Research Shell

Maciej Trawka

December 27, 2022

Contents

1 Overview **2**

1.1 Architecture of Research Shell . . . 2

2 Class Polynomial **4**

2.1 Polynomial object methods . . . 4

2.2 Static Polynomial methods . . . 9

3 class Lfsr **10**

3.1 Lfsr types . . . 10

3.1.1 Fibonacci . . . 10

3.1.2 Galois . . . 10

3.1.3 Ring Generator . . . 11

3.1.4 Ring with manually specified taps . . . 11

3.1.5 Hybrid Ring Generator . . . 11

3.1.6 Tiger Ring Generator . . . 11

Chapter 1

Overview

The Research Shell is in fact a Python3 shell wrapped by `PtPython` with some useful modules, classes and methods included. This document covers those items assuming, that a reader is familiar with Python syntax.

1.1 Architecture of Research Shell

Look at Figure 1.1. It shows Research Shell wrappers from the top to the Python3 core. User calls a Bash script, which prepares and executes a command containing Python3 call, module PtPython loading, and then importing all useful libraries included in module `aio`. So to run the Research Shell you need to call:

```
research_shell [python_file_name_to_execute]
```

If no argument, then the Research Shell appears and is ready to execute Python commands. If a script file is specified as an argument, then its content is executed after importing all modules and the shell closes.

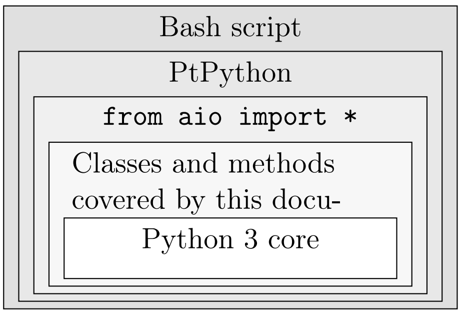

Figure 1.1: Research Shellarchitecture.

There is also a special mode of Research Shell, called **Testcase Mode**. It makes easy to execute a complete testcases. The testcase is a directory having a regular structure:

`testcase_name`
- `data` .... automatically added to the searching path
- `results` .... Created automatically by Research Shell
- `driver.py` .... Main script - your Testcase code

By running the command:

```
research_shell_drun
```

the Research Shell runs in the Testcase mode. In such case it checks if the `driver.py` file exists. If so, then it removes and recreates the `results` directory and goes there (so `results` is now the Current Directory). Now, a content of `dirver.py` is executed. In `results` directory a `transcript.txt` is created. To print something to the screen and also to the transcript file, you need to call the `print(*args)` method of class `Aio`, i.e.:

```
# This text will be printed to the screen only:
print("Text on the screen only")

# This also appears in the transcript file:
Aio.print("Text on the screen and in the transcript")
```

Chapter 2

Class Polynomial

`Polynomial` is an object intended to analyze polynomials over GF(2). An object of type `Polynomial` holds polynomial coefficients (as a list of positive integers) and a list of signs of those coefficients. Of course in case of GF(2) coefficient $x_i = -x_i$. However, negative coefficients make sense in case of some types of LFSRs, as `Polynomial` objects are used to create other objects, of type of `Lfsr`.

Below you can see an example of how to create a `Polynomial` object representing the polynomial $x^{16} + x^5 + x^2 + x^0$:

```
p1 = Polynomial ( [16, 5, 2, 0] )
p2 = Polynomial ( 0b10000000000100101 )
p2 = Polynomial ( 0x10025 )
```

`Polynomial` class includes also a couple of static methods, especially useful to search for primitive polynomials and other ones discussed in the next part of this chapter.

2.1 Polynomial object methods

`Polynomial_object.__str__()`

`Polynomial` objects are convertible to strings.

```
p1 = Polynomial ( [16, 5, 2, 0] )
print(p1)
# >>> [16, 5, 2, 0]
```

`Polynomial_object.__hash__()`

`Polynomial` objects are hashable. Can be used as a dictionary keys.:

```
p1 = Polynomial ( [16, 5, 2, 0] )
d = {}
d[p1] = "p1 value"
```

`Polynomial_object.copy()`

Returns a deep copy of the `Polynomial` object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
p2 = p1.copy()
print(p1 == p2)
```

```
# >>> True
```

`Polynomial_object.derivativeGF2()`

Returns symbolic derivative Polynomial.

```
p1 = Polynomial ( [16, 15, 2, 1, 0] )
print(p1.derivativeGF2())
# >>> [14, 0]
p2 = Polynomial ( [16, 14, 5, 2, 0] )
p3 = p2.derivativeGF2()
print(p3)
# >>> [4]
```

`Polynomial_object.getBalancing()`

Returns a difference between distances of furthest and closest coefficients.

```
p1 = Polynomial ( [16, 10, 2, 0] )
#     distances:        6   8  2
#     furthest:             8
#     closest:                 2
#   furthest-closest:  8-2 = 6
p1.getBalancing()
# >>> 6
```

`Polynomial_object.getCoefficients()`

Returns a REFERENCE to the sorted list of (unsigned) coefficients.

```
p1 = Polynomial ( [16, 2, -5, 0] )
print(p1.getCoefficients())
# >>> [16, 5, 2, 0]
```

`Polynomial_object.getCoefficientsCount()`

Returns count of the `Polynomial` object coefficients.

```
p1 = Polynomial ( [16, 5, 2, 0] )
coeffscount1 = p1.getCoefficientsCount()
print(coeffscount1)
# >>> 4
```

`Polynomial_object.getDegree()`

Returns degree of the `Polynomial` object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
deg1 = p1.getDegree()
print(deg1)
# >>> 16
```

`Polynomial_object.getDifferentTapCount(AnotherPolynomial)`

Imagine, that the `Polynomial_object` is a characteristic polynomial of a Ring Generator. Then this method compares the `Polynomial_object` with another polynomial (also being a characteristic one of a Ring Generator) and returns a number of NOT matching taps. Tap direction (given by coefficient sign) does not matter.

```
p1 = Polynomial ( [16, -5, 2, 0] )
p2 = Polynomial ( [16, 5, 2, 0] )
p3 = Polynomial ( [16, 5, 1, 0] )
p4 = Polynomial ( [16, 6, 0] )
p1.getDifferentTapCount(p2)
# >>> 0
p1.getDifferentTapCount(p3)
# >>> 1
p1.getDifferentTapCount(p4)
# >>> 2
p4.getDifferentTapCount(p1)
# >>> 1
```

`Polynomial_object.getMinDistance()`

Returns a distance between closest Polynomial's coefficients.

```
p1 = Polynomial ( [16, 5, 2, 0] )
#     distances:     11  3  2
p1.getMinDistance()
# >>> 2
```

`Polynomial_object.getReciprocal()`

Returns a new, reciprocal Polynomial object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
print(p1.getReciprocal())
# >>> [16, 14, 11, 0]
```

`Polynomial_object.getSignedCoefficients()`

Returns sorted list of signed coefficients.

```
p1 = Polynomial ( [16, 2, -5, 0] )
print(p1.getSignedCoefficients())
# >>> [16, -5, 2, 0]
```

`Polynomial_object.getSigns()`

Returns signs of all sorted coefficients (as a list of 1s and -1s).

```
p1 = Polynomial ( [16, -5, 2, 0] )
print(p1.getSigns())
# >>> [1, -1, 1, 1]
```

`Polynomial_object.isLayoutFriendly()`

Returns True if a Ring Generator, based on the Polynomial_object, is layout friendly. It checks if the minimum distance between successive coefficients is at least 2.

```
p1 = Polynomial ( [16, 15, 2, 1, 0] )
p1.isLayoutFriendly()
# >>> False
p2 = Polynomial ( [16, 14, 5, 2, 0] )
p2.isLayoutFriendly()
# >>> True
```

`Polynomial_object.isPrimitive()`

Returns True if the given polynomial is primitive over GF(2). All coefficients are considered to be positive. Note, that the first call of this method may take more time than usual, because of prime dividers database loading. This methods bases on fast simulation of LFSRs described in [1].

```
p1 = Polynomial ( [16, 5, 2, 0] )
p1.isPrimitive()
# >>> False
p2 = Polynomial ( [4, 1, 0] )
p2.isPrimitive()
# >>> True
```

`Polynomial_object.iterateThroughSigns()`

This is generator method. Each time yields new `Polynomial` object with other combinations of coefficient signs. Note, that the highest and lowest coefficients are untouched. All-positive and all-negative combinations are also not yielded.

```
p1 = Polynomial ( [16, -5, 2, 1, 0] )
for pi in p1.iterateThroughSigns(): print(pi)
# >>> [16, -5, 2, 1, 0]
# >>> [16, 5, -2, 1, 0]
# >>> [16, -5, -2, 1, 0]
# >>> [16, 5, 2, -1, 0]
# >>> [16, -5, 2, -1, 0]
# >>> [16, 5, -2, -1, 0]
```

`Polynomial_object.nextPrimitive(Silent=True)`

Tries to find next polynomial which is primitive over GF(2). Returns True if found, otherwise returns False. if `Silent` argument is False, then searching process is shown in the terminal.

```
p1 = Polynomial ( [16, 15, 2, 1, 0] )
p1.nextPrimitive()
# >>> True
print(p1)
# >>> [16, 12, 3, 1, 0]
p1.nextPrimitive()
# >>> True
print(p1)
# >>> [16, 6, 4, 1, 0]
```

`Polynomial_object.printFullInfo()`

Prints (also to the transcript in testcase mode) full info about the Polynomial_object. See the example below:

```
p1 = Polynomial ( [16, 5, 2, 0] )
p1.printFullInfo()
#
# ---------------------------
# Polynomial  deg=16, bal=9
# ---------------------------
#
# Degree             :  16
# Coefficients count:  4
# Hex with degree    :  10(2)25
# Hex without degree:  25
# Balancing          :  9
# Is layout-friendly:  True
# Coefficients       :  [16, 5, 2, 0]
```

`Polynomial_object.setStartingPointForIterator(StartingPolynomial)`

`Polynomial` object may be used as generators, to iterate through all possible polynomials with respect to some requirements (see createPolynomial() method). This one is used to set starting point for iterator. See the example below. Note, that `StartingPolynomial` may be another `Polynomial` object, or a list of coefficients. The starting polynomial is checked to have the same degree and coefficients count as the Polynomial_object.

```
p1 = Polynomial ( [6,1,0] )
for pi in p1: print(pi)
# >>> [6, 1, 0]
# >>> [6, 2, 0]
# >>> [6, 3, 0]
# >>> [6, 4, 0]
# >>> [6, 5, 0]
p1.setStartingPointForIterator( [6,3,0] )
for pi in p1: print(pi)
# >>> [6, 3, 0]
# >>> [6, 4, 0]
# >>> [6, 5, 0]
p1.setStartingPointForIterator( Polynomial([6,4,0]) )
for pi in p1: print(pi)
# >>> [6, 4, 0]
# >>> [6, 5, 0]
```

`Polynomial_object.toBitarray()`

Returns a bitarray object representing the Polynomial.

```
p1 = Polynomial ( [16, 5, 2, 0] )
p1.toBitarray()
# >>> bitarray('10100100000000001')
```

`Polynomial_object.toHexString(IncludeDegree=True, shorten=True)`

Returns a string of hexadecimal characters describing the Polynomial_object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
p1.toHexString()
# >>> '10(2)25'
p1.toHexString(IncludeDegree=False)
# >>> '25'
p1.toHexString(shorten=False)
# >>> '10025'
```

`Polynomial_object.toInt()`

Returns an integer representing the Polynomial object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
bin(p1.toInt())
# >>> '0b10000000000100101'
```

`Polynomial_object.toMarkKassabStr()`

Returns a string used by Mark Kassab's C++ code to add a polynomial to the internal database.

```
p1 = Polynomial ( [16, -5, 2, 0] )
p1.toMarkKassabStr()
# >>> 'add_polynomial(16, 5, 2, 0);'
```

2.2 Static Polynomial methods

Chapter 3

class Lfsr

`Lfsr` is an object type allowing to simulate and analyze any type of Linear Feedback Shift Register. Simulations are performed using `bitarray` objects, where `bitarray_object[N]` holds value of Flip-Flop having index *N*. `Lfsr` objects are always simulated assuming, that data is shifted from higher to lower indexed flip-flop (from FF[1] to FF[0], from FF[2] to FF[1] and so on).

3.1 Lfsr types

3.1.1 Fibonacci

Look at the Figure 3.1. It shows an example of Fibonacci LFSR implementing the polynomial of $x^8 + x^6 + x^5 + x^2 + 1$. There are couple of ways to create such `Lfsr` object:

```
# using existing Polynomial object:
p1 = Polynomial([8,6,5,2,0])
lfsr1 = Lfsr(p1, FIBONACCI)
# using Polynomial created in place:
lfsr1 = Lfsr(Polynomial([8,6,5,2,0]), FIBONACCI)
# using coefficients list:
lfsr1 = Lfsr([8,6,5,2,0], FIBONACCI)
```

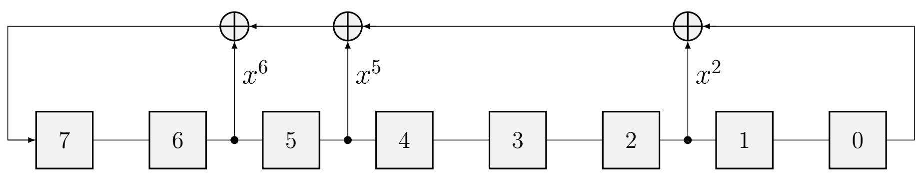

Figure 3.1: Fibonacci LFSR implementing polynomial $x^8 + x^6 + x^5 + x^2 + 1$.

3.1.2 Galois

Figure 3.1 shows an example of Galois LFSR implementing the polynomial of $x^8+x^6+x^5+x^2+1$. There are couple of ways to create such `Lfsr` object:

```
# using existing Polynomial object:
p1 = Polynomial([8,6,5,2,0])
lfsr1 = Lfsr(p1, GALOIS)
# using Polynomial created in place:
lfsr1 = Lfsr(Polynomial([8,6,5,2,0]), GALOIS)
# using coefficients list:
lfsr1 = Lfsr([8,6,5,2,0], GALOIS)
```

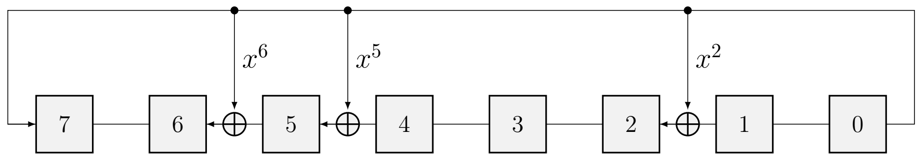

Figure 3.2: Galois LFSR implementing polynomial $x^8 + x^6 + x^5 + x^2 + 1$.

3.1.3 Ring Generator

3.1.4 Ring with manually specified taps

3.1.5 Hybrid Ring Generator

3.1.6 Tiger Ring Generator

Bibliography

[1] Nilanjan Mukherjee, Janusz Rajski, Grzegorz Mrugalski, Artur Pogiel, and Jerzy Tyszer. Ring generator: An ultimate linear feedback shift register. *Computer*, 44(6):64–71, 2011.

Index

Research Shell
  architecture, 2
__hash__, 4
__str__, 4

copy, 4

derivativeGF2, 5

Fibonacci LFSR, 10

Galois LFSR, 10
getBalancing, 5
getCoefficients, 5
getCoefficientsCount, 5
getDegree, 5
getDifferentTapCount, 5
getMinDistance, 6
getReciprocal, 6
getSignedCoefficients, 6
getSigns, 6

Hybrid Ring Generator, 11

isLayoutFriendly, 6
isPrimitive, 7
iterateThroughSigns, 7

LFSR, 10
  Fibonacci, 10
  Galois, 10
  Hybrid Ring Generator, 11
  Ring Generator, 11
  Ring with specified taps, 11
  Tiger Ring Generator, 11

nextPrimitive, 7

Polynomial, 4
Polynomial_object
  __hash__, 4
  __str__, 4
  copy, 4
  derivativeGF2, 5
  getBalancing, 5
  getCoefficients, 5
  getCoefficientsCount, 5
  getDegree, 5
  getDifferentTapCount, 5
  getMinDistance, 6
  getReciprocal, 6
  getSignedCoefficients, 6
  getSigns, 6
  isLayoutFriendly, 6
  isPrimitive, 7
  iterateThroughSigns, 7
  nextPrimitive, 7
  printFullInfo, 7
  setStartingPointForIterator, 8
  toBitarray, 8
  toHexString, 8
  toInt, 9
  toMarkKassabStr, 9
printFullInfo, 7

Ring Generator, 11
Ring with manually specified taps, 11

setStartingPointForIterator, 8

Tiger Ring Generator, 11
toBitarray, 8
toHexString, 8
toInt, 9
toMarkKassabStr, 9