Contents

1 How to read this document

TODO mention browser, slide show, clickable links, listing blocks appearing in odd places

2 Introduction

Cornifer is an easy-to-use data manager for computational and experimental mathematics, written in Python.

TODO: bridge

Roughly speaking, computers have two different kinds of memory, disk memory and RAM. Disk memory is used for long-term, persistent storage. Documents, music, pictures, and program files are stored in a computer's disk memory. Even after the computer is turned off, the disk memory persists.

RAM, short for *random access memory*, is used for temporary storage. A program reads and writes data from and to RAM whenever the program does not need long-term, persistent access to that data; otherwise, it writes the data to disk. The size of RAM is usually between one and two orders of magnitude smaller than the size of the disk memory: On most personal computers, RAM is between 2 and 16 GB, whereas disk memory may be between 100 and 1000 GB. The tradeoff is that read and write operations to RAM are significantly faster than read and write operations to disk.

If a program is frequently reading and writing a large amount of data from and to RAM, it may overload the RAM. Therefore, programs sometimes *free* their data, essentially forgetting

```python
from pickle import dump, load

lst = list(range(1000))

# `file` is the location where we will put the list `lst` on the disk.
file = "C:/Users/Michael/my_list.pickle"

# These two lines save the list `lst` to `file`.
with open(file, "wb") as fh:
    dump(lst, fh)

# Now you can close the Python program or turn off your computer. The data
# at `file` will persist and you can load it back in.

# These two lines load the data.
with open(file, "rb") as fh:
    lst = load(fh)
```

Listing 1: Simple `pickle` example.

```python
from cornifer import Numpy_Register, Block, Apri_Info

my_saves_dir = "C:/Users/Michael/cornifer_database"
message = "My first Register!"
reg = Numpy_Register(my_saves_dir, message)

data = [x**2 for x in range(1000)]

apri = Apri_Info(name = "consecutive perfect squares")

blk = Block(data, apri)

with reg.open() as reg:
    reg.add_disk_block(blk)
```

Listing 2: Saving a `Block`. A walkthrough of this code can be found in Section 3.1.

it, which prevents overloading the RAM. When a program terminates, *all* of its data in RAM is automatically freed, so that other programs may use it.

In a Python environment, all data is written to RAM unless specified otherwise. If you say, for example, `lst = list(range(1000))`, then the list `lst` exists in RAM. You can manually free this memory by writing either `lst.clear()` or `del lst`.

Python has a variety of different methods for saving data to disk memory. The easiest to use is the `pickle` module, which saves Python objects straight to disk. An example is given in Listing 2.

TODO say a few words about sequences, blocks, what kinds of sequences Cornifer works well for, why and how to dump blocks,

3 Basic usage

A `Register` is an interface used to save and load sequences to and from the disk. TODO bridge

3.1 Saving data

In Listing 2, we have given an example of code that creates a new `Register` and saves a segment of a sequence to the disk.

```
1 from cornifer import Numpy_Register, Block, Apri_Info
```

This line just imports code from Cornifer.

```
3 my_saves_dir = "C:/Users/Michael/cornifer_database"
4 message = "Example Register from the docs."
5 reg = Numpy_Register(my_saves_dir, message)
```

In lines 3 to 5, we create a `Register` and name it `reg`. The directory `my_saves_dir` is where `reg` will put your save data. That directory must be created before you initialize `reg`, otherwise Cornifer will `raise FileNotFoundError`. The string `message` is a brief summary the the `Register`'s save data.

You do not need to create a new save directory for every new `Register`. In fact, it is good practice to put all of your `Register`s into a single directory. (To create a backup of your save directory, just copy it to a different location.)

`Numpy_Register` refers to the format of the `Register`'s save data. `Numpy_Register` uses the functions `numpy.save` and `numpy.load` to save and load Numpy arrays. However, you do not need to call these two functions on your own, because Cornifer does that for you! Cornifer comes with three different preprogrammed `Register`s, namely `Numpy_Register`, `HDF5_Register`, and `Pickle_Register`. If you would like to create your own `Register` derived type, please see Section 7.1.

```
7 seg = [x**2 for x in range(1000)]
```

Line 7 is a standard Python idiom. `seg` is a `list` of the perfect squares from 0 to 999^2, inclusive.

```
9 apri = Apri_Info(name = "consecutive perfect squares")
```

The name `Apri_Info` is an abbreviation of *a priori information*. We use `Apri_Info` to encode a brief description of the sequence we are saving to disk. You (the user) have a fair amount of freedom in how you use `Apri_Info` to describe your sequences, but you must follow a few rules and guidelines (see Section 4).

```
11 blk = Block(seg, apri)
```

A `Block` is a finite, contiguous segment of a sequence. A `Block` wraps three pieces of information:

1. The segment of the sequence itself. The segment is the first argument of `Block(seg, apri)`.
2. The `Apri_Info` that describes the sequence. This is the second argument of `Block(seg, apri)`.
3. The first index of the segment. We could specify that the first index is, say, 5 by writing `blk = Block(seg, apri, 5)`. The default value is 0, so when we write `blk = Block(seg, apri)`, the start index defaults to 0.

```
13 with reg.open() as reg:
```

The data on the disk is behind a closed door. You must open the door before Cornifer will let you save or load any data behind the door. Line 13 opens the door. Cornifer automatically closes the door once your Python code leaves the `with` block.

```
14     reg.add_disk_block(blk)
```

Line 14 finally saves the `Block` named `blk` to the disk. In Sections 3.2 and 3.3, we will see how we can load the data back in from the disk.

3.2 Searching for data

In Section 3.1, we saved the `list` of integers from 0 to 999, inclusive, to disk. That `list` persists on your disk; you can terminate the Python interpreter or even turn off your computer and it will still be accessible.

Before we can load the data back in, we need to find the `Register` we used to save it. This is pretty easy using Cornifer's `search` and `load` functions.

You should use `search` in a Python interactive environment, also called a Python console. We first import the necessary functions.

```
> from cornifer import search, load
```

The function `search` can take a wide variety of arguments, it is very flexible (TODO: ref). In Listing 2, we described the `Block` we saved by

```
Apri_Info(name = "consecutive perfect squares")
```

We can search for the `Register` we made in Listing 2 in the following way.

```
> my_saves_dir = "C:/Users/Michael/cornifer_database"
> search(saves_directory = my_saves_dir, name = "consecutive perfect squares")
```

The Python console will print the following.

```
"""
REGISTERS:
XXXXXX "Example Register from the docs."
    Apri_Info(name = "consecutive perfect squares")
        [0, 1000)
"""
```

This is a description of the `Register` we created earlier in Listing 2. The string `"XXXXXX"` is called the `identifier` of the `Register`. The `identifier` for your `Register` will almost surely be different than `"XXXXXX"`. I cannot say what `identifier` you have, because `identifiers` are created randomly.

You can use the `identifier` to load in the `Register`. The `identifier` is permanent, so if you already know what it is, then you can skip using the `search` function.

```
> reg = load("XXXXXX", my_saves_dir)
```

Now you have the `Register`! In Section 3.3, we will see how to load data from it.

3.3 Loading data

Using `Register`, there are several methods for loading data from the disk. Which method you use depends entirely what you are doing with the data. Sometimes it will be better to load in a single index at a time; sometimes you want to load `Block`s in chunks. Each method will be described in the subsequent subsections. For each method, `reg` is the `Register` that we created in Section 3.1 and loaded in Section 3.2. Likewise, for each method, denote

```
apri = Apri_Info(name = "consecutive perfect squares")
```

3.3.1 Single indices

Say you only want the element with index 343. You can get this element via

```
reg[apri, 343]
```

The line above will return $343^2 = 117649$.

3.3.2 Slices of indices

Say you want all elements with indices between 10 and 20, inclusive. You can get these elements via

```
list( reg[apri, 10:21] )
```

The semantics are exactly the same for `Register` as they are for Python `list`. So, for example, if you wanted every other element starting with index 50 and ending at the last index that you have saved, then

```
list( reg[apri, 50:2:-1] )
```

will get you what you want. Namely, the line above returns the list

```
[50**2, 52**2, 54**2, ..., 998**2]
```

To iterate over all elements of the sequence, you can do

```
for x in reg[apri, :]:
    # do some stuff with `x`
```

The iterator `reg[apri, :]` will keep returning elements as long as it has not reached the end of the sequence saved to disk. So for example, say you write something like this:

```
1 for x in reg[apri, :]:
2     # do some stuff with `x`
3     if x == 0:
4         seg = [x**2 for x in range(1000, 2000)]
5         blk = Block(apri, seg, 1000)
6         reg.add_disk_block(blk)
```

The iterator on line 1 will eventually yield the elements you saved on line 6.

3.3.3 Single `Blocks`

You can load in single blocks using the instance-methods `Register.get_disk_block_by_n` and `Register.get_disk_block_by_metadata`. Please read the API specification for their usage.

3.3.4 All `Blocks`

If you want to iterate over all disk `Blocks`, do the following:

```
for blk in reg.get_all_disk_blocks(apri):
    # do stuff with `blk`
```

For many applications, the iterator `get_all_disk_blocks` will be faster than the iterator `reg[apri, :]`. But for other applications, it may be easier to use `reg[apri, :]`.

4 How to use `Apri_Info`

You (the user) use the class `Apri_Info` to describe and identify sequences. Cornifer uses `Apri_Info` to store, search for, and retrieve sequence that you have calculated. The name `Apri_Info` is short for *a priori information*.

Due to the important function of `Apri_Info`, it is critical that you adhere to several rules and guidelines when constructing and working with `Apri_Info`. These rules and guidelines are outlined in Sections 4.1 and 4.2.

For the sake of concreteness, here are a few examples of `Apri_Info` and the sequences they describe. You can use any valid Python name you like on the left-hand-side of each = sign. The left-hand-side of each = sign is called a *key* and the right-hand-side is called a *keyword argument*.

```
Apri_Info(name = "primes")
```

The sequence of prime numbers. This `Apri_Info` appears throughout Section 6.1.

```
Apri_Info(name = "primes", mod4 = 1)
```

The primes p that satisfy $p \equiv 1 \mod 4$.

```
Apri_Info(name = "matrix powers", base = ((2,1),(1,1)))
```

The sequence of matrices $\begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}^k$ for some k.

```
Apri_Info(name = "function iterates", function = "z^2 - 1", starting_point = 0)
```

The iterates $f^n(0)$, where $f(z) = z^2 - 1$, for some n.

4.1 The rules

If you break any of the following rules, one of the following two things will happen: Either Cornifer is able to detect the misuse and will `raise` an `Exception`, or Cornifer is not able to detect the misuse and you will get bugs in your code and in your Cornifer database.

1. **You cannot create, modify, or delete any attributes of an `Apri_Info` instance.** The following lines will not `raise` an `Exception`, althouth the line `apri.mod4 = 1` will quickly confuse Cornifer and may corrupt your database.

   ```
   apri = Apri_Info(name = "primes")
   apri.mod4 = 1
   ```

   You should instead say

   ```
   apri = Apri_Info(name = "primes", mod4 = 1)
   ```

2. **You must use hashable types for all keyword arguments.** The following line will `raise Keyword_Argument_Error`.

   ```
   Apri_Info(name = "matrix powers", base = [[2,1],[1,1]])
   ```

   The type `list` is not hashable. Here are two lists of common datatypes, separated into hashable and un-hashable types. Hashable types: `str`, `int`, `float`, `tuple`. Un-hashable types: `list`, `dict`, `set`, Numpy `ndarray`.

3. **Keyword arguments of type `str` must only contain ASCII characters and cannot contain the null character `"\0"`.**
   The following line will `raise Keyword_Argument_Error`.

```
Apri_Info(name = "digits", base = 10, num = "π")
```

   The character π is not an ASCII character. You should instead say

```
Apri_Info(name = "digits", base = 10, num = "pi")
```

4.2 The guidelines

In contrast with the rules listed in Section 4.1, these guidelines are bendable and breakable. Not adhering to these guidelines will not directly lead to bugs or `raise` an `Exception`. However, it is not necessary to bend or break any of these guidelines. If you find yourself doing so, you are misusing `Apri_Info`.

1. **Your `Apri_Info` consists of information you know at the beginning of a calculation and nothing more,** hence the name *a priori information*.

   For example, say you are calculating the decimal digits $(d_i)_{i=-N}^{\infty}$ of a number x : Choose $N \in \mathbb{N}$ and and choose $0 \leqslant d_i < 10$ for $-N \leqslant i < \infty$ such that

$$x = \sum_{i=-N}^{\infty} d_i \cdot 10^{-i}.$$

   The `Apri_Info` you make for the sequence $(d_i)_{i=-N}^{\infty}$ is

```
Apri_Info(name = "digits", base = 10, num = x)
```

   Using Cornifer, you save the digits to disk as you calculate them. At the beginning of the calculation, you do not know whether $x \in \mathbb{Q}$ or not. After calculating some large number of digits, you discover that the digits are eventually periodic, hence $x \in \mathbb{Q}$. Say that the decimal digits $d_m, d_{m+1}, \ldots, d_{m+n-1}$ form the periodic portion of the sequence.

   You want to save the numbers n and m to the disk. You may be inclined to simply create a new `Apri_Info` instance with the information you have discovered, as follows:

```
Apri_Info(name = "digits", base = 10, num = x, prd = n, start_of_prd = m)
```

   However, you did not know m and n at the beginning of the calculation, so you are violating guideline 1.

   The correct way to save the numbers n and m is by using the class `Apos_Info`, short for *a posteriori information*. The class `Apos_Info` is described in Section TODO:ref.

2. **Your `Apri_Info` is unambiguous.**

   For example, the following `Apri_Info` is ambiguous:

```
Apri_Info(name = "digits", num = "pi")
```

   It is unclear if `"digits"` refers to base-10 digits or base-2, or any other base for that matter. It is better to be precise:

```
Apri_Info(name = "digits", base = 10, num = "pi")
```

3. **Your `Apri_Infi` describes exactly one sequence.**

   The following example is a bit contrived, but it serves to illustrate the point: Say you want to encode samples from a uniform distribution over the interval $[0,1]$. You make the following `Apri_Info`.

```
Apri_Info(name = "uniform samples", sample_space = "[0,1]")
```

   However, this does not describe a single sequence: If you changed the random seed, then the sequence of samples would change. So you include the seed with the `Apri_Info`.

```
Apri_Info(name = "uniform samples", sample_space = "[0,1]", numpy_seed = 1337)
```

5 Intermediate usage

5.1 `Apos_Info`

You (the user) use the class `Apos_Info` to encode additional information about sequences, beyond the scope of `Apri_Info`. The name `Apos_Info` is short for *a posteriori information*.

In Section 4, we listed several rules and guidelines for the class `Apri_Info`, such as that `Apri_Info` only encodes descriptive information about the sequence that you know before you have calculated any of the sequence's elements. In contrast to `Apri_Info`, the class `Apos_Info` encodes information that you have discovered after calculating some of the sequence's elements.

The classes `Apri_Info` and `Apos_Info` share a superficially similar syntax, but that is where the similarities end. `Apos_Info` is significantly more flexible than `Apri_Info`. For example, `Apos_Info` can have unhashable keyword arguments, as in the following example.

```
Apos_Info(max = {"interval" = (0,1000), ""}
```

5.2 RAM `Blocks`

A RAM `Block` is a `Block` that a `Register` knows about, but has not saved to the disk. Whenever you request information from the `Register`, the `Register` will include information from RAM `Blocks` whenever possible. It is useful to add a `Block` to a `Register` as a RAM `Block` when you want to get the most up-to-date information from a `Register`, but it is not yet necessary to save the `Block` to disk.

RAM `Blocks` do not alter the `Register` database in any way. So if you close the Python program or turn off your computer, RAM `Blocks` that have not been added as disk `Blocks` will be deleted.

A good use-case of a RAM `Block` is Floyd's algorithm for finding periods of recursively-defined sequences. An example of this use-case is given in Listing 4.

5.3 Subregisters

6 Line-by-line examples

6.1 Generating prime numbers

In this section, we will step through, line-by-line, an example of code that generates and saves prime numbers to the disk. The code we will look at is Listing 3.

```
1 from math import floor, sqrt
2 from cornifer import Apri_Info, Numpy_Register, Block
```

```python
from math import floor, sqrt
from cornifer import Apri_Info, Numpy_Register, Block

def is_prime(m):
    if not isinstance(m, int) or m <= 1:
        return False
    for k in range( 2, floor(sqrt(m)) + 1 ):
        if m % k == 0:
            return False
    return True

seg = []
apri = Apri_Info(name="primes")
blk = Block(seg, apri, 1)
my_saves_dir = "C:/Users/Michael/cornifer_saves"
reg = Numpy_Register(my_saves_dir, "primes example")

capacity = 100000
total_primes = 0
max_m = 10**9

with reg.open() as reg:

    for m in range(2, max_m+1):
        if is_prime(m):
            total_primes += 1
            seg.append(m)

        num = total_primes % capacity
        if (num == 0 and total_primes > 0) or (num > 0 and m == max_m):
            reg.add_disk_block(blk)

            seg.clear()
            blk.set_start_n(total_primes+1)
```

Listing 3: Generating prime numbers. A walkthrough of this code can be found in Section 6.1.

These lines import code from packages. Line 1 imports code from the Python Standard Library. Line 2 imports code from Cornifer.

```
4 def is_prime(m):
5     if not isinstance(m, int) or m <= 1:
6         return False
7     for k in range( 2, floor(sqrt(m)) + 1 ):
8         if m % k == 0:
9             return False
10    return True
```

The function `is_prime(m)` returns `True` if `m` is a prime number and `False` otherwise. This is a very simple implementation of a primality tester via trial division.[1]

```
12 seg = []
13 apri = Apri_Info(name = "primes")
14 my_saves_dir = "C:/Users/Michael/cornifer_saves"
15 blk = Block(seg, apri, 1)
16 reg = Numpy_Register(my_saves_dir, "primes example")
```

Lines 12 to 16 are Cornifer boilerplate. Please see Section 3.1 for a walkthrough of what these lines do.

On line 15, notice that we set the start index to 1, because usually 2 is called the “first prime number”, rather than the “zeroeth prime number”.

```
18 capacity = 10 ** 5
19 total_primes = 0
20 max_m = 10 ** 9
```

The name `capacity` is the maximal length of a `Block`. The name `total_primes` is the total number of prime numbers we have found so far; since we have not calculated any primes, we initialize `total_primes = 0`. The name `max_m` is the stopping point for the iteration; we test all numbers up to and including `max_m` and go no further.

```
22 with reg.open() as reg:
```

Line 22 opens the `Register` database for reading and writing. The database is automatically closed once your Python code leaves the `with` block.

```
24     for m in range(2, max_m+1):
25         if is_prime(m):
26             total_primes += 1
27             seg.append(m)
```

Lines 24 to 27 loop over all integers between 2 and `max_m`, inclusive, test each one for primality, and add the primes to the the `list` named `seg`.

```
29         num = total_primes % capacity
30         if (num == 0 and total_primes > 0) or (num > 0 and m == max_m):
31             reg.add_disk_block(blk)
```

[1] As an exercise for the reader, change the implementation of `is_prime` so that it is faster.

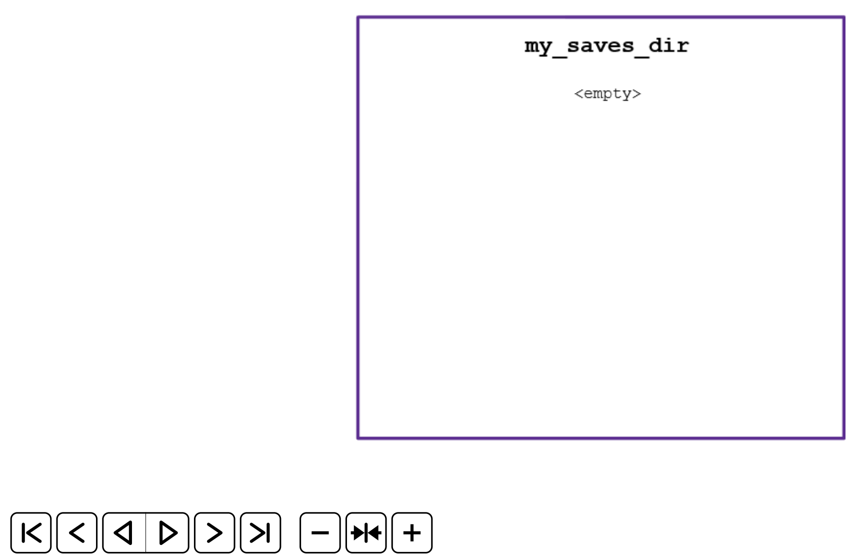
Lines 29 to 31 save `blk` to the disk whenever `total_primes` is a nonzero multiple of `capacity`, or when `m` reaches its maximum value, namely `max_m`.

```
33            seg.clear()
34            blk.set_start_n(total_primes+1)
```

On line 33, we modify the data segment `seg` by deleting its contents. Calling `seg.clear()` will not delete the `Block` we just saved to the disk, but it will free up RAM. On line 34, we move the start index of `blk` from its old position to the new one, namely to `total_primes + 1`.

6.1.1 A slide show of the prime number generator

Below is a slide show that illustrates lines 12 to 16 and lines 22 to 34 of Listing 3.



6.1.2 More comments on the prime number generator

It is important to always write line 22 as shown in Listing 3. **Do not write**

```
with reg.open():
```

because you will get bizarre bugs and you may corrupt your Cornifer database.

Lines 33 and 34 modify the existing `blk` *in-place*. Rather than lines 33 and 34, we could create an entirely new `Block`.

```
            seg = []
            blk = Block(seg, apri, total_primes+1)
```

Do not write the two lines below in place of lines 33 and 34.

```
            seg = []
            blk.set_start_n(total_primes+1)
```

The two lines above do not delete the data segment associated to `blk`. Rather, `seg = []` creates a new list and assigns the name `seg` to the new list. The line `seg = []` does not modify `blk`.

6.2 RAM `Block` usage and Floyd's algorithm

In this section, we will step through, line-by-line, an example of code that finds (not necessarily minimal) periods of a recursively defined sequence $(x_i)_{i=1}^{\infty}$. The code we will look at is Listing 4.

By recursively defined sequence, we mean that there exists a function f such that $f(x_i) = x_{i+1}$ for all $i \geqslant 1$. The algorithm that finds the period is called *Floyd's algorithm*. Floyd's algorithm works by checking if $x_i = x_{i/2}$ whenever i is even. If $x_i = x_{i/2}$, then the segment $x_{i/2}, x_{i/2+1}, \ldots, x_{i-1}$ is periodic.

This example contains a good use case of RAM `Block`s. Floyd's algorithm repeatedly queries the `Register` for information, information that has sometimes not been saved to disk. We use the `add_ram_block` method to make the `Register` aware of the information that has not yet been saved.

```
7 x_1 = 0
8 def f(x):
9     return (x**2 + 2) % 8407901
```

For the sake of concreteness, we have given examples of possible inputs for `x_1` and `f` in lines 7 to 9. But you can replace these two with whatever you want, in principle.

```
11 my_saves_dir = "C:/Users/Michael/cornifer_saves"
12 apri = Apri_Info(
13     name = "recursively defined",
14     initial_point = x_1,
15     next_function = "x^2 + 2 mod 8,407,901"
16 )
17 seg = [x_1]
18 blk = Block(seg, apri)
19 reg = Numpy_Register(my_saves_dir, "Floyd's algorithm example.")
```

Lines 11 to 19 are Cornifer boilerplate. Please see Section 3.1 for a walkthrough of what these lines do.

```
21 capacity = 1000
22 max_i = 10**7
```

The name `capacity` is the maximal length of any `Block`, both RAM `Block` and disk `Block`. The name `max_i` is the maximal index that we will compute of the recursively defined sequence $(x_i)_{i=1}^{\infty}$. If Floyd's algorithm reaches `max_i` and has not found a period, then the algorithm will return and report that no period has been found.

```
24 reg.add_ram_block(blk)
```

With line 24, the `Register` named `reg` is now aware of the `Block` named `blk`. Line 24 does not save anything to disk. Note that we did not need to `open` the `Register` in order to call `add_ram_block`, since that function does not actually alter the `Register` database in any way. We will see the consequences of line 24 later in this line-by-line walkthrough.

```
26 with reg.open() as reg:
```

Line 26 just opens the `Register` for reading and writing, as usual.

```
28     floyd_iter = reg[apri, :]
```

```python
# Input: A function f and an initial point x_1
# Output: A period of the recursively defined sequence x_{i+1} = f(x_i),
#     assuming it exists
# Note: This implementation will not return the minimal period,
#     but rather an integer multiple of the minimal period.

x_1 = 0
def f(x):
    return (x**2 + 2) % 8407901

my_saves_dir = "C:/Users/Michael/cornifer_saves"
apri = Apri_Info(
    name = "recursively defined",
    initial_point = x_1,
    next_function = "put a description of f here"
)
seg = [x_1]
blk = Block(seg, apri, 1)
reg = Numpy_Register(my_saves_dir, "Floyd's algorithm example.")

capacity = 1000
max_i = 10**6

reg.add_ram_block(blk)

with reg.open() as reg:

    floyd_iter = reg[apri, :]

    x = x_1
    for i in range(2, max_i + 1):
        x = f(x)
        seg.append(x)

        if i % 2 == 0:
            halfway_x = next(floyd_iter)
            if halfway_x == x:
                period = i//2

                if len(seg) > 0:
                    reg.add_disk_block(blk)
                break

        num = len(seg) % capacity
        if (num == 0 and i > 0) or (num > 0 and i == max_i):
            reg.add_disk_block(blk)
            seg.clear()
            blk.set_start_n(i+1)
```

Listing 4: An example of RAM `Block` usage, illustrated with Floyd's algorithm.

Line 28 creates an iterator and names it `floyd_iter`. Note that the call `reg[apri, :]` does not immediately yield any elements of the sequence we have calculated so far, either from RAM or from disk. In order to get that information, you must call the `next` method of the iterator, like `next(floyd_iter)`. The `next` method appears in line 36 of Listing 4, which we will discuss later in this walkthrough.

```
30     x = x_1
31     for i in range(2, max_i + 1):
32         x = f(x)
33         seg.append(x)
```

Lines 30 to 33 generate the elements of the recursively defined sequence and append them to the `list` named `seg`.

```
35         if i % 2 == 0:
36             halfway_x = next(floyd_iter)
37             if halfway_x == x:
38                 period = i//2
```

Lines 35 to 38 are Floyd's algorithm. The next method on line 36 is called only every other iterate of the `for` loop starting on line 31, so line 36 is equivalent to saying `halfway_x = reg[apri, i//2]`. However, writing `halfway_x = reg[apri, i//2]` is very bad practice; please see the explanation in Section 6.2.2. TODO

```
40                 if len(seg) > 0:
41                     reg.add_disk_block(blk)
42                 break
```

Lines 40 to 42 save the remaining elements to disk and terminate the `for` loop starting on line 31.

```
44         num = len(seg) % capacity
45         if (num == 0 and i > 0) or (num > 0 and i == max_i): |\label{stuff}|
46             reg.add_disk_block(blk)
47             seg.clear()
48             blk.set_start_n(i+1)
```

Lines 44 to 48 save the `Block` named `blk` to disk whenever `len(seg)` is a nonzero multiple of `capacity`, or when `i` reaches its maximum value, namely `max_i`. Lines 44 to 48 are very similar to lines 29 to 34 of Listing 3.

6.2.1 A slide show of Floyd's algorithm

TODO

6.2.2 More comments on Floyd's algorithm

TODO

7 Advanced usage

7.1 Creating your own `Register` derived type

8 Glossary

- method

- class
- RAM
- disk
- `Register`
- `Apri_Info`
- `Apos_Info`
- `Block` (RAM/disk)
- Subregister