from Chapter 7 of *Math adventures with Python* [10]

[8] these cascading values are much like brightness in Astronomy

- Observe that the `magnitude` function wasn't used:

    1. This is because a `sqrt` is a costly operation and comparing two squares saves an operation

```python
from math import sqrt
def magnitude(z):
    # return sqrt(z[0]**2 + z[1]**2)
    x = z[0]
    y = z[1]
    return sqrt(sum(map(lambda x: x**2, [x, y])))

def cAdd(a, b):
    x = a[0] + b[0]
    y = a[1] + b[1]
    return [x, y]


def cMult(u, v):
    x = u[0]*v[0]-u[1]*v[1]
    y = u[1]*v[0]+u[0]*v[1]
    return [x, y]
```

Listing 12: Defining Complex Operations with vectors

This is precisely what we expected, but this is where things get interesting, consider now the result if we apply this same procedure to $f_1 : z \to z^2 - 1$ or something arbitrary like $f_{\frac{1}{4}+\frac{i}{2}} : z \to z^2 + (\frac{1}{4} + \frac{i}{2})$, the result is something particularly unexpected, as shown in figures 6 and 7.

Now this is particularly interesting, to investigate this further consider the more general function $f_{0.8e^{\pi i\tau}} : z \to z^2 + 0.8e^{\pi i\tau},\ \tau \in \mathbb{R}$, many fractals can be generated using this set by varying the value of τ[9].

Python is too slow for this, but the *Julia* programming language, as a compiled language, is significantly faster and has the benefit of treating complex numbers as first class citizens, these images can be generated in *Julia* in a similar fashion as before, with the specifics shown in listing 14. The `GR` package appears to be the best plotting library performance wise and so was used to save corresponding images to disc, this is demonstrated in listing 15 where 1200 pictures at a 2.25 MP resolution were produced. [10]

A subset of these images can be combined using *ImageMagick* and `bash` to create a collage, *ImageMagick* can also be used to produce a `gif` but it often fails and a superior approach is to use `ffmpeg`, this is demonstrated in listing 16, the collage is shown in figure 8 and a corresponding animation is available online[11]].

¶ 1.6 MandelBrot

Investigating these fractals, a natural question might be whether or not any given c value will produce a fractal that is an open disc or a closed disc.

[9] This approach was inspired by an animation on the *Julia Set* Wikipedia article [15]
[10] On my system this took about 30 minutes.
[11] https://dl.dropboxusercontent.com/s/rbu25urfg8sbwfu/out.gif?dl=0

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'
import numpy as np
def escape_test(z, num):
    ''' runs the process num amount of times and returns the count of
    divergence'''
    c = [0, 0]
    count = 0
    z1 = z  #Remember the original value that we are working with
    # Iterate num times
    while count <= num:
        dist = sum([n**2 for n in z1])
        distc = sum([n**2 for n in c])
        # check for divergence
        if dist > max(2, distc):
            #return the step it diverged on
            return count
        #iterate z
        z1 = cAdd(cMult(z1, z1), c)
        count+=1
        #if z hasn't diverged by the end
    return num



p = 0.25 #horizontal, vertical, pinch (zoom)
res = 200
h = res/2
v = res/2

pic = np.zeros([res, res])
for i in range(pic.shape[0]):
    for j in range(pic.shape[1]):
        x = (j - h)/(p*res)
        y = (i-v)/(p*res)
        z = [x, y]
        col = escape_test(z, 100)
        pic[i, j] = col

import matplotlib.pyplot as plt

plt.axis('off')
plt.imshow(pic)
# plt.show()
```

Listing 13: Circle of Convergence of z under recursion

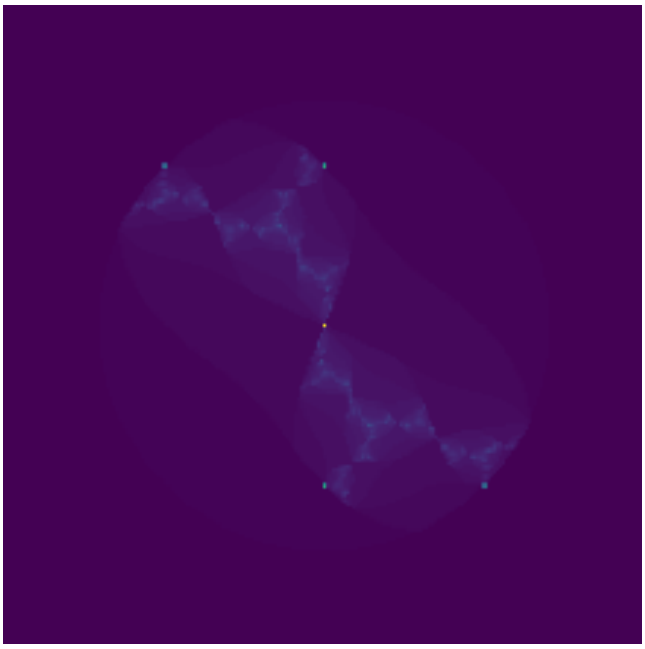


Figure 6: Circle of Convergence for $f_0 : z \to z^2 - 1$

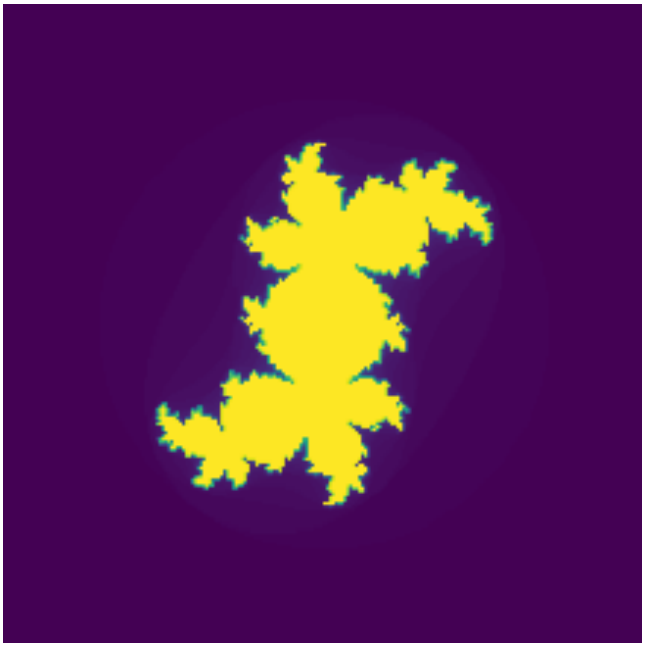


Figure 7: Circle of Convergence for $f_{\frac{1}{4}+\frac{i}{2}} : z \to z^2 + \frac{1}{4} + \frac{i}{2}$

Figure 8: Various fracals corresponding to $f_{0.8e^{\pi i \tau}}$

```
# * Define the Julia Set
"""
Determine whether or not a value will converge under iteration
"""
function juliaSet(z, num, my_func)
    count = 1
    # Remember the value of z
    z1 = z
    # Iterate num times
    while count  num
        # check for divergence
        if abs(z1)>2
            return Int(count)
        end
        #iterate z
        z1 = my_func(z1) # + z
        count=count+1
    end
        #if z hasn't diverged by the end
    return Int(num)
end

# * Make a Picture
"""
Loop over a matrix and apply apply the julia-set function to
the corresponding complex value
"""
function make_picture(width, height, my_func)
    pic_mat = zeros(width, height)
    zoom = 0.3
    for i in 1:size(pic_mat)[1]
        for j in 1:size(pic_mat)[2]
            x = (j-width/2)/(width*zoom)
            y = (i-height/2)/(height*zoom)
            pic_mat[i,j] = juliaSet(x+y*im, 256, my_func)
        end
    end
    return pic_mat
end
```

Listing 14: Produce a series of fractals using julia

```
# * Use GR to Save a Bunch of Images
  ## GR is faster than PyPlot
using GR
function save_images(count, res)
    try
        mkdir("/tmp/gifs")
    catch
    end
    j = 1
    for i in (1:count)/(40*2*)
        j = j + 1
        GR.imshow(make_picture(res, res, z -> z^2 + 0.8*exp(i*im*9/2))) #
        ↪  PyPlot uses interpolation = "None"
        name = string("/tmp/gifs/j", lpad(j, 5, "0"), ".png")
        GR.savefig(name)
    end
end

save_images(1200, 1500) # Number  and Res
```

Listing 15: Generate and save the images with GR

```
# Use montage multiple times to get recursion for fun
montage (ls *png | sed -n '1p;0~600p') 0a.png
montage (ls *png | sed -n '1p;0~100p') a.png
montage (ls *png | sed -n '1p;0~50p') -geometry 1000x1000  a.png

# Use ImageMagick to Produce a gif (unreliable)
convert -delay 10 *.png 0.gif

# Use FFMpeg to produce a Gif instead
ffmpeg                      \
    -framerate 60           \
    -pattern_type glob      \
    -i '*.png'              \
    -r 15                   \
    out.mov
```

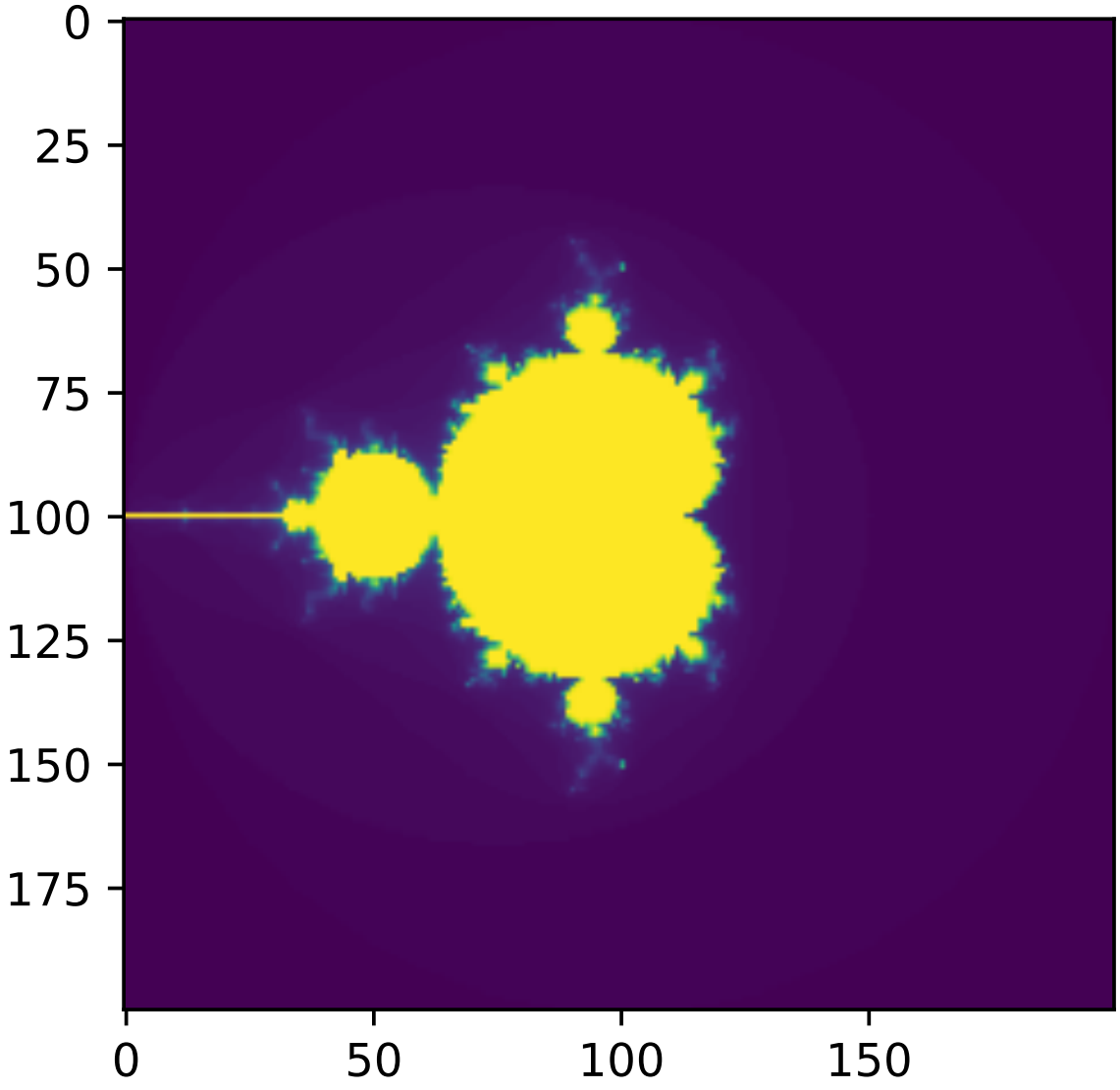
Listing 16: Using `bash`, `ffmpeg` and *ImageMagick* to combine the images and produce an animation.

So pick a value $|\gamma| < 1$ in the complex plane and use it to produce the julia set f_γ, if the corresponding prisoner set P is closed we this value is defined as belonging to the *Mandelbrot* set.

It can be shown (and I intend to show it generally), that this set is equivalent to re-implementing the previous strategy such that $z \to z^2 + z_0$ where z_0 is unchanging or more clearly as a seqeuence:

$$z_{n+1} = z_n^2 + cz_0 \qquad = c \tag{31}$$

This strategy is implemented in listing



This is however fairly underwhelming, by using a more powerful language a much larger image can be produced, in *Julia* producing a 4 GB, 400 MP image will take about 10 minutes, this is demonstrated in listing and the corresponding FITS image is available-online.[12]

[12] https://www.dropbox.com/s/jd5qf1pi2h68f2c/mandelbrot-400mpx.fits?dl=0

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'
def mandelbrot(z, num):
    ''' runs the process num amount of times and returns the count of
    divergence'''
    count = 0
    # Define z1 as z
    z1 = z
    # Iterate num times
    while count <= num:
        # check for divergence
        if magnitude(z1) > 2.0:
            #return the step it diverged on
            return count
        #iterate z
        z1 = cAdd(cMult(z1, z1),z)
        count+=1
        #if z hasn't diverged by the end
    return num

import numpy as np


p = 0.25 # horizontal, vertical, pinch (zoom)
res = 200
h = res/2
v = res/2

pic = np.zeros([res, res])
for i in range(pic.shape[0]):
    for j in range(pic.shape[1]):
        x = (j - h)/(p*res)
        y = (i-v)/(p*res)
        z = [x, y]
        col = mandelbrot(z, 100)
        pic[i, j] = col

import matplotlib.pyplot as plt
plt.imshow(pic)
# plt.show()
```

Listing 17: All values of c that lead to a closed *Julia-set*

```
function mandelbrot(z, num, my_func)
    count = 1
    # Define z1 as z
    z1 = z
    # Iterate num times
    while count  num
        # check for divergence
        if abs(z1)>2
            return Int(count)
        end
        #iterate z
        z1 = my_func(z1) + z
        count=count+1
    end
        #if z hasn't diverged by the end
    return Int(num)
end

function make_picture(width, height, my_func)
    pic_mat = zeros(width, height)
    for i in 1:size(pic_mat)[1]
        for j in 1:size(pic_mat)[2]
            x = j/width
            y = i/height
            pic_mat[i,j] = mandelbrot(x+y*im, 99, my_func)
        end
    end
    return pic_mat
end


using FITSIO
function save_picture(filename, matrix)
    f = FITS(filename, "w");
    # data = reshape(1:100, 5, 20)
    # data = pic_mat
    write(f, matrix)  # Write a new image extension with the data

    data = Dict("col1"=>[1., 2., 3.], "col2"=>[1, 2, 3]);
    write(f, data)  # write a new binary table to a new extension

    close(f)
end

# * Save Picture
#-------------------------------------------------------------
my_pic = make_picture(20000, 20000, z -> z^2) 2000^2 is 4 GB
save_picture("/tmp/a.fits", my_pic)
```

1.6.1 GNU Plot

Another approach to visualise this set is by creating a 3d surface plot where the z-axis is mapped to the time taken until divergence, this can be acheived by using gnuplot as demonstrated in listing 18.[13]

All the following code was adapted from online sources, they however correspond to an older release and newer versions of GNUPlot:

- have a recursion limit
- methods to loop functions

So one of our desires in this project is to visualise a much more detailed 3d model in GNUPlot by modifying the code to use iteration as opposed to recursion. Generally I would like to write a

I'd also like to implement some process to generate a golden spiral as shown in figure 9.

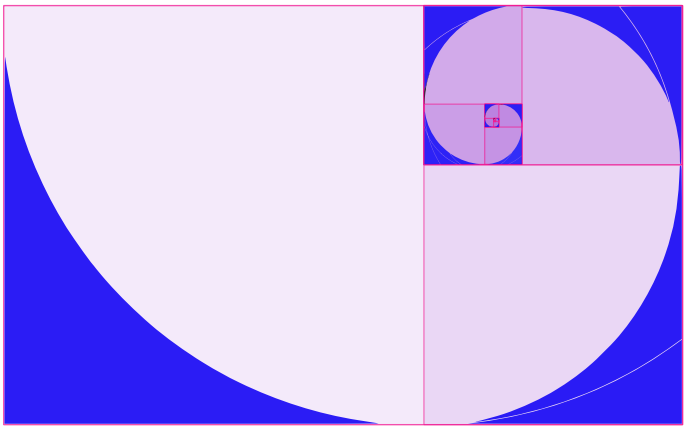


Figure 9: Circle of Convergence for $f_0 : z \to z^2 - 1$

[13] See [12] for an excellent, albeit quite old, resource on GNUPlot.

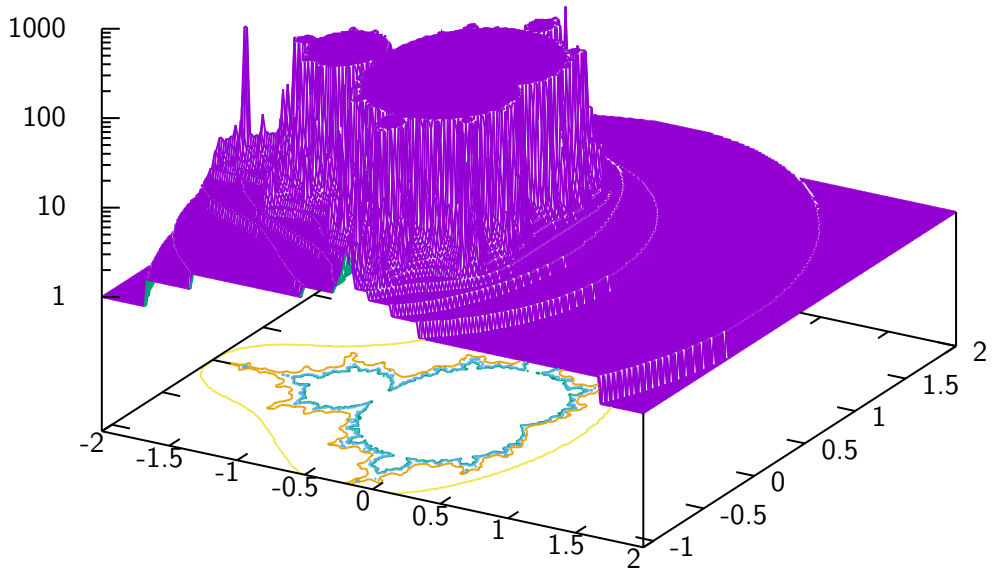
```
complex(x,y) = x*{1,0}+y*{0,1}
julia(x,y,z,n) = (abs(z)>2.0 || n>=200) ? \
                  n : julia(x,y,z*z+complex(x,y),n+1)

mandelbrot(x,y,z,n) = (abs(z)>2.0 || n>=200) ? \
                  n : mandelbrot(x,y,z*z+complex(x,y),n+1)

set xrange [-1.5:1.5]
set yrange [-1.5:1.5]
set logscale z
set isosample 150
set hidden3d
set contour
a= 0.36
b= 0.1
# Julia Set
splot julia(a,b,complex(x,y),0)  notitle

# MandelBrot Set
# splot mandelbrot(x,y,complex(x,y),0) notitle
```

Listing 18: Visualising the Mandelbrot set as a 3D surface Plot

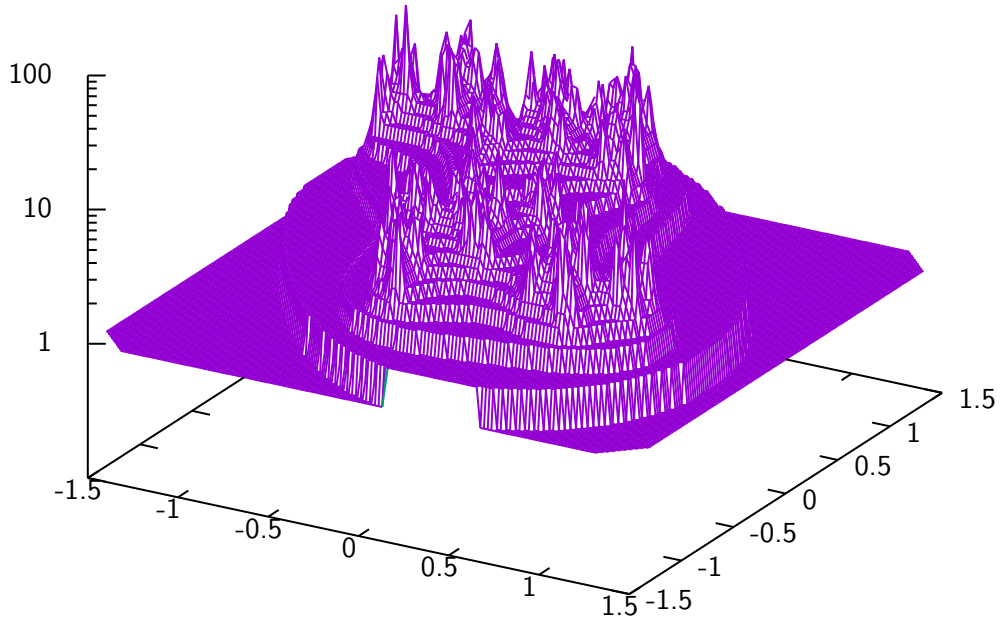


reference for image

```
complex(x,y) = x*{1,0}+y*{0,1}
julia(x,y,z,n) = (abs(z)>2.0 || k>=200) ? \
                  k : julia(x,y,z*z+complex(x,y),n+1)

set xrange [-1.5:1.5]
set yrange [-1.5:1.5]
set logscale z
set isosample 150
set hidden3d
set contour
a= 0.25
b= 0.75
splot julia(a,b,complex(x,y),0) notitle
```

Listing 19: Use GNUPlot to produce plot of julia set



GNU Plot can also make excellent 2d renditions of fractals, an example of how to perform this can be found on *Rosetta Code* [20] and is demonstrated in listing 20.

```
R = 2
k = 100
complex (x, y) = x * {1, 0} + y * {0, 1}
mandelbrot (z, z0, n) = n == k || abs (z) > R ? n : mandelbrot (z ** 2 + z0,
↪  z0, n + 1)
set samples 200
set isosamples 200
set pm3d map
set size square
splot [-2 : 2] [-2 : 2] mandelbrot (complex (0, 0), complex (x, y), 0) notitle
```

Listing 20: Flat Mandelbrot set built using rosetta code.

¶ 1.7 Relevant Sources

To guide research in this area the following books were going to act as guides, in particular *Chaos and Fractals* by Otto is a primary resource:

List the books from Dropbox here.

Also Ron Knotts website looks very helpful [26]

§ 2 Outline

1. Intro Prob
2. Variable Scope
3. Problem Showing Recursion
    - All Different Methods
        - Discuss all Different Methods
        - Discuss Vectorisation
        - Is this needed in Julia
        - Comment on Faster to go column Wise
4. Discuss Loops
5. Show Rug
6. Fibonacci
    - The ratio of fibonacci converges to ϕ
    - Golden Ratio
        - If you make a rectangle with the golden ratio you can cut it up under recursion to get another one, keep doing this and eventually a logarithmic spiral pops out, also the areas follow a fibonacci sequence.
        - Look at the spiral of nautilus shells
7. Discuss isomorphisms for recursive Relations

8. Jump to Lorenz Attractor
9. Now Talk about Morphogenesis
10. Fractals
    - Many Occur in Nature
      - Mountain Ranges, compare to MandelBrot
      - Sun Flowers
      - Show the golden Ratio
    - Fractals are all about recursion and iteration, so this gives me an excuse to look at them
      - Show MandelBrot
        * Python
          · Sympy Slow
          · Numpy Fast
        * Julia brings Both Benefits
          · Show Large MandelBrot
        * Show Julia Set
          · Show Julia Set Gif
11. Things I'd like to show
    - Simulate stripes and animal patterns
    - Show some math behind spirals in Nautilus Shells
    - Golden Rectangle
      - Throw in some recursion
      - Watch the spiral come out
      - Record the areas and show that they are Fibonacci
    - That the ratio of Fibonacci Converges to Phi
    - Any Connection to the Reimann Sphere
    - Lorrenz Attractor
      - How is this connected to the lorrenz attractor
    - What are the connections between discrete iteration and continuous systems such as the julia set and the lorrenz attractor
12. Things I'd like to Try (in order to see different ways to approach Problems)
    - Programming Languages and CAS
      - Julia
        * SymEngine
      - Maxima
      - Julia
    - Visualisation
      - Makie
      - Plotly
      - GNUPlot

13. Open Questions:

    - can we simulate animal patterns
    - can we simulate leaves
    - can we show that the gen func deriv 1.3.2
    - can we prove homogenous recursive relation
    - I want to look at the lorrenz attractor
    - when partiles are created by the the LHC, do they follow a fractal like pattern?
    - Create a Fractal Landscape, does this resemble things seen in nautre? [25, p. 464]
    - Can I write an algorighm to build a tree in the winter?
    - Can I develop my own type of persian recursion?
    - Show the relationship between the golden ratio and the logarithmic spiral.
        - and show that the fibonacci numbers pop out as area
            * Prove this
    - Is there any relationship between the Cantor Prisoner set and the Julia Sets?
    - Work with Matt to investigate Julia Sets for Quaternion [25, §13.9]
    - I'd like to write a program to solve sudoku problems as well

§ 3 Download RevealJS

So first do `M-x package-install ox-reveal` then do `M-x load-library` and then look for `ox-reveal`

```
(load "/home/ryan/.emacs.d/.local/straight/build/ox-reveal/ox-reveal.el")
```

Download Reveal.js and put it in the directory as `./reveal.js`, you can do that with something like this:

```
# cd /home/ryan/Dropbox/Studies/2020Spring/QuantProject/Current/Python-Quant/Outline/
wget https://github.com/hakimel/reveal.js/archive/master.tar.gz
tar -xzvf master.tar.gz && rm master.tar.gz
mv reveal.js-master reveal.js
```

Then just do `C-c e e R R` to export with RevealJS as opposed to PHP you won't need a fancy server, just open it in the browser.

§ 4 Heres a Gif

So this is a very big Gif that I'm using:

How did I make the Gif??

https://dl.dropboxusercontent.com/s/rbu25urfg8sbwfu/out.gif?dl=0

§ 5 Give a brief Sketch of the project

Of particular interest are the:

- gik
- fits image

```
code /home/ryan/Dropbox/Studies/QuantProject/Current/Python-Quant/ & disown
xdg-open /home/ryan/Dropbox/Studies/2020Spring/QuantProject/Current/Python-Qu
↪ ant/Problems/Chaos/mandelbrot-400mpx.fits
```

Here's what I gatthered from the week 3 slides

¶ 5.1 Topic / Context

We are interested in the theory of problem solving, but in particular the different approaches that can be taken to attacking a problem.

Essentially this boils down to looking at how a computer scientist and mathematician attack a problem, although originally I thought there was no difference, after seeing the odd way Roozbeh attacks problems I see there is a big difference.

¶ 5.2 Motivation

¶ 5.3 Basic Ideas

- Look at FOSS CAS Systems
  - Python (Sympy)
  - Julia
    - Sympy integration
    - symEngine
    - Reduce.jl
    - Symata.jl
- Maybe look at interactive sessions:
  - Like Jupyter
  - Hydrogen
  - TeXmacs
  - org-mode?

After getting an overview of SymPy let's look at problems that are interesting (chaos, morphogenesis and order from disarray etc.)

¶ 5.4 Where are the Mathematics

- Trying to look at the algorithms underlying functions in Python/Sympy and other Computer algebra tools such as Maxima, Maple, Mathematica, Sage, GAP and Xcas/Giac, Yacas, Symata.jl, Reduce.jl, SymEngine.jl
  - For Example Recursive Relations
- Look at solving some problems related to chaos theory maybe
  - Mandelbrot and Julia Sets
- Look at solving some problems related to Fourier Transforms maybe

AVOID DETAILS, JUST SKETCH THE PROJECT OUT.

¶ 5.5 Don't Forget we need a talk

5.5.1 Slides In Org Mode

- Without Beamer
- With Beamer

§ 6 Undecided

6.0.1 Determinant

Computational thinking can be useful in problems related to modelling, consider for example some matrix $n \times n$ matrix B_n described by (32) :

$$b_{ij} = \begin{cases} \frac{1}{2j-i^2}, & \text{if } i > j \\ \frac{i}{i-j} + \frac{1}{n^2-j-i}, & \text{if } j > i \\ 0 & \text{if } i = j \end{cases} \tag{32}$$

Is there a way to predict the determinant of such a matrix for large values?

From the perspective of linear algebra this is an immensely difficult problem and there isn't really a clear place to start.

From a numerical modelling perspective however, as will be shown, this a fairly trivial problem.

Create the Matrix

Using *Python* and `numpy`, a matrix can be generated as an `array` and by iterating through each element of the matrix values can be attributed like so:

```
import numpy as np
n = 2
mymat = np.empty([n, n])
for i in range(mymat.shape[0]):
    for j in range(mymat.shape[1]):
        print("(" + str(i) + "," + str(j) + ")")
```

```
(0,0)
(0,1)
(1,0)
(1,1)
```

and so to assign the values based on the condition in (32), an `if` test can be used:

```
   def BuildMat(n):
       mymat = np.empty([n, n])
       for i in range(n):
           for j in range(n):
               # Increment i and j by one because they count from zero
               i += 1; j += 1
               if (i > j):
                   v = 1/(2*j - i**2)
               elif (j > i):
                   v = 1/(i-j) + 1/(n**2 - j - i)
               else:
                   v = 0
               # Decrement i and j so the index lines up
               i -= 1; j -= 1
               mymat[j, i] = v
       return mymat

   BuildMat(3)
```

```
array([[ 0.        , -0.5       , -0.14285714],
       [-0.83333333,  0.        , -0.2       ],
       [-0.3       , -0.75      ,  0.        ]])
```

Find the Determinant

Python, being an object orientated language has methods belonging to objects of different types, in this case the `linalg` method has a `det` function that can be used to return the determinant of any given matrix like so:

```
-0.11928571428571424
```

```
    def detMat(n):
        ## Sympy
        # return Determinant(BuildMat(n)).doit()
        ## Numpy
        return np.linalg.det(BuildMat(n))
    detMat(3)
```

Listing 21: Building a Function to return the determinant of the matrix described in (32)

Find the Determinant of Various Values

To solve this problem, all that needs to be considered is the size of the n and the corresponding determinant, this could be expressed as a set as shown in (**??**):

$$\left\{\det\left(M(n)\right) \mid M \in \mathbb{Z}^{+} \leq 30\right\} \tag{33}$$

where:

- M is a function that transforms an integer to a matrix as per (32)

Although describing the results as a set (33) is a little odd, it is consistent with the idea of list and set comprehension in *Python* [1] and *Julia* [22] as shown in listing 22

Generate a list of values Using the function created in listing 21, a corresponding list of values can be generated:

```
    def detMat(n):
        return abs(np.linalg.det(BuildMat(n)))

    # We double all numbers using map()
    result = map(detMat, range(30))

    # print(list(result))
    [round(num, 3) for num in list(result)]
```

Listing 22: Generate a list using list-comprehension

```
[1.0,
 0.0,
 0.0,
 0.119,
 0.035,
 0.018,
 0.013,
 0.01,
 0.008,
```

```
 0.006,
 0.005,
 0.004,
 0.004,
 0.003,
 0.003,
 0.002,
 0.002,
 0.002,
 0.002,
 0.001,
 0.001,
 0.001,
 0.001,
 0.001,
 0.001,
 0.001,
 0.001,
 0.001,
 0.001,
 0.001]
```

Create a Data Frame

```
import pandas as pd

data = {'Matrix.Size': range(30),
        'Determinant.Value': list(map(detMat, range(30)))
}



df = pd.DataFrame(data, columns = ['Matrix.Size', 'Determinant.Value'

print(df)
```

```
   Matrix.Size  Determinant.Value
0             0           1.000000
1             1           0.000000
2             2           0.000000
3             3           0.119286
4             4           0.035258
5             5           0.018062
6             6           0.013023
7             7           0.009959
8             8           0.007822
9             9           0.006288
10           10           0.005158
11           11           0.004304
12           12           0.003645
```

13	13	0.003125
14	14	0.002708
15	15	0.002369
16	16	0.002090
17	17	0.001857
18	18	0.001661
19	19	0.001494
20	20	0.001351
21	21	0.001228
22	22	0.001121
23	23	0.001027
24	24	0.000945
25	25	0.000872
26	26	0.000807
27	27	0.000749
28	28	0.000697
29	29	0.000650

Plot the Data frame Observe that it is necessary to use copy, *Julia* and *Python* **unlike** *Mathematica* and ***R*** only create links between data, they do not create new objects, this can cause headaches when rounding data.

```
from plotnine import *
import copy

df_plot = copy.copy(df[3:])
df_plot['Determinant.Value'] =
↪  df_plot['Determinant.Value'].astype(float).round(3)
df_plot

(
    ggplot(df_plot, aes(x = 'Matrix.Size', y = 'Determinant.Value')) +
        geom_point() +
        theme_bw() +
        labs(x = "Matrix Size", y = "|Determinant Value|") +
        ggtitle('Magnitude of Determinant Given Matrix Size')

)
```

```
<ggplot: (8770001690691)>
```

In this case it appears that the determinant scales exponentially, we can attempt to model that linearly using `scikit`, this is significantly more complex than simply using ***R***. ^lrpy

```
    import numpy as np
    import matplotlib.pyplot as plt  # To visualize
    import pandas as pd  # To read data
    from sklearn.linear_model import LinearRegression

    df_slice = df[3:]

    X = df_slice.iloc[:, 0].values.reshape(-1, 1)  # values converts it into a
    ↪  numpy array
    Y = df_slice.iloc[:, 1].values.reshape(-1, 1)  # -1 means that calculate
    ↪  the dimension of rows, but have 1 column
    linear_regressor = LinearRegression()  # create object for the class
    linear_regressor.fit(X, Y)  # perform linear regression
    Y_pred = linear_regressor.predict(X)  # make predictions



    plt.scatter(X, Y)
    plt.plot(X, Y_pred, color='red')
    plt.show()
```

```
array([5.37864677])
```

Log Transform the Data

The `log` function is actually provided by `sympy`, to do this quicker in `numpy` use `np.log()`

```
    # # pyperclip.copy(df.columns[0])
    # #df['Determinant.Value'] =
    # #[ np.log(val) for val in df['Determinant.Value']]

    df_log = df

    df_log['Determinant.Value'] = [ np.log(val) for val in
    ↪  df['Determinant.Value'] ]
```

In order to only have well defined values, consider only after size 3

```
    df_plot = df_log[3:]
    df_plot
```

```
   Matrix.Size  Determinant.Value
3            3          -2.126234
4            4          -3.345075
```

```
5              5          -4.013934
6              6          -4.341001
7              7          -4.609294
8              8          -4.850835
9              9          -5.069048
10            10          -5.267129
11            11          -5.448099
12            12          -5.614501
13            13          -5.768414
14            14          -5.911529
15            15          -6.045230
16            16          -6.170659
17            17          -6.288765
18            18          -6.400347
19            19          -6.506082
20            20          -6.606547
21            21          -6.702237
22            22          -6.793585
23            23          -6.880964
24            24          -6.964704
25            25          -7.045094
26            26          -7.122390
27            27          -7.196822
28            28          -7.268592
29            29          -7.337885
```

A limitation of the *Python* `plotnine` library (compared to *Ggplot2* in ***R***) is that it isn't possible to round values in the `aesthetics` layer, a further limitation with `pandas` also exists when compared to ***R*** that makes rounding data very clusy to do.

In order to round data use the `numpy` library:

```
import pandas as pd
import numpy as np
df_plot['Determinant.Value'] =
    ↪ df_plot['Determinant.Value'].astype(float).round(3)
df_plot
```

```
    Matrix.Size  Determinant.Value
3             3             -2.126
4             4             -3.345
5             5             -4.014
6             6             -4.341
7             7             -4.609
8             8             -4.851
9             9             -5.069
10           10             -5.267
11           11             -5.448
12           12             -5.615
13           13             -5.768
```

14	14	-5.912
15	15	-6.045
16	16	-6.171
17	17	-6.289
18	18	-6.400
19	19	-6.506
20	20	-6.607
21	21	-6.702
22	22	-6.794
23	23	-6.881
24	24	-6.965
25	25	-7.045
26	26	-7.122
27	27	-7.197
28	28	-7.269
29	29	-7.338

```
from plotnine import *


(ggplot(df_plot[3:], aes(x = 'Matrix.Size', y = 'Determinant.Value')) +
   geom_point(fill= "Blue") +
   labs(x = "Matrix Size", y = "Determinant Value",
        title = "Plot of Determinant Values") +
   theme_bw() +
   stat_smooth(method = 'lm')
)
```

```
<ggplot: (8770002281897)>
```

```
from sklearn.linear_model import LinearRegression

df_slice = df_plot[3:]

X = df_slice.iloc[:, 0].values.reshape(-1, 1)  # values converts it into a numpy array
Y = df_slice.iloc[:, 1].values.reshape(-1, 1)  # -1 means that calculate the dimension of rows, but have 1 column
linear_regressor = LinearRegression()  # create object for the class
linear_regressor.fit(X, Y)  # perform linear regression
Y_pred = linear_regressor.predict(X)  # make predictions



plt.scatter(X, Y)
plt.plot(X, Y_pred, color='red')
plt.show()
```

```
m = linear_regressor.fit(X, Y).coef_[0][0]
b = linear_regressor.fit(X, Y).intercept_[0]

print("y = " + str(m.round(2)) + "* x" + str(b.round(2)))
```

```
y = -0.12* x-4.02
```

So the model is:

$$\text{abs}(\text{Det}(M)) = -4n - 0.12$$

where:

- n is the size of the square matrix

Largest Percentage Error

To find the largest percentage error for $n \in [30, 50]$ it will be necessary to calculate the determinants for the larger range, compressing all the previous steps and calculating the model based on the larger amount of data:

```
    import pandas as pd

    data = {'Matrix.Size': range(30, 50),
            'Determinant.Value': list(map(detMat, range(30, 50)))
    }
    df = pd.DataFrame(data, columns = ['Matrix.Size', 'Determinant.Value'])
    df['Determinant.Value'] = [ np.log(val) for val in df['Determinant.Value']]
    df
    from sklearn.linear_model import LinearRegression


    X = df.iloc[:, 0].values.reshape(-1, 1)  # values converts it into a numpy
    ↪ array
    Y = df.iloc[:, 1].values.reshape(-1, 1)  # -1 means that calculate the
    ↪ dimension of rows, but have 1 column
    linear_regressor = LinearRegression()  # create object for the class
    linear_regressor.fit(X, Y)  # perform linear regression
    Y_pred = linear_regressor.predict(X)  # make predictions

    m = linear_regressor.fit(X, Y).coef_[0][0]
    b = linear_regressor.fit(X, Y).intercept_[0]

    print("y = " + str(m.round(2)) + "* x" + str(b.round(2)))
```

```
y = -0.05* x-5.92
```

```
    Y_hat = linear_regressor.predict(X)
    res_per = (Y - Y_hat)/Y_hat
    res_per
```

```
array([[-5.41415364e-03],
       [-3.51384602e-03],
       [-1.90798428e-03],
       [-5.74487234e-04],
       [ 5.06726599e-04],
       [ 1.35396448e-03],
       [ 1.98395424e-03],
       [ 2.41201322e-03],
       [ 2.65219545e-03],
       [ 2.71742022e-03],
       [ 2.61958495e-03],
       [ 2.36966444e-03],
       [ 1.97779855e-03],
       [ 1.45336983e-03],
       [ 8.05072416e-04],
       [ 4.09734813e-05],
       [-8.31432011e-04],
```

```
       [-1.80517224e-03],
       [-2.87375452e-03],
       [-4.03112573e-03]])
```

```python
max_res = np.max(res_per)
max_ind = np.where(res_per == max_res)[0][0] + 30

print("The Maximum Percentage error is " + str(max_res.round(4) * 100) + "% which corresponds to a matrix of size " + str(max_ind))
```

```
The Maximum Percentage error is 0.27% which corresponds to a matrix of size 39
```

§ 7 What we're looking for

- Would a reader know what the project is about?
- Would a reader become interested in the upcoming report?
- Is it brief but well prepared?
- Are the major parts or phases sketched out

§ 8 Appendix

¶ 8.1 Persian Recursian Examples

¶ 8.2 Figures

¶ 8.3 Why Julia

The reason we resolved to make time to investigate *Julia* is because we see it as a very important tool for mathematics in the future, in particular because:

- It is a new modern language, designed primarily with mathematics in mind
  - First class support for UTF8 symbols
  - Full Support to call ***R*** and *Python*.
- Performance wise it is best in class and only rivalled by compiled languages such as *Fortran Rust* and ***C***
  - *Just in Time Compiling* allows for a very useable *REPL* making Julia significantly more appealing than compiled languages
  - The syntax of Julia is very similar to *Python* and ***R***
- The `DifferentialEquations.jl` library is one of the best performing libraries available.

```
    from __future__ import division
    from sympy import *
    x, y, z, t = symbols('x y z t')
    k, m, n = symbols('k m n', integer=True)
    f, g, h = symbols('f g h', cls=Function)
    init_printing()
    init_printing(use_latex='mathjax', latex_mode='equation')


    import pyperclip
    def lx(expr):
        pyperclip.copy(latex(expr))
        print(expr)

    import numpy as np
    import matplotlib as plt

    import time

    def timeit(k):
        start = time.time()
        k
        print(str(round(time.time() - start, 9)) + "seconds")
```

Listing 23: Preamble for *Python* Environment

```
%config InlineBackend.figure_format = 'svg'
main(5, 9, 1, cx)
```

Listing 24: Modify listing 11 to create 9 folds

```
%config InlineBackend.figure_format = 'svg'
def cx(l, r, t, b, m):
    new_col = (main.mat[t,l] + main.mat[t,r] +  main.mat[b,l] +
    ↪  main.mat[b,r]-7) % m
    return new_col.astype(int)
main(8, 8, 1, cx)
```

Listing 25: Modify the Function to use $f(w, x, y, z) = (w + x + y + z - 7) \mod 8$

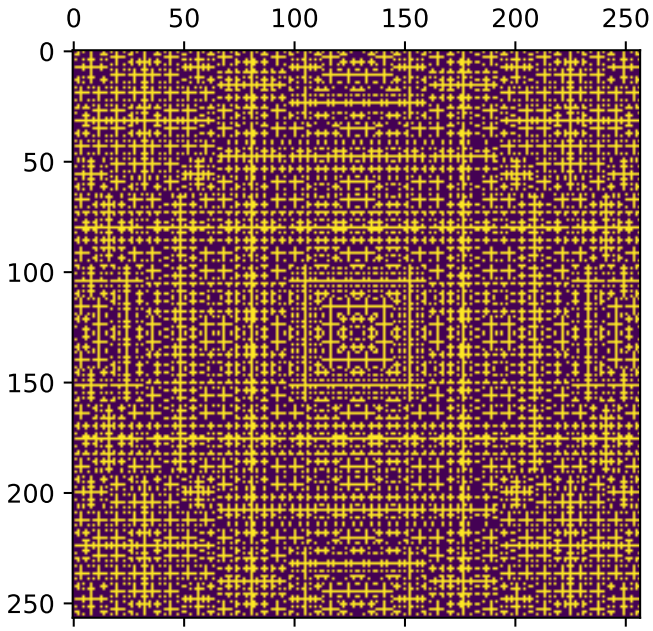


Figure 10: Output produced by listing 25 using $f(w, x, y, z) = (w + x + y + z - 7) \mod 8$

```
%config InlineBackend.figure_format = 'svg'
import numpy as np
def cx(l, r, t, b, m):
    new_col = (main.mat[t,l] + main.mat[t,r]*m +  main.mat[b,l]*(m) +
    ↪  main.mat[b,r]*(m))**1 % m + 1
    return new_col.astype(int)
main(8, 8, 1, cx)
```

Listing 26: Modify the function to use $f(w, x, y, z) = (w + 8x + 8y + 8z) \mod 8 + 1$

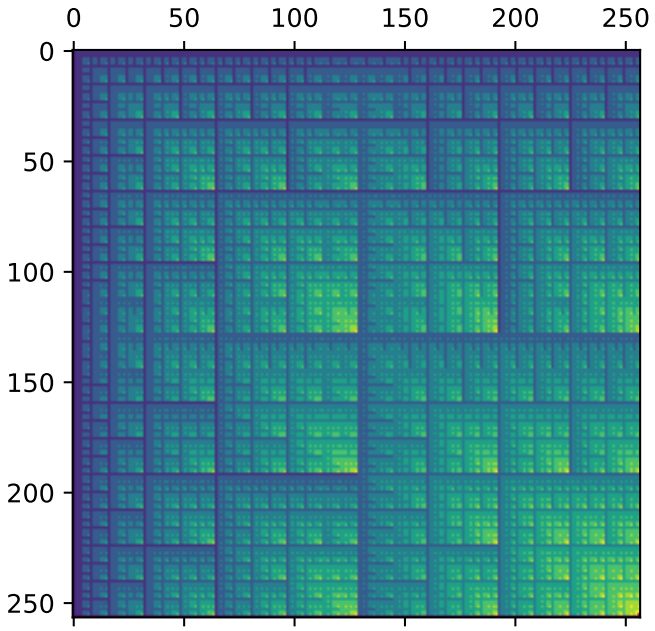


Figure 11: Output produced by listing 26 using $f(w, x, y, z) = (w + 8x + 8y + 8z) \mod 8 + 1$

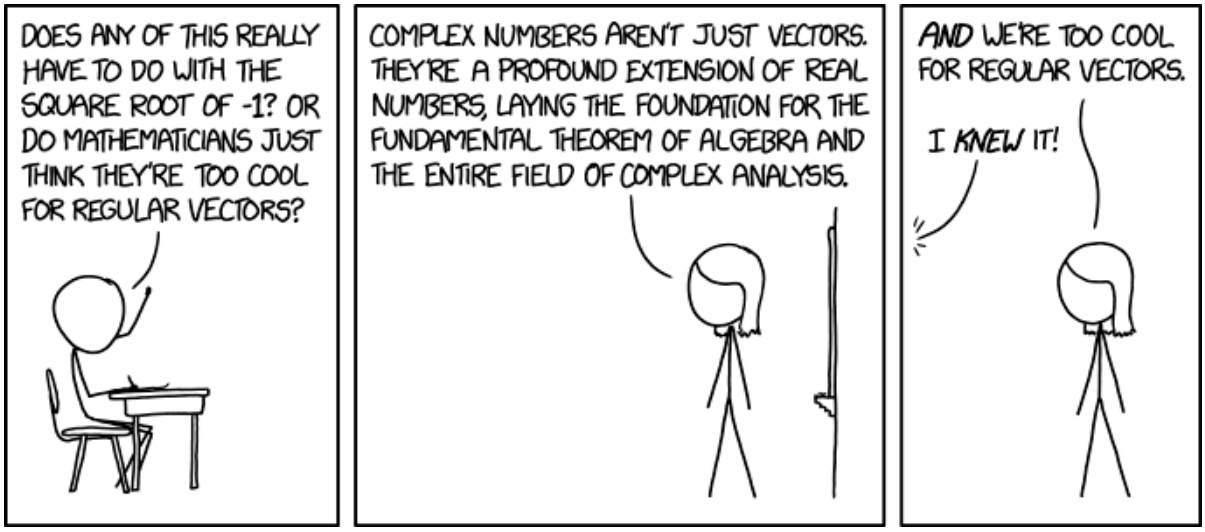


Figure 12: XKCD 2028: Complex Numbers

Other Pakcages

Other packages that are on our radar for want of investigation are listed below, in practice it is unlikely that time will permit us to investigate many packages or libraries

- Programming Languages and CAS
  - Julia
    - SymEngine.jl
    - Symata.jl
    - SymPy.jl
  - Maxima
    - Being the oldest there is probably a lot too learn
  - Julia
  - Reduce
  - Xcas/Gias
  - Python
    - Numpy
    - Sympy
- Visualisation
  - Makie
  - Plotly
  - GNUPlot

References

[1] *5. Data Structures Python 3.8.5 Documentation*. URL: https://docs.python.org/3/tutorial/datastructures.html (visited on 08/24/2020) (cit. on p. 42).

[2] A. C Benander, B. A Benander, and Janche Sang. "An Empirical Analysis of Debugging Performance Differences between Iterative and Recursive Constructs". In: *Journal of Systems and Software* 54.1 (Sept. 30, 2000), pp. 17–28. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(00)00023-6. URL: http://www.sciencedirect.com/science/article/pii/S0164121200000236 (visited on 08/24/2020) (cit. on p. 3).

[3] Benedetta Palazzo. *The Numbers of Nature: The Fibonacci Sequence*. June 27, 2016. URL: http://www.eniscuola.net/en/2016/06/27/the-numbers-of-nature-the-fibonacci-sequence/ (visited on 08/28/2020) (cit. on p. 4).

[4] Jeff Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (Jan. 2017), pp. 65–98. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/141000671. URL: https://epubs.siam.org/doi/10.1137/141000671 (visited on 08/28/2020) (cit. on pp. 2, 19).

[5] Corrado Böhm. "Reducing Recursion to Iteration by Algebraic Extension: Extended Abstract". In: *ESOP 86*. Ed. by Bernard Robinet and Reinhard Wilhelm. Red. by G. Goos et al. Vol. 213. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 111–118. ISBN: 978-3-540-16442-5 978-3-540-39782-3. DOI: 10.1007/3-540-16442-1_8. URL: http://link.springer.com/10.1007/3-540-16442-1_8 (visited on 08/24/2020) (cit. on p. 3).

[6] Corrado Böhm. "Reducing Recursion to Iteration by Means of Pairs and N-Tuples". In: *Foundations of Logic and Functional Programming*. Ed. by Mauro Boscarol, Luigia Carlucci Aiello, and Giorgio Levi. Red. by G. Goos et al. Vol. 306. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 58–66. ISBN: 978-3-540-19129-2 978-3-540-39126-5. DOI: 10.1007/3-540-19129-1_3. URL: http://link.springer.com/10.1007/3-540-19129-1_3 (visited on 08/24/2020) (cit. on p. 3).

[7] Simon Brass. *CC Search*. 2006, September 5. URL: https://search.creativecommons.org/ (visited on 08/28/2020) (cit. on p. 21).

[8] Anne M. Burns. ""Persian" Recursion". In: *Mathematics Magazine* 70.3 (1997), pp. 196–199. ISSN: 0025-570X. DOI: 10.2307/2691259. JSTOR: 2691259 (cit. on p. 19).

[9] *Emergence How Stupid Things Become Smart Together*. Nov. 16, 2017. URL: https://www.youtube.com/watch?v=16W7c0mb-rE (visited on 08/25/2020) (cit. on p. 22).

[10] Peter Farrell. *Math Adventures with Python: An Illustrated Guide to Exploring Math with Code*. San Francisco: No Starch Press, 2019. 276 pp. ISBN: 978-1-59327-867-0 (cit. on p. 24).

[11] *Functools Higher-Order Functions and Operations on Callable Objects Python 3.8.5 Documentation*. URL: https://docs.python.org/3/library/functools.html (visited on 08/25/2020) (cit. on p. 6).

[12] *Gnuplot / Fractal / Mandelbrot (E)*. URL: http://folk.uio.no/inf3330/scripting/doc/gnuplot/Kawano/fractal/mandelbrot-e.html (visited on 08/25/2020) (cit. on p. 33).

[13] Roozbeh Hazrat. *Mathematicaő: A Problem-Centered Approach*. 2nd ed. 2015. Springer Undergraduate Mathematics Series. Cham: Springer International Publishing : Imprint: Springer, 2015. 1 p. ISBN: 978-3-319-27585-7. DOI: 10.1007/978-3-319-27585-7 (cit. on pp. 2, 6).

[14] *Iteration vs. Recursion - CS 61A Wiki*. Dec. 19, 2016. URL: https://www.ocf.berkeley.edu/~shidi/cs61a/wiki/Iteration_vs._recursion (visited on 08/24/2020) (cit. on p. 3).

[15] *Julia Set*. In: *Wikipedia*. Page Version ID: 967264809. July 12, 2020. URL: https://en.wikipedia.org/w/index.php?title=Julia_set&oldid=967264809 (visited on 08/25/2020) (cit. on p. 25).

[16] Sophia Kivelson and Steven A. Kivelson. "Defining Emergence in Physics". In: *npj Quantum Materials* 1.1 (1 Nov. 25, 2016), pp. 1–2. ISSN: 2397-4648. DOI: 10.1038/npjquantmats.2016.24. URL: https://www.nature.com/articles/npjquantmats201624 (visited on 08/25/2020) (cit. on p. 22).

[17] Robert Lamb. *How Are Fibonacci Numbers Expressed in Nature?* June 24, 2008. URL: https://science.howstuffworks.com/math-concepts/fibonacci-nature.htm (visited on 08/28/2020) (cit. on p. 4).

[18] Eric Lehman, Tom Leighton, and Albert Meyer. *Readings | Mathematics for Computer Science | Electrical Engineering and Computer Science | MIT OpenCourseWare*. Sept. 8, 2010. URL: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-science-fall-2010/readings/ (visited on 08/10/2020) (cit. on p. 11).

[19] Oscar Levin. *Solving Recurrence Relations*. Jan. 29, 2018. URL: http://discrete.openmathbooks.org/dmoi2/sec_recurrence.html (visited on 08/11/2020) (cit. on p. 13).

[20] *Mandelbrot Set - Rosetta Code*. URL: https://rosettacode.org/wiki/Mandelbrot_set#Python (visited on 08/25/2020) (cit. on p. 35).

[21] Nikoletta Minarova. "The Fibonacci Sequence: Natures Little Secret". In: *CRIS - Bulletin of the Centre for Research and Interdisciplinary Study* 2014.1 (2014), pp. 7–17. ISSN: 1805-5117 (cit. on p. 4).

[22] *Multi-Dimensional Arrays ů The Julia Language*. URL: https://docs.julialang.org/en/v1/manual/arrays/#man-comprehensions-1 (visited on 08/24/2020) (cit. on p. 42).

[23] *Nature, The Golden Ratio and Fibonacci Numbers*. 2018. URL: https://www.mathsisfun.com/numbers/nature-golden-ratio-fibonacci.html (visited on 08/28/2020) (cit. on pp. 4, 19).

[24] Olympia Nicodemi, Melissa A. Sutherland, and Gary W. Towsley. *An Introduction to Abstract Algebra with Notes to the Future Teacher*. OCLC: 253915717. Upper Saddle River, NJ: Pearson Prentice Hall, 2007. 436 pp. ISBN: 978-0-13-101963-8 (cit. on p. 13).

[25] Heinz-Otto Peitgen, H. Jürgens, and Dietmar Saupe. *Chaos and Fractals: New Frontiers of Science*. 2nd ed. New York: Springer, 2004. 864 pp. ISBN: 978-0-387-20229-7 (cit. on pp. 22, 38).

[26] Ron Knott. *The Fibonacci Numbers and Golden Section in Nature - 1*. Sept. 25, 2016. URL: http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html (visited on 08/28/2020) (cit. on pp. 4, 36).

[27] Shelly Allen. *Fibonacci in Nature*. URL: https://fibonacci.com/nature-golden-ratio/ (visited on 08/28/2020) (cit. on p. 4).

[28] A.P. Sinha and I. Vessey. "Cognitive Fit: An Empirical Study of Recursion and Iteration". In: *IEEE Transactions on Software Engineering* 18.5 (May 1992), pp. 368–379. ISSN: 00985589. DOI: 10.1109/32.135770. URL: http://ieeexplore.ieee.org/document/135770/ (visited on 08/24/2020) (cit. on p. 3).

[29] S Smolarski. *Math 60 – Notes A3: Recursion vs. Iteration*. Feb. 9, 2000. URL: http://math.scu.edu/~dsmolars/ma60/notesa3.html (visited on 08/24/2020) (cit. on pp. 3, 4).

[30] Alan Turing. "The Chemical Basis of Morphogenesis". In: *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences* 237.641 (Aug. 14, 1952), pp. 37–72. ISSN: 2054-0280. DOI: 10.1098/rstb.1952.0012. URL: https://royalsocietypublishing.org/doi/10.1098/rstb.1952.0012 (visited on 08/25/2020) (cit. on p. 22).

[31] Dennis G Zill and Michael R Cullen. *Differential Equations*. 7th ed. Brooks/Cole, 2009 (cit. on pp. 12, 13).

[32] Dennis G. Zill and Michael R. Cullen. "8.4 Matrix Exponential". In: *Differential Equations with Boundary-Value Problems*. 7th ed. Belmont, CA: Brooks/Cole, Cengage Learning, 2009. ISBN: 978-0-495-10836-8 (cit. on pp. 13, 15).