../Resources/references.bib

Thinking about Problems

Ryan Greenup & James Guerra

August 27, 2020

Contents

1 Introduction

During preperation for this outline, an article published by the *Mathematical Association of America* caught my attention, in which mathematics is referred to as the *Science of Patterns* [**?**], this I feel, frames very well the essence of the research we are looking at in this project. Mathematics, generally, is primarily concerned with problem solving (that isn't, however, to say that the problems need to have any application[1]), and it's fairly obvious that different strategies work better for different problems. That's what we want to investigate, Different to attack a problem, different ways of thinking, different ways of framing questions.

The central focus of this investigation will be with computer algebra and the various libraries and packages that exist in the free open source [2] space to solve and visualise numeric and symbolic problems, these include:

- Programming Languages and CAS
  - Julia
    - SymEngine
  - Maxima
    - Being the oldest there is probably a lot too learn
  - Julia
  - Reduce
  - Xcas/Gias
  - Python
    - Numpy

[1]Although Hardy made a good defence of pure math in his 1940s Apology [**?**], it isn't rare at all for pure math to be found applications, for example much number theory was probably seen as fairly pure before RSA Encryption [**?**].

[2]Although proprietary software such as Magma, Mathematica and Maple is very good, the restrictive licence makes them undesirable for study because there is no means by which to inspect the problem solving tecniques implemented, build on top of the work and moreover the lock-in nature of the software makes it a risky investment with respect to time.

        * Sympy

- Visualisation
  - Makie
  - Plotly
  - GNUPlot

Many problems that look complex upon initial inspection can be solved trivially by using computer algebra packages and our interest is in the different approaches that can be taken to *attack* each problem. Of course however this leads to the question:

> Can all mathematical problems be solved by some application of some set of rules?

This is not really a question that we can answer, however, determinism with respect to systems is appears to make a very good area of investigation with respect to finding ways to deal with problems.

This is not an easy question to answer, however, while investigating this problem

Determinism

Are problems deterministic? can the be broken down into a step by step way? For example if we *discover all the rules* can we then simply solve all the problems?

chaos to look at patterns generally to get a deeper understanding of patterns and problems, loops and recursion generally.

To investigate different ways of thinking about math problems our investigation

laplaces demon

but then heisenberg,

but then chaos and meh.

1.1 Preliminary Problems

1.1.1 Iteration and Recursion

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

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

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

let's modify this for the sake of discussion:

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

The function h can be expressed by the series:

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

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.

```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 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, [**?**, **?**] see also [**?**, **?**]

there is, however, evidence to suggest that recursive functions are easier for people to understand [**?**] . 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 [**?**].

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 [**?**] 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 1.1.1).

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

1.1.2 Fibonacci Sequence

Computational Approach The *Fibonacci* Numbers are given by:

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

```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 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 [**?**], 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.

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

## 8.3923e-05seconds
```

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

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 .

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:

```python
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

```julia
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

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

This would implify that $f(x) \propto e^{mx}, \quad \exists m \in \mathbb{Z}$ because $\frac{\mathrm{d}(e^x)}{\mathrm{d}x} = e^x$, and so by using a power series it's quite feasable to move between discrete and continuous problems:

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

Example Consider using the following generating function, (the derivative of the generating function as in (7) and (8) is provided in section 1.1.2)

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

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

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

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

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

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:

$$\begin{aligned} a_n &= \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \\ &= \frac{\varphi^n - \psi^n}{\sqrt{5}} \\ &= \frac{\varphi^n - \psi^n}{\varphi - \psi} \end{aligned} \tag{9}$$

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: [**?**]

$$
\begin{aligned}
f\left(x\right) &= \sum_{n=0}^{\infty}\left[a_n\frac{x^n}{n!}\right] && (10)\\
f'\left(x\right)) &= \frac{\mathrm{d}}{\mathrm{d}x}\left(\sum_{n=0}^{\infty}\left[a_n\frac{x^n}{n!}\right]\right)\\
&= \frac{\mathrm{d}}{\mathrm{d}x}\left(a_0\frac{x^0}{0!}+a_1\frac{x^1}{1!}+a_2\frac{x^2}{2!}+a_3\frac{x^3}{3!}+\dots\frac{x^k}{k!}\right)\\
&= \sum_{n=0}^{\infty}\left[\frac{\mathrm{d}}{\mathrm{d}x}\left(a_n\frac{x^n}{n!}\right)\right]\\
&= \sum_{n=0}^{\infty}\left[\frac{a_n}{(n-1)!}x^{n-1}\right]\\
\implies f'(x) &= \sum_{n=0}^{\infty}\left[\frac{x^n}{n!}a_{n+1}\right] && (11)
\end{aligned}
$$

If $f\left(x\right)=\sum_{n=0}^{\infty}\left[a_n\frac{x^n}{n!}\right]$ can it be shown by induction that $\frac{\mathrm{d}^k}{\mathrm{d}x^k}\left(f\left(x\right)\right)=f^k\left(x\right)\sum_{n=0}^{\infty}\left[x^n\frac{a_{n+k}}{n!}\right]$

Homogeneous Proof An equation of the form:

$$\sum_{n=0}^{\infty}\left[c_i\cdot f^{(n)}(x)\right]=0 \tag{12}$$

is said to be a homogenous linear ODE: [**?**, 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, [**?**, 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 (**??**)). 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 [**?**]
- $\exists i, j \in \mathbb{Z}^+ \cap [0, k]$
  - These is often referred to as repeated roots [**?**, **?**] with a multiplicity corresponding to the number of repetitions of that root [**?**, §3.2]

1. Unique Roots of Characteristic Equation
   (a) Example An example of a recurrence relation with all unique roots is the fibonacci sequence, as described in section 1.1.2.
   (b) Proof Consider the linear recurrence relation (**??**):

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

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

$$\sum_{i=0}^{k} [c_i \cdot a_n] = 0$$

By Multiplying through and summing:

$$\begin{aligned}\implies \sum_{i=0}^{k}\left[\sum_{n=0}^{\infty}\left[c_i a_n \frac{x^n}{n!}\right]\right] &= 0\\ \sum_{i=0}^{k}\left[c_i \sum_{n=0}^{\infty}\left[a_n \frac{x^n}{n!}\right]\right] &= 0\end{aligned} \tag{13}$$

Recall from (6) the generating function $f(x)$:

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

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 [**?**, 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):

$$\begin{aligned}\sum_{i=0}^{k}[k_i e^{m_i x}] &= \sum_{i=0}^{k}\left[k_i \sum_{n=0}^{\infty}\frac{(m_i x)^n}{n!}\right]\\ &= \sum_{i=0}^{k}\sum_{n=0}^{\infty} k_i m_i^n \frac{x^n}{n!}\\ &= \sum_{n=0}^{\infty}\sum_{i=0}^{k} k_i m_i^n \frac{x^n}{n!}\\ &= \sum_{n=0}^{\infty}\left[\frac{x^n}{n!}\sum_{i=0}^{k}[k_i m_i^n]\right], \quad \exists k_i \in \mathbb{C},\ \ \forall i \in \mathbb{Z}^+ \cap [1, k]\end{aligned} \tag{15}$$

Recall the definition of the generating function from 14, by relating this to (15):

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

$\square$

This can be verified by the fibonacci sequence as shown in section 1.1.2, 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:

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

2. Repeated Roots of Characteristic Equation

   (a) Example Consider the following recurrence relation:

$$a_n - 10a_{n+1} + 25a_{n+2} = 0 \quad (16)$$

$$\implies \sum_{n=0}^{\infty}\left[a_n\frac{x^n}{n!}\right] - 10\sum_{n=0}^{\infty}\left[\frac{x^n}{n!}+\right] + 25\sum_{n=0}^{\infty}\left[a_{n+2}\frac{x^n}{n!}\right] = 0$$

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:

$$\begin{aligned} m^2 - 10m + 25 &= 0 \\ (m-5)^2 &= 0 \\ m &= 5 \end{aligned} \tag{17}$$

Herein lies a complexity, in order to solve this, the solution produced from (17) can be used with the *Reduction of Order* technique to produce a solution that will be of the form [**?**, §4.3].

$$f(x) = c_1 e^{5x} + c_2 x e^{5x} \tag{18}$$

(18) 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):

$$x^k e^x = \sum_{n=0}^{\infty} \left[\frac{x^n}{(n-k)!} \right], \quad \exists k \in \mathbb{Z}^+ \tag{19}$$

by applying identity (19) to equation (18)

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

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

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

$\square$

(b) Generalised Example

(c) 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)):

$$
\sum_{i=0}^{k} \left[c_i f^{(k)}(x) \right] \tag{20}
$$

Assume that (20) corresponds to a charecteristic polynomial with only 1 root of multiplicity k, the solution would hence be of the form:

$$
\begin{aligned}
\sum_{i=0}^{k} \left[c_i m^i \right] = 0 \wedge m = B, \quad \exists! B \in \mathbb{C} \\
\implies f(x) = \sum_{i=0}^{k} \left[x^i A_i e^{mx} \right], \quad \exists A \in \mathbb{C}^+, \ \forall i \in [1, k] \cap \mathbb{N} \quad (21) \\
(22)
\end{aligned}
$$

Recall the following power series identity (proved in section xxx):

$$
x^k e^x = \sum_{n=0}^{\infty} \left[\frac{x^n}{(n-k)!} \right]
$$

By applying this to (21) :

$$\begin{aligned} f(x) &= \sum_{i=0}^{k}\left[A_i\sum_{n=0}^{\infty}\left[\frac{(xm)^n}{(n-i)!}\right]\right] \\ &= \sum_{n=0}^{\infty}\left[\sum_{i=0}^{k}\left[\frac{x^n}{n!}\frac{n!}{(n-i)}A_i m^n\right]\right] && (23) \\ &= \sum_{n=0}^{\infty}\left[\frac{x^n}{n!}\sum_{i=0}^{k}\left[\frac{n!}{(n-i)}A_i m^n\right]\right] && (24) \end{aligned}$$

Recall the generating function that was used to get 20:

$$\begin{aligned} f(x) &= \sum_{n=0}^{\infty}\left[a_n\frac{x^n}{n!}\right] \\ \implies a_n &= \sum_{i=0}^{k}\left[A_i\frac{n!}{(n-i)!}m^n\right] \\ &= \sum_{i=0}^{k}\left[m^n A_i\prod_{0}^{k}\left[n-(i-1)\right]\right] && (25) \end{aligned}$$

$\because \quad i \leq k$

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

$\square$

3. General Proof In sections 1 and 1 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 here it will be shown that these two can be combined and that the solution will still hold.

Consider a Recursive relation with constant coefficients:

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

This can be expressed in terms of the exponential generating function:

$$\sum_{n=0}^{\infty} [c_i \cdot a_n] = 0 \implies \sum_{n=0}^{\infty} \left[\sum_{n=0}^{\infty} [c_i \cdot a_n] \right] = 0$$

- 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

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*:

$$\begin{aligned}
F_n &= \frac{\varphi^n - \psi^n}{\varphi - \psi} = \frac{\varphi^n - \psi^n}{\sqrt{5}} \\
\iff \frac{F_{n+1}}{F_n} &= \frac{\varphi^{n+1} - \psi^{n+1}}{\varphi^n - \psi^n} \\
\iff \lim_{n\to\infty} \left[\frac{F_{n+1}}{F_n} \right] &= \lim_{n\to\infty} \left[\frac{\varphi^{n+1} - \psi^{n+1}}{\varphi^n - \psi^n} \right] \\
&= \frac{\varphi^{n+1} - \lim_{n\to\infty} \left[\psi^{n+1} \right]}{\varphi^n - \lim_{n\to\infty} \left[\psi^n \right]} \\
\text{because } \mid \psi \mid < 0 \ n \to \infty &\implies \psi^n \to 0\text{:} \\
&= \frac{\varphi^{n+1} - 0}{\varphi^n - 0} \\
&= \varphi
\end{aligned}$$

We'll come back to this later on when looking at spirals and fractals.
This can also be shown by using analysis, let \$L=lim$_{\mathrm{n} \to \infty} \left[\frac{\{}{F}\mathrm{n+1}\}\{F_{\mathrm{n}}\} \right]$, *then* :

$$L\& = \frac{\{}{F}^{\mathrm{n+\,1}}\}\{f_{\mathrm{n}}\} \tag{26}$$

$$\& = \frac{\{}{F}^{\mathrm{n}} + F_{\mathrm{n\text{-}\,1}}\}\{F_{\mathrm{n}}\} \tag{27}$$

$$\& = 1 + \frac{\{}{F}^{\mathrm{n\,\text{-}\,1}}\}\{F_{\mathrm{n}}\} \tag{28}$$

$$highlightcolor \tag{29}$$

Se This Tutorial

1.1.3 Persian Recursion

Although some recursive problems are a good fit for mathematical thinking such as the *Fibonacci Sequence* discussed in section 1.1.2 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 [**?**] that produces fantastic patterns upon feedback and iteration

The procedure 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 10 and in figure 1 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

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

This is demonstrated in listing 10 with python and produces the output shown in figures 1, 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 [**?**, **?**], 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 [**?**, p. 16], this is a centreal theme of Alan Turing's *The Chemical Basis For Morphogenesis* [**?**] which we hope to look in the course of this research.

[width=6cm]./persian-recursion-0

Figure 1: Output produced by listing 10 with 6 folds

1.1.4 Julia

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

```python
%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 10: Implementation of the persian recursion scheme in *Python*

- 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.

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[5] and define the operations as described in listing 11.[6]

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 [7], this is implemented in listing 12 and the corresponding output shown in .

with respect to listing 12:

- 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

This is precisely what we expected, but this is where things get interesting, consider now the result if we apply this same procedure to $f_1 : z \to z^2 - 1$

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

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

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

```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 11: Defining Complex Operations with vectors

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 2 and 3.

[width=9cm]./julia-1

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

[width=9cm]./julia-rab

Figure 3: Circle of Convergence for $f_{\frac{1}{4}+\frac{i}{2}} : z \to z^2 + \frac{1}{4} + \frac{i}{2}$

Now this is particularly interesting, to investigate this further consider the more general function $f_{0.8e^{\pi i \tau}} : z \to z^2 + 0.8e^{\pi i \tau}, \enspace \tau \in \mathbb{R}$, many fractals can be generated using this set by varying the value of τ[8].

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 13.

[8]This approach was inspired by an animation on the *Julia Set* Wikipedia article [**?**]

```python
%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 12: Circle of Convergence of z under recursion

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 14 where 1200 pictures at a 2.25 MP resolution were produced. [9]

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 15, the collage is shown in figure 4 and a corresponding animation is available online[10]].

Figure 4: Various fracals corresponding to $f_{0.8e^{\pi i \tau}}$

1.1.5 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.

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.

This strategy is implemented in listing

[width=.9]./mandelbrot-py

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.[11]

```
function mandelbrot(z, num, my_func)
    count = 1
    # Define z1 as z
    z1 = z
    # Iterate num times
    while count  num
```

[9]On my system this took about 30 minutes.
[10]https://dl.dropboxusercontent.com/s/rbu25urfg8sbwfu/out.gif?dl=0
[11]https://www.dropbox.com/s/jd5qf1pi2h68f2c/mandelbrot-400mpx.fits?dl=0

```julia
# * 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 13: Produce a series of fractals using julia

```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
        name = string("/tmp/gifs/j", lpad(j, 5, "0"), ".png")
        GR.savefig(name)
    end
end

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

Listing 14: Generate and save the images with GR

```bash
# 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')  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 15: Using `bash`, `ffmpeg` and *ImageMagick* to combine the images and produce an animation.

```python
%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 16: All values of c that lead to a closed *Julia-set*

```
        # 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)
```

1.1.6 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 17.[12]

limit of recursion is 250

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

set xrange [-2:2]
set yrange [-2:2]
set logscale z
set isosample 240
set hidden3d
set contour
splot mandel(x,y,{0,0},0) notitle
```

Listing 17: Visualising the Mandelbrot set as a 3D surface Plot

[width=.9]one

reference for image

[width=.9]two

GNU Plot can also make excellent 2d renditions of fractals, an example of how to perform this can be found on *Rosetta Code* [?] and is demonstrated in listing 19.

1.1.7 Determinant??

2 Outline

1. Intro Prob

[12]See [?] 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 || 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 mandel(a,b,complex(x,y),0) notitle
```

Listing 18: Use GNUPlot to produce plot of julia set

```
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 +
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 19: Flat Mandelbrot set built using rosetta code.

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 1.1.2
- 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? [**?**, 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 [**?**, §13.9]

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-Quant/Prob
```

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 (30) :

$$b_{ij} = \begin{cases} \frac{1}{2j-i^2}, & \text{if } i > j \\ \frac{i}{i-j} + \frac{1}{n^2-j-i}, & \text{if } j > i \\ 0 & \text{if } i = j \end{cases} \tag{30}$$

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 (30), 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:

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

Listing 20: Building a Function to return the determinant of the matrix described in (30)

```
-0.11928571428571424
```

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 (**??**):

$$\left\{\det\left(M(n)\right) \mid M \in \mathbb{Z}^{+} \leq 30\right\} \tag{31}$$

where:

- M is a function that transforms an integer to a matrix as per (30)

Although describing the results as a set (31) is a little odd, it is consistent with the idea of list and set comprehension in *Python* [**?**] and *Julia* [**?**] as shown in listing 21

Generate a list of values Using the function created in listing 20, a corresponding list of values can be generated:

```python
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 21: 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 ar
Y = df_slice.iloc[:, 1].values.reshape(-1, 1)  # -1 means that calculate the dimens
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:

```python
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)>
```

```python
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 ar
Y = df_slice.iloc[:, 1].values.reshape(-1, 1)  # -1 means that calculate the dimens
linear_regressor = LinearRegression()  # create object for the class
linear_regressor.fit(X, Y)  # perform linear regression
Y_pred = linear_regressor.predict(X)  # make predictions
```

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

```python
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
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]])
```

```
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 c
```

```
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

[width=9cm]./persian-recursion-new-func

Figure 5: Output produced by listing 24 using $f(w,x,y,z) = (w + x + y + z - 7) \mod 8$

[width=9cm]./persian-recursion-new-func2

Figure 6: Output produced by listing 25 using $f(w,x,y,z) = (w + 8x + 8y + 8z) \mod 8 + 1$

```python
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 22: Preamble for *Python* Environment

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

Listing 23: Modify listing 10 to create 9 folds

```python
%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 24: Modify the Function to use $f(w, x, y, z) = (w + x + y + z - 7)$ 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]*(
    return new_col.astype(int)
main(8, 8, 1, cx)
```

Listing 25: Modify the function to use $f(w, x, y, z) = (w + 8x + 8y + 8z) \mod 8 + 1$

8.2 Figures

Figure 7: XKCD 2028: Complex Numbers