

Deep Encoding: Where Genetic  
Algorithms meet Numeral Systems

Daniel Ward-Williams

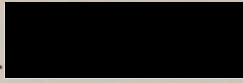
Submitted to Swansea University in fulfilment of the requirements for the Degree of MRes Logic and  
Computation

Swansea University

2022

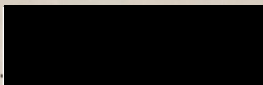
**Declarations**

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed.....  .....


Date..... 06/10/2022 .....

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed.....  .....

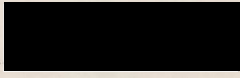
Date..... 06/10/2022 .....

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed.....  06/10/2022 .....

Date..... 06/10/2022 .....

The University's ethical procedures have been followed and, where appropriate, that ethical approval has been granted.

Signed.....  .....

Date..... 06/10/2022 .....

# Deep Encoding: Where Genetic Algorithms meet Numeral Systems

Daniel Ward-Williams

2021  
September

## **Abstract**

It has been shown empirically that certain benefits can be gained by modelling genetic algorithm encoding as a numeral system and implementing mutation as a form of the numeral system's arithmetic. It is the aim of this project to strengthen these findings. We will do this in three stages.

Firstly, by creating meaningful classifications of numeral systems and formally proving crucial properties such as termination of standardisation and normalisation.

Secondly, by developing a programming framework centered around these system classes. The framework is used on strings and can impose numeral systems on them. This allows the user to write code that can be run with a selection of systems to see different results. For example, taking the string "10" and treating it as binary or decimal depending on what the user dictates.

Thirdly, by writing a genetic algorithm and using the aforementioned framework to write an encoding method and mutation function that are based off of numeral system arithmetic. The mutation function adds a random unit value to the digit string and mutates the string by utilising arithmetic overflow.

# 1 Introduction

The motivation of this thesis is to shed light on the logical aspect of modelling search spaces as numeral systems. This will be done through the lens of genetic algorithms. Evolutionary algorithms (EAs) attempt to solve or optimise a problem over a period of cycles. The process mimics biological evolution over generations of mating. The algorithm consist of a population of candidate solutions and by means of some measurement function they select the best candidates to breed the new population from. After a perfect solution is found, or after the cycle limit is reached, the best candidate is returned as the result. There are different types of evolutionary algorithm such as genetic algorithms (GAs) and genetic programming (GPs) but they all have the same framework in common: 1) encoding solutions, 2) selecting solutions, 3) manipulating solutions. A problem can be a great manner of things including and not limited to solving for  $x$  in an equation, designing a circuit-board, or making a game character walk. In the most simple case of solving for  $x$  we can do a simple model. The most complicated part of this, and most evolutionary algorithms, is the fitness function. In the example,  $i$  is the index of the current candidate solution being looked at. So with a population of 100 there are one hundred possible values of  $x$ . The fitness function normalises each fitness over the sum of the whole population.

- Problem:  $p(x) = x^2 - x - 2$
- Goal: Find an  $x_i$  where  $p(x_i) = 0$
- Fitness:  $f(x) = |x^2 - x - 2|$
- Encoding: Binary 4bit
- Mutation: randomly flip some 0s to 1s and vice versa

Here we actually have two possible perfect solutions (global optima) which are  $x = 2$  and  $x = -1$  but because of our encoding only  $x = 2$  will be available to us as there is no signed bit flag in our encoding method. Mutation is our way of manipulating the best of the population when we create the next population. Using an example population size of four with individuals  $\{0000, 0101, 1100, 1001\}$ . The decoded versions (interpreting them as binary) are  $\{0000_2 = 0, 0101_2 = 5, 1100_2 = 12, 1001_2 = 9\}$ . We can now use the fitness function which calculates the individual fitnesses giving us  $\{f(0) = 2, f(5) = 18, f(12) = 130, f(9) = 70\}$ . So for each  $x_i$  we have a corresponding fitness  $y_i$ , to get a normalised result we do  $y'_i = 1 - \frac{y_i}{y_{worst}}$  where  $y_{worst} = 130$  the worst fitness in the population. This gives us the normalised fitness for each individual in the population  $\{\frac{64}{65}, \frac{56}{65}, 0, \frac{6}{13}\}$ . The best solution from that generation was 0000, if we mutate this to create another population we might get  $\{0100, 0010, 1101, 0110\}$  and now on generation two we have our perfect solution 0010 (which has a fitness  $f(2) = 0$  which normalised is 1). For larger problems



more complicated encoding methods and fitness functions are needed. This thesis looks at the encoding and mutation stages of a genetic algorithm and their relationship to positional numeral systems such as binary and decimal etc. The term deep encoding refers to the combined model of encoding and mutation as a numeral system. Encoding both representation of the search space and traversal of it.

The author presented empirical findings for the increased accuracy of a GA result when using deep encoding with a specific numeral system related to the problem being solved[1]. The two significant results were a comparison of different encoding types for a GA run 1000 times. One of the problems being optimised was the Rosenbrock function, discussed [Eq. (27), p55], which showed noticeably higher fitness values when using encoding and mutation based on a special golden ratio numeral system. The other was a GA evolving an image of coloured squares following the line of a Fibonacci spiral, this problem benefited from a binary encoding and mutation. While the results were interesting there was no logical analysis of numeral systems or the algorithms used. The code was also constricted to exactly what was being tested.

This thesis builds on [1] but focuses more so on underpinning concepts, in particular it adds formal proofs of properties of numeral systems, and presents a powerful Common Lisp library built specifically for easy numeral system manipulation.

## 1.1 No Free Lunch Theorem

When looking at optimisation functions it is pertinent to note that there is no optimal method for all optimisation problems. This is the No Free Lunch Theorem (NFL), which states that in the context of evolutionary algorithms "any two algorithms are equivalent when their performance is averaged across all possible problems" [2]. The algorithms discussed in [2] are from a limited group of black-box optimisation algorithms such as genetic algorithms, and some other comparable algorithms. The performance of the algorithms is not measured by run-time to remove any bias of computational complexity [2]. Roughly speaking the benefit a search algorithm might gain when applied to a certain problem will be a loss when applied to some other problem. So there is no absolute best algorithm for universal use.

A version of NFL underpins the work in this thesis. When a problem is numeric by nature then there should exist numeral systems that best fit it, though there is no one numeral system that best represents all numeric search spaces. At the end of section 2 we give an example of how one might find an optimal numeral representation.

In their paper [2] Wolpert and Macready state "This highlights the need for exploiting problem-specific knowledge to achieve better than random performance." which is the aim of our work. To better fit the search space to the problem.

## 1.2 The Doorless Hotel Hypothesis

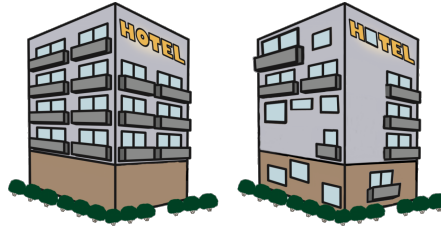


Figure 1: Imagine two hotels with no doors, only windows. One hotel has uniform window placements of equal size, the other hotel has a non-uniform unaligned differently sized distribution of windows. Now imagine to get into any arbitrary room through a window we need to build scaffolding around the hotel.

Genetic algorithms, are layered with abstraction and so can be hard to augment in ways that will certainly benefit the problem being optimised. Our research will be looking solely at function optimisation as we believe it is where the combination of numeral systems and genetic algorithms will show the most obvious and explainable benefit. It may be better described in an analogy we shall refer to as The Doorless Hotel which focuses on two parts. Imagine there is a tall hotel with no doors but many non-uniformly placed different sized windows, to get into specific rooms we would need to build scaffolding around the building. Normally scaffolding will have set heights and flat uniform levels, so only certain windows will be accessible.

Depending on which windows we want to get to we may have to change the height or include slopes. The hotel can be seen as the problem space and the scaffolding as the search space given to us by our numeral system encoding. When building the scaffolding we need to think about two specific things:

1. Alignment characteristics - The height of each level of the scaffolding can be seen as a representation of the numeral systems radix. For example each floor of the hotel is the next power up  $\{1,10,100,1000,\dots\}$ .
2. Traversal characteristics - The different arithmetic of each numeral system can be seen as different methods of moving through the scaffolding (e.g., ladders, slides, wormholes, reinterpretations, etc).

**Claim 1.** The Doorless Hotel Hypothesis, Part 1

*There is an optimal alignment and traversal method for navigating the search space of optimisation problems. As shown with the machine numbers there is a mathematically defined benefit to optimising encoding (alignment), and in previous work empirical results attest to the inclusion of matching traversal methods [1].*

### 1.2.1 Colour Spaces

An example of different scaffolding being used to access a doorless hotel can be seen in the use of different colour-spaces. There already exists classifications for representation systems in the form of colour-spaces [3] where RGB, HSL, CIE LMS, CIE Lab and other spaces all hold attributes from human visual perception, another classing can be application-based models such as YUV, YIQ, CMYK which are used for specific purposes like TV signals or printing. Another class could be the CIE standard colour-spaces that all derive from CIE XYZ and have the properties of being measurably device-independent and perceptually linear, these are CIE XYZ, Lab, Luv, LMS, LCH ab , LCH uv and the CIE  $\Delta E$  colour measurement.

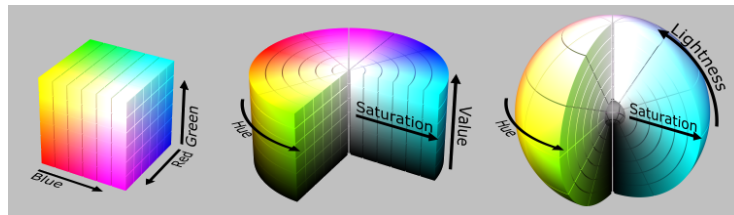


Figure 2: Three different colour-space models (RGB cube, HSL cylinder, and a spherical space such as LCH), representing the same information for different uses. Image credit [4].

### 1.3 Philosophy and Semantics of Number

Bertrand Russel states a subtle distinction between 'number' and 'numbers': *Number is what is characteristic of numbers, as man is what is characteristic of men. A plurality is not an instance of number, but of some particular number. A trio of men, for example, is an instance of the number 3, and the number 3 is an instance of number; but the trio is not an instance of number.*

In a similar vein we would like to separate the concept of number and representation. Number stands apart from representation in that instances of number hold the same value regardless of the instance of representation. Representation is a symbolic system that, until evaluated, stands apart from number in that arithmetic rules can exist as symbolic substitutions (seen as a state machine) without the need of a number.

By this somewhat Platonistic statement we're saying arithmetic exists in both numeric and symbolic forms. That is to say for some arithmetic procedure  $\alpha$  that can be performed on some number  $\varphi$  to obtain  $\psi$ , there is a corresponding string substitution state machine model that can transition from  $s$  to  $t$  via some rule set  $\aleph_f$ . Where  $s$  and  $t$  are strings of digits and  $f$  is some numeral system encoding.

$$\begin{array}{ccc}
 \varphi & \xrightarrow{\alpha} & \\
 f \uparrow & & f \uparrow \\
 s & \xrightarrow{\aleph_f} & t
 \end{array} \tag{1}$$

With this in mind we can imagine a set of representations that relate to each other as a class  $F = \{f, f', f'', \dots\}$ , and possible other classes in a universal set of all possible representations  $\mathfrak{U} = \{F, G, H, \dots\}$ . We may choose to see  $F$  as all numeral systems with an integer radix, giving the class  $F = \{f_1, f_2, f_3, \dots\}$  where  $f_1$  would be unary and  $f_8$  would be octal and so on. Then we might want to talk about a class as all systems using a radix of 3 but with differing digit sets  $G = \{g_{maximal}, g_{balanced}, g_{minimal}, \dots\}$ , here 'maximal' uses digit values  $\{0,1,2\}$ , 'balanced' uses  $\{-1,0,1\}$ , and 'minimal' uses  $\{1,2,3\}$ . There are many different ways we can choose to classify the representations and it is our goal to create meaningful classifications when used as search space encoding for genetic algorithms.

### 1.4 Summary of Findings

In this these we are able to complete the following goals

- Review of historic and modern numeral systems.
- Define a class of numeral systems that are compatible for use in search space encoding.
- Define a general method for generating arithmetic rules for said systems.

- Introduce a family of numeral systems called the metallic numeral systems for our focus
- Prove that arithmetic operations devised for metallic numeral systems will terminate on any valid digit string.
- Write a Common Lisp library, System Interpreted Numbers, for designing and using different numeral systems in a dynamic manner.
- Define a new method for generating fractals using the dynamic nature of System Interpreted Numbers.
- Write an evolutionary algorithm that uses System Interpreted Numbers in its genotype decoding and in a custom mutation function - Arithmutation.
- Research methods of evolutionary algorithm analysis.
- Put forward a search problem and analysis method that could be used to measure benefits of numeral system encoding.

## 1.5 Overview

Because this project covers three main topics it will be split into three main sections.

The first is a broad look at positional numeral systems, henceforth referred to as numeral systems, and their various properties. The goal of this section is to classify the characteristics of the numeral systems we'll be using and has various proofs.

The second section is on the development of a Common Lisp library for defining numeral systems and manipulating them in various ways. This section has a proofs relating to the code implementation.

The third section will summarise evolutionary/genetic algorithms, describe a Common Lisp genetic algorithm, then finally combine together numeral systems and genetic algorithms for exploring novel search spaces.

## 1.6 Preliminaries

$\mathbb{N}$	is the set of natural numbers (not including zero)	$\{1, 2, 3, 4, 5, \dots\}$
$\mathbb{N}^{>1}$	is the set of natural numbers greater than one	$\{2, 3, 4, 5, \dots\}$
$\mathbb{Z}^+$	is the set of non-negative integers	$\{0, 1, 2, 3, \dots\}$
$\mathbb{Z}[i]$	for $i^2 = -1$ is the set of Gaussian integers	$\{a + bi   a, b \in \mathbb{Z}\}$
$[d_0; d_1, d_2, \dots]$	is shorthand for the continued fraction	$d_0 + \frac{1}{d_1 + \frac{1}{d_2 + \frac{1}{\dots}}}$





Here we find the very well known and widely used Hindu-Arabic numerals. In fact our first three examples (Arabic, Eastern Arabic, and Devanagari) all use the same numeration model and only differ by the character glyphs used as the digit symbols. The Latin script gives the familiar 0123456789, Arabic script champions ٠١٢٣٤٥٦٧٨٩, and Devanagari, the script used for writing Hindi, uses ०१२३४५६७८९. These use positional notation and along with the many other linguistic script variations are the most universally used numeral systems.

Positional notation is where the symbol and position of the digits determine the value (its place effects its value). For example the system we know well is decimal or base 10, the 'base' refers to what is called a radix which lies at the heart of a number. If we break down 2021 the relationship becomes clear:

$$2021_{10} = (2 * 10^3) + (0 * 10^2) + (2 * 10^1) + (1 * 10^0)$$

The general case where  $d_n$  is the  $n$ th numeral (digit) and  $\beta$  is the radix we have

$$(d_n d_{n-1} \dots d_0)_\beta = d_n * \beta^n + d_{n-1} * \beta^{n-1} + \dots + d_0 * \beta^0 \tag{2}$$

which encompasses decimal, octal, hexadecimal, etc, along with other symbol systems such as the mixed value Mayan and Babylonian numeral systems. Uniquely the tally system of Unary is both place-value and non-place-value, as each new number is represented by adding a 1 while never violating the positional notation (because  $1^n = 1$  for all  $n$ ).

The Incan quipu is a physical numeral system, see [Fig. 3, p9]. Each hanging string acts as a unit column much like the index of  $d_n$  in equation 1, and the combination of knots in said string acts as the numeral value of  $d_n$  (representing 1, 2, 3, ...) [7].

There is a Chinese positional system that doesn't use positional notation like the above examples, it uses an almost conversational method for describing numbers. The numeral values 0 to 9 are present alongside values for 10, 100, 1000, 10000. The system works by pairing a unit value with a power of ten except in the case of  $10^1$  as the unit value on its own will suffice. This system has a standard glyph set for the numeral symbol and a financial glyph set designed to stop forgers changing between values easily. For example 一千〇三 is 1003 for everyday use, but for financial or governmental use 一仟〇叁 is 1003.

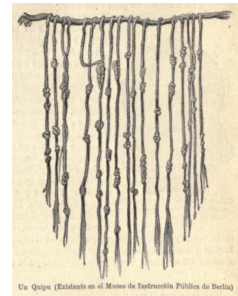
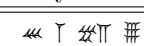

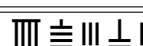




Figure 3: Sketch: Incan quipu [6].

#### Mixed-value Systems

Babylonian	Mayan	Chinese Rod	Attic	Roman
 (6486729)	 (2012)	 (99361)	 (50067)	 (1958)

Note that here Babylonian numerals have their own repeating symbol construction rules for the values from 1-59 and then those symbols are used in the place-value system. The values 1-59 have unique glyphs constructed by the two internal symbols  $\Upsilon$  and  $\leftarrow$ . Roman Numerals are another mixed-value system because of the special case where putting a lesser numeral before a larger one indicated a subtraction,  $XI = 11$  but  $IX = 9$ . This is the inverse, a place-value system within a non-place-value system.

While there are many visually differing symbols used for numbering methods, the one we will be focused on is the Western Arabic Numerals

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

and otherwise defined extensions of these such as adding letters for hexadecimal and so on.

The idea of using non-decimal numeral systems for their advantages is not new, we use binary for logical computation, octal and hexadecimal for compact storage of binary values, even a mixed-radix positional system for date-time in the form [60s, 60m, 24h, 30d, 12m, 365y] with various rules. In the example of date-time each unit is internally represented in decimal with standard numerals, but it is essentially one combinatorial numeral system describing one value.

## 2.2 Overview of Modern Numeral Systems

The system of the Hindu-Arabic symbols is called a positional numeral system and is defined by a radix (often referred to as the *base*) and a set of valid digits. What is usually called a decimal point is more generally a radix point for  $\beta = 10$ .

### Definition 1. Positional Numeral System

Let  $\beta \in \mathbb{C}, |\beta| > 1$  be the radix of the system. Where  $\mathbb{C}$  denotes the set of complex numbers and  $|\beta|$  denotes the absolute value of  $\beta$ .

Let the digit set  $D = \{0, 1, \dots, \lceil |\beta| \rceil - 1\}$  and digits  $d_i \in D$ .

The positional numeral encoding of value  $x$  as digits  $d_n d_{n-1} d_{n-2} \dots$  is

$$x = d_n * \beta^n + d_{n-1} * \beta^{n-1} + d_{n-2} * \beta^{n-2} + \dots$$

The digit string representing  $x$  can be of finite or infinite length. We describe the system as  $\langle \beta, D \rangle$ . In some cases a unique subscript is used i.e.  $x_{bt}$  (balanced ternary).

### 2.2.1 Notation

We shall now give an example of the syntax and corresponding semantics of [Eq. (1), p6] by means of describing the relationship between notation and value.

### Definition 2. Positional Notation for General Radixes

Let  $+$  be the arithmetic addition operator  $+: \mathbb{C}^2 \rightarrow \mathbb{C}$ .

Let  $\beta \in \mathbb{C}$  be a radix.

Let  $S$  be the set of digit strings valid in language  $L_\beta$

Let  $f_\beta$  be the function for evaluating a digit string  $f_\beta : L_\beta \rightarrow \mathbb{C}$ .

Let  $+_\beta$  be the function that performs the digit string operation of addition for radix  $\beta$ . Correctness of the function  $+_\beta$  means that the following diagram commutes:

$$\begin{array}{ccc} \mathbb{C}^2 & \xrightarrow{+} & \mathbb{C} \\ f_\beta \times f_\beta \uparrow & & f_\beta \uparrow \\ S \times S & \xrightarrow{+_\beta} & S \end{array}$$

We will often write  $s_\beta$  instead of  $f_\beta(s)$  and omit the radix if it equals 10 (the decimal system). The value 1 omits  $\beta$  as  $\beta^0 = 1$  in every positional system. The exception to this is if it is required contextually such as  $1_3 + 1 + 1$  to state that a radix of 3 is being discussed.

If the operator contains the radix it is expected that the operands are digit strings as opposed to numerical values. There is a semantic equivalence, as an example  $10_{+2} 10 \sim 10_2 + 10_2$  because the result of both is a number  $100_2$ . One is a typed operation  $+_2$  on implicit digit strings (untyped), and the other are two typed digit strings  $10_2$  and  $10_2$  which are being acted on by the operator  $+$  of which the implicit type is  $+_2$ . This distinction is used later on when type casting digit strings is introduced. For the rest of the thesis  $s_\beta + s_\beta$  is the preferred syntax. It is a corollary that [Def. 2, p11] can be extended to the other arithmetic operations if and when needed.

### 2.2.2 Negative Systems

We can use any other positive integer for the radix of a system such as binary  $\beta = 2$ , ternary  $\beta = 3$ , quaternary  $\beta = 4$ , and so on. One characteristic of all systems using an integer  $\beta > 1$  is that to represent negative values it requires a prefixed sign, for example  $-58$  or  $-101_3$ . It is possible to represent negative values without a prefixed sign when using a negative integer radix  $\beta < -1$ .

A nomenclature for these systems is the name of their positive sister system with the prefix 'nega', for example negabinary defined as  $\langle -2, \{0, 1\} \rangle$ . Because the index of the digit position is either odd or even this means the resulting value will be either positive or negative. The even digit positions evaluate to the same as the sister system, for example  $2^2 = (-2)^2$ . A direct conversion from binary to negabinary can be done by looking at the odd positions in binary vs negabinary. Here  $k$  is an odd integer:

$$2^k = (-2)^{k+1} + (-2)^k \tag{3}$$

We can take any positive integer in binary, such as  $1010_2 = 2^3 + 2^1 = 10$ , and using [Eq. (3), p11] we can arrive at the negabinary representation for it  $11110_{-2} = (-2)^4 + (-2)^3 + (-2)^2 + (-2)^1 = 10$ . This can be extended to all integer radices with a general definition

**Definition 3.** Conversion from positive to negative system

Let  $p \in \mathbb{N}^{>1}$  be the positive radix and  $q = -p$  be the negative radix.

Let  $k \in \mathbb{O}$ , the set of odd integers.

$$p^k = q^{k+1} + (q - 1)q^k$$

Alongside negative radices another method for representing negative numbers without signage is to use negative valued digits. The balanced ternary system is a great example of this, it is defined as  $\langle 3, \{T, 0, 1\} \rangle$  where  $T = -1$ . Once again a direct conversion can be made from the sister system of ternary. Here we look at the first digit that is not in the balanced ternary digit set, which is  $2_3$ . Note that  $2_3 = 1T_{bt}$ , and compare.

$$2 * 3^n = 3^{n+1} + -1 * 3^n \tag{4}$$

Balanced systems require an odd radix and are defined to be

**Definition 4.** Balanced Digit Sets

$$\langle 2n + 1, \{-n, \dots, -2, -1, 0, 1, 2, \dots, n\} \rangle$$

So for example, quinary can have a balanced digit set but binary would not meet the requirements as it wouldn't be balanced around the digit 0. For systems with an  $\beta > 3$  the usual notation for negative digits is  $\bar{n} = -n$ . Balanced systems have a mirror-reflective property [8] in that a number can be converted to its negative or positive counterpart by changing all the positive digits their negative equivalent and all the negative digits with their positive counterparts. As an example and using the generalised notation  $10\bar{1}_{bt} = 8$  and  $\bar{1}01_{bt} = -8$ .

Representation for Numbers 1 to 5  
(radix notation omitted for readability)

Decimal	1	2	3	4	5
Ternary	1	2	10	11	12
Negaternary	1	2	120	121	122
Balanced Ternary	1	$\bar{1}\bar{1}$	10	11	$1\bar{1}\bar{1}$

Representation for Numbers -1 to -5  
(radix notation omitted for readability)

Decimal	-1	-2	-3	-4	-5
Ternary	-1	-2	-10	-11	-12
Negaternary	12	11	10	22	21
Balanced Ternary	$\bar{1}$	$\bar{1}\bar{1}$	$\bar{1}0$	$\bar{1}\bar{1}$	$\bar{1}\bar{1}\bar{1}$

### 2.2.3 Irrational Systems

Positional systems can have non-integer radices too. The main system looked at in this thesis is called Phinary and is defined  $\langle \varphi, \{0, 1\} \rangle$  where  $\varphi = \frac{1+\sqrt{5}}{2}$  also known as the Golden Ratio. The Golden Ratio is a well known number that is linked to Fibonacci sequences. The Fibonacci sequence is defined as

**Definition 5.** Fibonacci Numbers

Let  $F$  be the inductively defined set of Fibonacci Numbers

$$F_0 = 1, F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

which gives us  $\{1, 1, 2, 3, 5, 8, 13, 21, \dots\}$ . The ratio of consecutive Fibonacci numbers tends to the golden ration  $\lim_{n \rightarrow \infty} \frac{F_{n+1}}{F_n} = \varphi$  and that describes the growth rate of the sequence.

Even though  $\varphi$  is irrational as a numeral system radix it is still able to represent rational values, for example  $101.01_\varphi = 3$ . It also possesses an interesting attribute that there are multiple representations of numbers in phinary, this stems from the fact that  $\varphi^n = \varphi^{n-1} + \varphi^{n-2}$ . This characteristic is discussed in more detail later in the thesis (in section 3.3).

### 2.2.4 Complex Systems

We can again extend our reach further for the radix of a positional system and go into the complex plane  $\beta \in \mathbb{C}$ . In a similar way to negative radices that oscillate between negative and positive, an imaginary radix such as  $i$  will rotate  $90^\circ$  around the complex plane because  $i^0 = 1, i^1 = i, i^2 = -1, i^3 = -i$ .

An imaginary system that is quite well known is Donald Knuth's Quater-Imaginary which is defined  $\langle 2i, \{0, 1, 2, 3\} \rangle$  [9]. Which can represent all the complex integers  $z = a + bi$  where  $a, b \in \mathbb{Z}$ . For its representation of  $z$  it needs no signage of negation, components ( $a$  and  $b$  separated), or an indicator of the imaginary unit.

In 1964 Solomon Khmelnik proposed a collection of complex numeral systems [10]. Among these were  $\langle -1 + i, \{0, 1\} \rangle$  and an interesting system we shall call Khmelnik's binary :  $\langle \frac{-1+\sqrt{7}i}{2}, \{0, 1\} \rangle$ . Khmelnik's binary is both complex and irrational and yet can still represent all the integers in a finite way. It also has an interesting property of not always increasing in representation length, for example  $11100110011_{kb} = 11, 11001100_{kb} = 12, 11001101_{kb} = 13, 11100010110_{kb} = 14$ .

Representation for Rotations of Ten  
(radix notation omitted for readability)

Decimal	10	-10	10i	10 + 10i
Dragonbinary	10	-10	10i	10 + 10i
Quater-Imaginary	10202	302	103010	113212
Khmelnik's Binary	11100110010	101011110		

### 3 Research in the Field of Numeral Systems

The purpose of this section is to put the tool of  $\beta$ -expansion into a wider context and introduce a useful set of numbers for potential numeral systems. While we give definitions in this section the full descriptions can be found in their relative paper citations.

#### 3.1 $f$ -expansions and $\beta$ -expansions

Radix expansions are the use of negative powers of the radix to represent a value. An infinite example being  $0.333\dots$  to represent  $\frac{1}{3}$  in the decimal system.

The study of radix expansions in numeral systems with a non-integer radix begins in 1957 with Renyi introducing the concept of  $f$ -expansions and  $\beta$ -expansions [11], and then with Parry's further work on  $\beta$ -expansions [12] in 1960.

For the definitions below we use  $[x]$  to denote the integer part of  $x$ , and we use  $\{x\}$  to denote the fractional part of  $x$ .

**Definition 6.**  $f$ -expansion

Given a real function  $f$  such that  $f : \mathbb{R}_+ \rightarrow \mathbb{R}_+$  a real number  $x \in \mathbb{R}_+$  can be encoded into an  $f$ -expansion using the following form

Let  $d_i \in \mathbb{Z}_+$ ,

$$x = d_0 + f(d_1 + f(d_2 + \dots)).$$

More precisely this means that the sequence

$$d_0, \quad d_0 + f(d_1), \quad d_0 + f(d_1 + f(d_2)), \quad \dots$$

converges to  $x$ , either in finitely many steps or infinitely converging. In the case that a value  $x$  has an  $f$ -expansion then you can calculate the digits  $d_i$  as such

$$d_0 = [x] \quad \text{and} \quad r_0 = \{x\}$$

then

$$d_{n+1} = [f^{-1}(r_n)]$$



$$r_{n+1} = \left\{ f^{-1}(r_n) \right\}$$

It follows from the above that  $f$  must be bijective for  $f^{-1}$  to be used. Renyi explores the constraints of  $f$  in detail in [11].

A note worthy definition of  $f(x)$  is the case where  $f(x) = \frac{1}{x}$ , here  $f$ -expansion becomes a simple continued fraction. Take, for example, the value  $\frac{355}{113}$ :

$$\begin{aligned} d_0 &= \left[ \frac{355}{113} \right] = 3, & r_0 &= \frac{16}{113} \\ d_1 &= \left[ f^{-1}\left(\frac{16}{113}\right) \right] = 7, & r_1 &= \frac{1}{16} \\ d_2 &= \left[ f^{-1}\left(\frac{1}{16}\right) \right] = 16, & r_2 &= 0 \end{aligned}$$

which can be written in simple continued fraction notation using only the  $d$  as  $[3; 7, 16]$  and can be fully expanded to

$$3 + \frac{1}{7 + \frac{1}{16}}$$

Renyi was looking at the ergodic properties of the interval  $(0, 1)$ , analysing various different definitions of  $f$  looking at the finite and periodic representations. The  $f$ -expansion of use to us is  $f(x) = \frac{x}{\beta}$  which defines  $\beta$ -expansion. Substituting in our new definition for  $f$  we get

$$x = d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \frac{d_3}{\beta^3} + \dots$$

In this area of research positional systems with a non-integer radix are sometimes referred to as  $\beta$ -representation, with  $\beta$ -expansion being a specific use of that form. Of particular interest is the expansion sequence omitting  $d_0$  which is everything past the radix point  $(d_1d_2d_3\dots)$ . While Renyi's original definition of  $\beta$ -expansion had only  $d_0$  for the integer part, later works [13] define  $\beta$ -expansion in its entirety as below.

**Definition 7.**  $\beta$ -expansion

Let  $\beta \in \mathbb{R}$  and  $\beta > 1$ .

Let  $k, i \in \mathbb{Z}$  and  $k \geq i \geq -\infty$ .

A real number  $x \geq 0$  is said to be a  $\beta$ -expansion when in the form

$$x = d_k\beta^k + d_{k-1}\beta^{k-1} + \dots + d_1\beta + d_0 + d_{-1}\beta^{-1} + d_{-2}\beta^{-2} + \dots$$

given by  $d_k = \left[ \frac{x}{\beta^k} \right]$  and  $r_k = \left\{ \frac{x}{\beta^k} \right\}$  and

$$d_i = \left[ \beta r_{i+1} \right]$$

$$r_i = \left\{ \beta r_{i+1} \right\}$$

It should be noted that while  $\beta$ -expansion is a  $\beta$ -representation of some value  $x$ , it is explicitly the representation formed by the method given in the definition. This method is also called the Greedy Algorithm.

There are two useful sets of numbers defined for  $\beta$ -expansions which are the periodic and the finite expansions.

**Definition 8.**  $Per(\beta)$ , the periodic expansions for  $\beta$

Let  $k \in \mathbb{Z}^+$ .

Let  $\beta \in \mathbb{R} > 1$ .

A real number  $x \geq 0$  that is represented using a digit string  $d_n d_{n-1} d_{n-2} \dots d_{n-k} d_{n-k-1} \dots$  such that

$$x = d_k \beta^k + d_{k-1} \beta^{k-1} + d_{k-2} \beta^{k-2} + \dots + d_{k-n} \beta^{k-n} + d_{k-n-1} \beta^{k-n-1} + \dots$$

where after  $k$  digits the representation ends or becomes periodic.  $Per(\beta)$  is the set of all such  $x$ .

subsequently

**Definition 9.**  $Fin(\beta)$ , the finite expansions for  $\beta$

Let  $k \in \mathbb{Z}^+$ .

Let  $\beta \in \mathbb{R} > 1$ .

A real number  $x \geq 0$  can be represented using a digit string  $d_n d_{n-1} d_{n-2} \dots d_{n-k}$  such that

$$x = d_k \beta^k + d_{k-1} \beta^{k-1} + d_{k-2} \beta^{k-2} + \dots + d_{k-n} \beta^{k-n}$$

where after  $k$  digits the representation ends.  $Fin(\beta)$  is the set of all such  $x$ .

It is the case that for different numeral systems there are different finite representations available, for example  $\frac{1}{3}$  has only an infinite representation in decimal but in ternary it has the finite representation  $0.1_3$ . So we can say  $\frac{1}{3} \in Fin(3)$  but  $\frac{1}{3} \notin Fin(10)$ .

It has been shown that when  $\beta$  is an integer that  $Per(\beta) = \mathbb{Q}(\beta) \cap \mathbb{R}_+$  and  $Fin(\beta) = \mathbb{Z}[\beta^{-1}] \cap \mathbb{R}_+$  [13].

## 3.2 Pisot Numbers

This area of research focuses greatly on a class of numbers called Pisot-Vijayaraghavan Numbers, or *Pisot Numbers* for short. They are defined as real algebraic integers greater than 1 where the absolute value of their conjugate is less than 1. We will now define algebraic integers and describe conjugates.

**Definition 10.** Algebraic Integer [14]

Let  $x \in \mathbb{C}$ .

Let  $d_i \in \mathbb{Z}$ .

$$x^n + d_{n-1}x^{n-1} + \dots + d_1x + d_0 = 0$$

If  $x$  solves the polynomial and the polynomial is minimal then  $x$  is called an algebraic integer of degree  $n$ . An algebraic number has a more general form of the polynomial with a leading coefficient of  $d_n$ , so algebraic integers are a special case of algebraic numbers where  $d_n = 1$ .

Conjugates are the alternative solutions to a polynomial, for example  $x^2 - 2 = 0$  has two solutions:  $\sqrt{2}$  and  $-\sqrt{2}$ . These are each other's conjugates.

If we take the polynomial  $x^2 - x - 1 = 0$  the solutions are the golden ratio  $\varphi = \frac{1+\sqrt{5}}{2}$  and its conjugate  $-\varphi^{-1}$ . A more useful form for comparison with its conjugate is  $\varphi = \frac{1}{2} + \frac{\sqrt{5}}{2}$ , and the conjugate value  $-\varphi^{-1} = \frac{1}{2} - \frac{\sqrt{5}}{2}$ . This means that because  $\varphi > 1$  and  $|-\varphi^{-1}| < 1$  then  $\varphi$  is an algebraic integer of degree 2 and a Pisot number.

It has been shown that when  $\beta$  is a Pisot number there will always be an either finite or periodic expansion of 1 such that

$$1 = \sum_{i=1}^{\infty} d_i \beta^{-i} \quad d_i \in \{0, 1, \dots, \lfloor \beta \rfloor\} \quad (5)$$

We refer to [Eq. (5), p17] as a numeral system's *expansions of 1*. It is also known that if  $\beta$  is a Pisot number then  $Per(\beta) = \mathbb{Q}(\beta) \cap \mathbb{R}^+$  and  $\mathbb{Z}^+ \subset Fin(\beta)$  [15].

### 3.3 Standard Form for Numbers

Value representations in positional systems are not unique, for example in decimal  $1 = 0.999\dots$ . With the usual integer radix values integers have unique finite representations.

Looking at phinary we find it holds the inherent property that  $1_\varphi = 0.11_\varphi$ , which is a consequence of  $\varphi^n = \varphi^{n-1} + \varphi^{n-2}$ . This equality comes from the polynomial  $x^2 - x - 1 = 0$  of which the solution is  $x = \varphi$ . Because  $0.11_\varphi$  is the shortest possible finite expansion of the unit value 1, it is trivial to see that it has infinitely many further expansions by simply taking the rightmost digit and performing this representation change again.

$$1_\varphi = 0.11_\varphi = 0.1011_\varphi = 0.101011_\varphi = \dots$$

Because there are many finite representations for numbers in this system we require a standard form, we can get to this rule by reversing the above and looking at the actions of  $011_\varphi \rightarrow 100_\varphi$

1. find part of the digit string,  $d_1d_2d_3$  with any digit followed by digits of at least 1s ( $d_1 \geq 0$ ,  $d_2 > 0$ ,  $d_3 > 0$ ).
2. add one to  $d_1$ , subtract one from  $d_2$ , and subtract one from  $d_3$ .

That is to say we apply a standardisation operation, denoted  $\rightarrow^S$ , on  $011_\varphi$  to produce  $100_\varphi$ . Giving us the rule  $([0, 1, 1], (1, -1, -1))$  where the first part  $[0, 1, 1]$  is a minimum requirements for the digit string values and if those are satisfied then the second part  $(1, -1, -1)$  is added to the relative digits.

Phinary Standardisation of 1001101  
(radix notation omitted for readability)

State	Action	Result	Notes
1001101	search	1001[101]	$d_2 = 0$ , no match.
1001101	search	100[110]1	$d_3 = 0$ , no match.
1001101	search	10[011]01	Match found.
	standardise	10[1]1101	$d_1 + 1$
	standardise	101[0]101	$d_2 - 1$
	standardise	1010[0]01	$d_3 - 1$
1010001	search	1[010]001	$d_3 = 0$ , no match.
1010001	search	[101]0001	$d_2 = 0$ , no match.
1010001	search	[010]10001	$d_3 = 0$ , no match.

The greedy construction we generated in section 3.4.1 for  $10.01_\varphi$  is also known as the minimal form as it uses the least amount of 1s as possible. The standardised form is thus the minimal form.

In the case of phinary the greedy construction will never allow for two (or more) 1s next to each other. This is because in phinary the expansion of 1 means if there are two 1s next to each other then there could have been a 1 before them

instead  $(011_\varphi \rightarrow 100_\varphi)$ . The greedy construction will use the largest value it can working from left to right, meaning two (or more) 1s next to each other is not possible.

### 3.4 Algorithmic Construction of Addition

We will now define the arithmetic operations of addition for the positional system phinary, in which  $\beta = \varphi = \frac{1+\sqrt{5}}{2}$  (the golden ratio) and  $D := \{0, 1\}$ .

As the positional systems use digit alphabets consisting of integer digits, we can form the rule of addition by looking at integer progression in these systems. Specifically looking at the initial unit values, which is a successive progression through the digit alphabet, plus one more increment. Inductively by looking at the progression of the unit values past the digit alphabet max we then know all possible transitions a digit can make during addition.

For example in decimal we would then know that  $0 \rightarrow^{+1} 1 \rightarrow^{+1} 2 \dots$  and so on and also that  $9 \rightarrow^{+1} 10$ .

To construct the rules we need the first  $|D|$  numbers in the system where  $|D|$  is the cardinality of the digit alphabet. Positional numeral system is a power series as seen in [Def. 1, p10], translating numbers to a certain positional system simply requires performing a greedy construction. We are looking at positive unit values larger than 1, the below algorithm will work for values of  $num \geq 1$ :

---

**Algorithm 1:** The greedy algorithm

---

```

let numString = ""
let index = Integer(log(num) / log(r))
while num != 0 do
    power =  $\beta^{index}$ 
    units = floor(num / power)
    num = num - (units * power)
    index = index - 1
    numString.append(units)

```

---

This allows us to construct numbers in a positional system without needing to know succession rules (as that is what we are trying to derive).

Computationally, there is always the problem of precision but in phinary for the numbers less than ten we can get away with truncating after four digits past the radix point.

#### 3.4.1 Overflow

Arithmetic carry of numeral systems using a whole number radix is a trivial property of powers. For  $\beta \in \mathbb{N}$  and  $n \in \mathbb{Z}$  the equality is

$$\underbrace{\beta^n + \dots + \beta^n}_{\beta \text{ times}} = \beta^{n+1}$$

Examples:

- $1_3 + 1_3 + 1_3 = 10_3$
- $300_4 + 100_4 = 1000_4$
- $10 \cdot 0.001 = 0.01$

The third example is true in all positional systems as it is an algebraic fact that  $x^n x = x^{n+1}$ .

For a systems such as [Def. 1, p10] overflow occurs when a unit value exceeds the digits of its alphabet. That is to say when an operation such as addition increases the value of a unit past the representable digits e.g.  $2_2$  exceeds the alphabet of binary which is  $\{0, 1\}$ . The digit alphabet of phinary is  $\{0, 1\}$ , with a cardinality of 2. Below are results of calling the greedy construction algorithm for 0, 1, and 2 to generate their representation in phinary:

*Greedy*(0,  $\varphi$ ) returns 0  
*Greedy*(1,  $\varphi$ ) returns 1  
*Greedy*(2,  $\varphi$ ) returns 10.01

Note here that  $10.01_\varphi$  means  $\varphi^1 + \varphi^{-2} = 2$ . Binary has an overflow that expands to the left only  $1_2 + 1 = 10_2$  whereas phinary has an overflow that expands bidirectionally  $1_\varphi + 1 = 10.01_\varphi$ . If we look at the pointwise difference between the result of  $1_\varphi + 1$  which is  $2_\varphi$  and the valid version of  $2_\varphi$  as a phinary number  $10.01_\varphi$  we can see the actions of the overflow rule:

Each of these four actions are starting from the position of the digit that is too high (requires overflow)

1. move left and +1                       $2 \rightarrow 12$
2. do not move and  $-2$                        $12 \rightarrow 10$
3. move right and +0                      (no effect\*)
4. move right twice and +1               $10 \rightarrow 10.01$

*\*we keep this in so we have fully described the neighbourhood. Later in other related numeral systems this neighbour is changed along with the rest.*

Each of these steps is performed from the position of the overflow. We can package these rules into a set of pairs  $\{(1, 1), (0, -2), (-1, 0), (-2, 1)\}$ . The pairs are actions of the rule and are in the form (shift, add). We shall refer to this overflow operation as  $\rightarrow^O$ .

It should be pointed out that the result of overflow  $2_\varphi \rightarrow^O 10.01_\varphi$  can be achieved by a combination of standardisation and inverse standardisation. If we recall standardisation to be  $011 \rightarrow^S 100$  and denote inverse standardisation as  $\rightarrow^{\frac{1}{S}}$ :  $100 \rightarrow^{\frac{1}{S}} 011$  then by allowing free choice of where we apply the standardisation

$$2_\varphi \rightarrow^{\frac{1}{S}} 1.11_\varphi \rightarrow^S 10.01_\varphi$$

equates



$$2_\varphi \rightarrow^O 10.01_\varphi$$

We will do an example of  $2_\varphi$  padded with zeros to help show the procedure.

Phinary Overflow of  $2_\varphi \rightarrow 10.01_\varphi$

State	Action	Result
$02.00_\varphi$	$(+1, +1)$	$[1]2.00_\varphi$
$12.00_\varphi$	$(+0, -2)$	$1[0].00_\varphi$
$10.00_\varphi$	$(-1, +0)$	$10.[0]0_\varphi$
$10.00_\varphi$	$(-2, +1)$	$10.0[1]_\varphi$

The rule for overflow of a unit can be applied to a digit string on each digit however many times is required. Because of this addition between two positive numbers is also defined.

**Lemma 1.** Phinary Addition  $a +_\varphi b$

Let  $a, b \in L_\varphi$  be a digit string valid in the language of  $L_\varphi$ .

We define digit string  $c$  to be the pointwise addition of  $a$  and  $b$  such that

$$c_i = a_i + b_i$$

Then we apply an overflow check on each digit of  $c$  to check if  $c_i \geq 2$ , if it is then the overflow rule is applied and the four actions are carried out. This procedure repeats until  $c \in L_\varphi$ . Once no digit  $\geq 2$  is left, the computation of addition is complete.  $\square$

It is later shown in [Theorem 1, p23] that addition for phinary does terminate.

### 3.4.2 Normalisation

Now we have the above two rules we can normalise any string of digits into a valid (using only digits from the system's alphabet) and standardised representation. We will apply [Lemma 1, p21] and do pointwise addition of  $10.01_\varphi = 2$  and  $101.01_\varphi = 4$ , the result when evaluated should be an invalid form that is equal to 6:

$$10.01_\varphi + 101.01_\varphi = 111.02_\varphi$$

then through a combination of overflow,  $\rightarrow^O$ , and standardisation,  $\rightarrow^S$ , we will arrive at the valid standard form

$$111.02 \rightarrow^O 111.1001 \rightarrow^S 120.0001 \rightarrow^O 200.1001 \rightarrow^O 1001.1001 \rightarrow^S 1010.0001$$

Leaving us with  $1010.0001_\varphi = 6$ . As one can see, the normalisation procedure moves around in the number going back and forth, we want to make sure that these rules will always terminate when being applied. The following proof does this.

### 3.5 Mathematical Construction of Addition

The golden ratio is the first instance of a larger family of numbers called the metallic means (golden ratio, silver ratio, bronze ratio, etc). These numbers satisfy the homogeneous continued fraction  $[b; b, b, b, \dots]$  where  $b \in \mathbb{N}$ , and they are solutions to the polynomial

$$\beta^2 - b\beta - 1 = 0. \quad (6)$$

The  $b^{\text{th}}$  metallic mean can be defined as

$$\beta_b = \frac{b + \sqrt{b^2 + 4}}{2} \quad (7)$$

So we can see that  $\beta_1 = \frac{1+\sqrt{5}}{2}$ ,  $x_2 = 1 + \sqrt{2}$ , and so on. Remarkably when used as the radix of numeral systems the metallic means can represent all the integers finitely.

In the following we fix  $b$  and study the  $b^{\text{th}}$  metallic numeral systems. The numeral systems for the  $b^{\text{th}}$  metallic mean use the digits  $[0, \dots, b]$ . Substitution rules for standardisation and addition in metallic numeral systems can be defined as below.

These rules are applied to a digit string with some value (encoded by metallic system  $b$ ) and produce another digit string changing the form but not the value.

There are required conditions and anywhere in the digit string that those conditions are met the rule is applied to that/those digits and the surrounding neighbours.

Let  $a = b - 1$  and  $c = b + 1$ ,

$$\begin{aligned} (r1) \quad klm &\rightarrow (k+1)(l-b)(m-1) && \text{if } l \geq b \text{ and } m \geq 1 \\ (r2) \quad klmn &\rightarrow (k+1)(l-c)(m+a)(n+1) && \text{if } l \geq c \end{aligned}$$

The correctness of these rules follows from the fact that the metallic mean  $\beta_b$  satisfies [Eq. (6), p22] which can be rearranged into the rules. By correctness we mean an application of the rule will not change the value of the number just the form. The validity of (r1) and (r2) follows directly from the following equations (e1) and (e2).

$$\begin{aligned} (e1) \quad \beta^2 &= b\beta + 1 \\ (e2) \quad c\beta^2 &= \beta^3 + a\beta + 1 \end{aligned}$$

It is trivial to derive (e1) from [Eq. (6), p22]. We can then use the (e1) to prove (e2) by rearranging the left side of the equation into the right side. We expand  $a = b - 1$  and  $c = b + 1$  for simplicity.

First we expand (e2) and put it in a general form by multiplying both sides by  $\beta^{-2}$

$$(b+1)\beta^k = \beta^{k+1} + (b-1)\beta^{k-1} + \beta^{k-2}$$

The actions performed below are using (e1) by expanding one  $\beta^k$  term into  $b\beta^{k-1} + \beta^{k-2}$ , then taking  $b\beta^k$  and one  $\beta^{k-1}$  using (e1) to reduce them to  $\beta^{k+1}$ .

$$\begin{aligned}(b+1)\beta^k &= b\beta^k + b\beta^{k-1} + \beta^{k-2} \\ &= \beta^{k+1} + (b-1)\beta^{k-1} + \beta^{k-2}\end{aligned}$$

Therefore by the correctness of (e1) and (e2) we know that applying (r1) or (r2) will not change the value of a digit string, just the form.

### 3.5.1 Proof of Termination

**Theorem 1.** Any normalisation sequence using rules (r1) and (r2) terminates.

Proof.

By a word we mean in the following sequence

$$w = \dots w_2 w_1 w_0 . w_{-1} w_{-2} \dots \in \mathbb{N}^{\mathbb{Z}}$$

such that  $w_i = 0$  for almost all  $i \in \mathbb{Z}$ .

We let

$$S(w) = \sum_{i \in \mathbb{Z}} w_i$$

be the sum of all (non-zero) digits in  $w$ . As remarked above, rule (r1) reduces the sum of digits while rule (r2) preserves it:

$$\text{if } w \xrightarrow{(r1)} w', \text{ then } S(w') = S(w) - b \quad (8)$$

$$\text{if } w \xrightarrow{(r2)} w', \text{ then } S(w') = S(w) \quad (9)$$

Therefore, every normalisation sequence can contain only finitely many applications of rule (r1).

We now look to prove that every normalisation sequence using only rule (r2) terminates.

Considering only words  $w$  with at least one non-zero digit, i.e.  $S(w) > 0$ , we define the left and right end of a word  $w$  as

$$L(w) = \max\{i \in \mathbb{Z} | w_i > 0\}$$

$$R(w) = \min\{i \in \mathbb{Z} | w_i > 0\}$$

and its *diameter* as

$$D(w) = L(w) - R(w)$$

Recall (r2):

$$(r2) \quad klmn \rightarrow (k+1)(l-c)(m+a)(n+1) \quad \text{if } l \geq c$$

Because (r2) is applied based on digit  $l$  and it increments the neighbouring digits  $k$  and  $n$  by 1 it follows,

$$\text{if } w \xrightarrow{(2)} w', \text{ then } L(w') \geq L(w) \geq R(w) \geq R(w') \quad (10)$$

$$\text{if } w \xrightarrow{(2)} w', \text{ then } D(w') \geq D(w) \quad (11)$$

We define the *centre of gravity* of a word  $w$  as

$$G(w) = \frac{g(w)}{S(w)}$$

where

$$g(w) = \sum_{i \in \mathbb{Z}} i * w_i$$

It follows that

$$L(w) \geq G(w) \geq R(w) \quad (12)$$

Because (r2) is conditional on digit  $l$  it has the local neighbourhood indexing of

$$k_{i+1}l_i m_{i-1} n_{i-2}$$

The sum of the indices of digits  $k$  and  $n$  are 1 less than the index of digit  $l$  and the index of digit  $m$  is 1 less than the index of digit  $l$ . Because  $k$  and  $n$  are incremented by a fixed value of 1 and  $m$  is incremented by a fixed value of  $(b-1)$  we know that,

$$\text{if } w \xrightarrow{(2)} w', \text{ then } G(w') = G(w) - b \frac{1}{S(w)} \quad (13)$$

We can solve  $G(w) - G(w') = b \frac{1}{S(w)}$  by substituting the  $klmn$  indices into  $g$  and showing that  $g(w) - g(w') = b$  (all other indices cancel out in the subtraction, only  $klmn$  can change)

$$\begin{aligned} g(w) - g(w') &= (i+1)k + il + (i-1)m + (i-2)n \\ &\quad - (i+1)(k+1) + i(l-b) + (i-1)(m+(b-1)) + (i-2)(n+1) \end{aligned}$$

To break it up we will reduce the  $k, l, m$  and  $n$  terms for  $g(w) - g(w')$ , starting with the  $k$  term

$$\begin{aligned} (i+1)k - (i+1)(k+1) &= ki - ki - i + k - k - 1 \\ &= -i - 1 \end{aligned}$$

and the  $l$  term

$$\begin{aligned} il - i(l - (b + 1)) &= il - il + ib + i \\ &= ib + i \end{aligned}$$

the  $m$  term

$$\begin{aligned} (i - 1)m - (i - 1)(m + (b - 1)) &= m - m + im - im - bi + i + b - 1 \\ &= i - ib + b - 1 \end{aligned}$$

and the  $n$  term

$$\begin{aligned} n(i - 2) - (n + 1)(i - 2) &= ni - ni - n2 - n2 - i + 2 \\ &= -i + 2 \end{aligned}$$

after which we can substitute it back into the full equation

$$\begin{aligned} g(w) - g(w') &= (-i - 1) + (ib + i) + (i - bi + b - 1) + (-i + 2) \\ &= b \end{aligned}$$

We are now able to arrive at a lower bound for the diameter of a word after  $k$  applications of  $(r2)$ , which we denote as  $w \rightarrow^{(2),k} w'$ .

After  $k$  applicatios of  $(r2)$ , given [Eq. (13), p24]

$$G(w) = G(w') + bk \frac{1}{S(w)} \quad (14)$$

knowing that [Eq. (10), p24] holds, we can say

$$G(w') + bk \frac{1}{S(w)} \geq R(w') + bk \frac{1}{S(w)} \quad (15)$$

$$L(w') - R(w') \geq bk \frac{1}{S(w)} \quad (16)$$

therefore

$$\text{if } w \rightarrow^{(2),k} w' \text{ then } D(w') \geq bk \frac{1}{S(w)} \quad (17)$$

By [Eq. (13), p24] we know that gravity is reduced by a fixed amount, and that [Eq. (12), p24] holds, therefore we can prove  $(r2)$  terminates by proving  $R(w')$  has an upper bound. Which we will do by finding an upper bound for  $D(w')$ .

We say a word  $w$  **has only small gaps** if any 'inner' sequence of consecutive zeros has length  $\leq 2$ . That is to say there is no  $i$  between  $L(w)$  and  $R(w)$  such that  $\forall i(w_i = 0 \wedge w_{i+1} = 0 \wedge w_{i+2} = 0)$ .

For applications of  $(r2)$  we can see that any zero introduced by  $l - c$ , and any zero preserved by  $m + a$  (in the case of  $a = 0$ ), there is a surrounding small

gap created by the additive actions of  $k + 1$  and  $n + 1$ . As  $k$  and  $n$  create an inner sequence of 2 digits, which is a small gap, we can say

$$\text{if } w \text{ has only small gaps and } w \xrightarrow{(2)} w', \text{ then } w' \text{ only has small gaps} \quad (18)$$

Furthermore, as the maximum length of a word with only small gaps is achieved by the pattern  $X00X00X00\dots X$  and the largest possible  $S(w)$  is achieved when all non-zero digits are  $b$ , we have

$$\text{if } w \text{ has only small gaps then } D(w) < \frac{3S(w)}{b} \quad (19)$$

A limit for the amount of applications of (r2) can be found by solving  $\frac{bk}{S(w)} < \frac{3S(w)}{b}$  for  $k$ , therefore a normalisation sequence starting with a word  $w$  with only small gaps that uses only (r2) must have a length less than

$$\frac{3S(w)^2}{b^2} \quad (20)$$

We order pointwise, i.e..

$$w \leq v \iff \forall i \in \mathbb{Z}(w(i) \leq v(i))$$

The following are trivial observations:

$$\text{if } w \leq v \text{ then } D(w) \leq D(v) \quad (21)$$

If  $w \xrightarrow{(2)} w'$  and  $w \leq v$ , then

$$\text{there exists } v' \text{ such that } v \xrightarrow{(2)} v' \text{ and } w' \leq v' \quad (22)$$

Given a word  $w$ , there is pointwise larger word  $v$  with small gaps such that

$$S(v) \leq S(w) + \frac{D(w)}{3} \quad (23)$$

since it is enough to replace at most  $\frac{D(w)}{3}$  zeros with 1. For example if we had  $w = 2000000$  then we fill the large gap and have  $v = 2001001$ .

For every reduction sequence starting with  $w$ , there is an equally long reduction sequence for  $v$ .

It is the case that  $v$  is bounded by [Eq. (20), p26], from here we can use [Eq. (23), p26] to speak about  $w$ .

A normalisation sequence starting with word  $w$  that uses only (r2) must have a length less than

$$\frac{3(S(w) + \frac{D(w)}{3})^2}{b^2} \quad (24)$$

Therefore, the theorem is proven, even with an explicit bound on lengths of normalisation sequences.  $\square$



### 3.6 A Requirement for Radix Expansion

A possible question to arise is does representation of the natural numbers always require expansion past the radix point? Below we prove that for all the metallic systems a radix expansion is required to represent any rational number.

**Theorem 2.** Let  $p, q \in \mathbb{Z}^+$ . When the solution to  $\beta^2 - p\beta - q = 0$  is irrational then the numeral system using radix  $\beta$  cannot express any rational number without the use of radix expansion.

Proof.

The representation of a value without radix expansion is  $x = d_n\beta^n + d_{n-1}\beta^{n-1} + \dots + d_1\beta^1 + d_0\beta^0$  where  $d \in \{0, 1, \dots, \lfloor \beta \rfloor\}$ . We can derive the standardisation rule by rearranging the root polynomial to  $\beta^n = p\beta^{n-1} + q\beta^{n-2}$ . We can then use this rule, in reverse, to reduce the representation of a value to  $x = e_0\beta + e_1$  where  $e_i \in \mathbb{Z}^+$ . Thus as  $\beta \in \mathbb{R} \setminus \mathbb{Q}$  it is the case that  $x$  can only be irrational.  $\square$

An example of the standardisation reduction is shown below using the second metallic system, which has the silver ratio, defined as  $\delta_s = \sqrt{2} + 1$ , as its radix. In the system (discussed in section 2) the standardisation rule is  $\delta_s^n = 2\delta_s^{n-1} + \delta_s^{n-2}$ . We will apply this to an arbitrary value in numeral representation:

$$1111_{\delta_s} \xrightarrow{\text{reduce}} 321_{\delta_s} \xrightarrow{\text{reduce}} 242_{\delta_s} \xrightarrow{\text{reduce}} 163_{\delta_s} \xrightarrow{\text{reduce}} 84_{\delta_s} \quad (25)$$

leaving us with  $e_0 = 8$  and  $e_1 = 4$ . Note that  $1111_{\delta_s} = 84_{\delta_s} = 12 + 8\sqrt{2}$ .

### 3.7 Numbers and Machines

Computations in programming languages use type systems which are limited in what they can represent. For example in the programming language C the 'float' type uses an IEEE-754 encoding to better maximise representation and allowing a far more useful range than fixed-point representation [Def. 11, p28]. We can look at the ranges of the type,  $\tau$ , being used by the machine.

The most simple numerical encoding would be fixed-point encoding, where there are a fixed amount of digits in the number and it is treated like a positional number system.

**Definition 11.** Fixed-point Encoding

Let  $\beta \in \mathbb{C}$ ,  $|\beta| > 1$  be the radix of the system.

Let the digits  $d_i \in \{0, 1, \dots, \lfloor |\beta| \rfloor - 1\}$ .

Let  $n, m \in \mathbb{Z}^+$  be the fixed-point constraints.

The fixed-point encoding of value  $x$  as digits  $d_n \dots d_2 d_1 d_0 . d_{-1} d_{-2} \dots d_{-m}$  is

$$x = d_n * \beta^n + \dots + d_2 * \beta^2 + d_1 * \beta^1 + d_0 * \beta^0 + d_{-1} * \beta^{-1} + d_{-2} * \beta^{-2} + \dots + d_{-m} * \beta^{-m}$$

The range is a set that consists of all the numbers that can be represented by some arbitrary string  $s \in L_\tau$  (where  $L_\tau$  is the language of valid strings in type  $\tau$ ).

As an example we can construct a fixed-point representation where  $n = 2, m = 2, \beta = 2$ . In such a system the representations 01.00, 10.11, 10.01 are valid but 20.00 wouldn't be valid as  $\beta = 2$  doesn't allow for the digit 2 to appear. Another invalid representation would be 010.01 which has 3 digits left of the radix point and  $n = 2$ .

**Definition 12.** Machine Numbers

Let  $\tau$  be a fixed-point representation.

Let  $s$  be a digit string ranging across  $L_\tau$  the set of valid string in the language of  $\tau$ .

The machine numbers for type  $\tau$  are defined

$$\mathbb{M}_\tau := \{m \in \mathbb{C} : \exists s(\tau(s) \mapsto m)\}$$

For all types using an integer radix  $\beta \in \mathbb{N}^{>1}$  it is the case that irrational numbers such as  $\pi$  are not in  $\mathbb{M}_\tau$ . For situations where close approximations are acceptable we can use a threshold  $\epsilon$  to collect values that are close to values such as  $\pi$ .

**Definition 13.** Approximation Set

Let  $\tau$  be an arbitrary fixed-point representation.

Let  $\epsilon \in \mathbb{R}$  be the approximation error bound.

Let  $x \in \mathbb{C}$  be the value being approximated.

$$\mathbb{M}_\tau \supset \Gamma_x^\tau := \{\gamma \in \mathbb{M}_\tau : |\gamma - x| \leq \epsilon\}$$

The cardinality of  $\Gamma_x^\tau$  will differ depending on the fixed-point encoding  $\tau$ , the value being approximated  $x$ , and the error bound  $\epsilon$ . We can refer to  $|\Gamma_x^\tau|$  as its representation density. [Def. 13, p28] contains representable values close to  $x$  and possibly also  $x$ .

We now look at when  $\tau$  is [Def. 11, p28] with a limited digit set. It will have a fixed length and a fixed position for the radix point. The constraints are  $n, m$ , and the digit set which we will fix to be  $\{0, 1\}$ . This means the only variable is  $\beta$ .

**Definition 14.** Optimal Radix for Fixed-point Representation

Let  $\tau(\beta)$  denote a fixed-point representation with a fixed  $n, m \in \mathbb{Z}^+$ , fixed digit set  $\{0, 1\}$ , and radix  $1 > \beta \geq 2$ .

Let values  $x_i$  and radix  $\rho$  range over  $\mathbb{C}$ .

For every choice of finitely many values  $x_1, \dots, x_k$  there is an optimal encoding  $\tau$  such that it presents the largest joint representation density, that is,

$$\forall \rho (|\bigcup_{i=1}^k \Gamma_{x_i}^{\tau(\beta)}| \geq |\bigcup_{i=1}^k \Gamma_{x_i}^{\tau(\rho)}|)$$

A simple example of this would be to set the error bound to  $\epsilon = 0$  (only accepting exact approximations), then we can see that if we want to represent the number 2 with 4bit fixed-point type ( $n = 2, m = 2$ ) then the  $\tau$  using  $\beta = 2$  can do so with the representation 10.00. We also see that phinary  $\beta = \varphi$  has a valid representation 10.01 as an exact approximation of the value 2. We can go further and increase the amount of  $x$  we want to represent and start to see differing optimality. Below we look at approximation sets for representing the values 2 and  $\varphi$ .

$$|\Gamma_2^{\tau(\varphi)} \cup \Gamma_\varphi^{\tau(\varphi)}| \geq |\Gamma_2^{\tau(2)} \cup \Gamma_\varphi^{\tau(2)}|$$

Because

$$|\{10.01, 10.00\}| \geq |\{10.00\}|$$

Thus if the aim is to represent both 2 and  $\varphi$  then for the above constraints of  $\tau$  it is the case that  $\beta = \varphi$  is more optimal than  $\beta = 2$ .

In this section we have put forward an example of value representation in computing (fixed-point representation). We defined a set of numbers called machine numbers which are all possibly representable numbers for a fixed-point encoding  $\tau$ . We then define approximation sets as the subset of machine numbers for some fixed-point encoding  $\tau$  where the members are close to some chosen value of  $x$  (the approximation value). This allows us to measure encodings for their ability to represent specific values or values close to them. The example shows us how we can compare two encodings,  $\tau = \varphi$  and  $\tau = 2$  to see that  $\tau = \varphi$  can represent the two values we wanted, making it the better choice.

## 4 Programming with Numeral Systems

In this section we will develop a library that extends Common Lisp's type system to house user defined numeral systems for dynamic type casting of strings as numbers. Because Common Lisp is not the most well known language we will preface this work with an introduction to and overview of Common Lisp.

### 4.1 A Common Lisp Primer

This section is for readers unfamiliar with the Lisp family of languages (specifically Common Lisp). Readers that have a familiarity with lisps might want to skip this chapter then come back to it as a reference guide if needed. Types/-variables in Common Lisp are rather different than they are in more mainstream programming languages, here we will go over the important differences.

#### 4.1.1 Types

Some of the basic types in Lisp are *Numbers*, *Characters*, *Symbols*, *Lists*, and *Functions*. Symbol is a type not often seen in other languages and acts as a name for things. Symbols are made up of five different slots called *cells*, which are used for assigning data. The cells are:

1. Value Cell
2. Function Cell
3. Property List
4. Print Name
5. Package

The different cells are activated contextually, a good example of this would be Lisp's *s-expression*. When the Lisp reader sees a list such as (A B C) then it interprets this as what is called an s-expression. Evaluating it means calling the function cell of the first symbol in the list, *A*, and passing the value cells of the remaining symbols, *B* and *C*, as arguments. The last three cells are usually not actively used by the user but the first two are set with macros like `defvar`, `defun`, etc.

Checking types in Lisp is akin to checking if a value would be valid as that type. If we check the type of a symbol what we are actually doing is evaluating the value cell and checking the type of the value. New types can be created using `deftype`, this macro registers a symbol as a type-specifier (no cell binding is performed). The type-specifier is associated with some code that contains predicate rules for the type:

```
(deftype even ()  
  '(and integer (satisfies evenp)))
```

```
(typep 4 'even) ; returns true -> t
(typep 5 'even) ; returns false -> nil
```

We mentioned that **deftype** is a macro. Macros and functions differ in that the parameters passed to a macro are not evaluated. This is because their job is to expand into actual code before executing. In the above code *even* is a currently unbound symbol and would throw an exception if it was evaluated. **deftype** receives the symbol name itself as a parameter without using the value cell. Later the function *typep* requires a special syntax to get even. The code is quoted (with one ' on its left) this is used to say "don't evaluate this form". Inside the unevaluated code we see another type-specifier, *integer*, and the macro *satisfies* which is passed the symbol *evenp*. The function **evenp** checks if a number is even. The function **typep** takes a value and a quoted type-specifier, it must be quoted because **typep** is a function not a macro and the value of even is unbound (there is nothing bound to the value cell even though it is a type-specifier).

#### 4.1.2 Variables

Variables don't have types, values have types. As we mentioned before, variables are just containers for values, they don't hold any meta-information about what is or can be held. In the below example we use the macro **defvar** to define a variable and assign it to a symbol:

```
(defvar foo 5)
```

A variable is created containing 5 then it is bound to the value cell of the symbol *foo*. Because there are multiple cells that a symbol can bind we can perform the following variable and function definitions:

```
(defvar foo 5)
(defun foo (foo)
  (* foo foo))

(foo foo) ; returns 25
(foo 4)   ; returns 16
```

The macro **defun** will bind the function body along with a local scoping of parameters to the function cell of *foo*. At this point we have the value cell and function cell of *foo* bound, and when the function of *foo* is called a new local binding of *foo* occurs (as function *foo* takes one argument 'foo').

## 4.2 System Interpreted Numbers

System Interpreted Numbers (Sin) is a language concept where numeric types (such as int, float, double, etc) don't exist, instead strings are given numeric meaning by being associated to a user-defined numeral system type. To do this there is a core mechanism for defining numeral systems which generates new types. The new type is accompanied by a family of functions, among these is the evaluation function that converts a string to a decimal value. With such a framework we can define numeral systems and then bestow value onto strings statically (or dynamically with the 'active system' mechanism discussed later).

The full language design for Sin is an ongoing project by the author. At the time this thesis began it was entirely conceptual with an incomplete plan to be a compiled language written in C++. The work in this thesis required implementing a subset of the planned Sin functionality in the form of a common lisp library. All implementations of concepts were done during the course of this project. This is what we discuss moving forward and what was used in later testing. Code developed for this project can be found at [16].

### 4.2.1 defnumsys

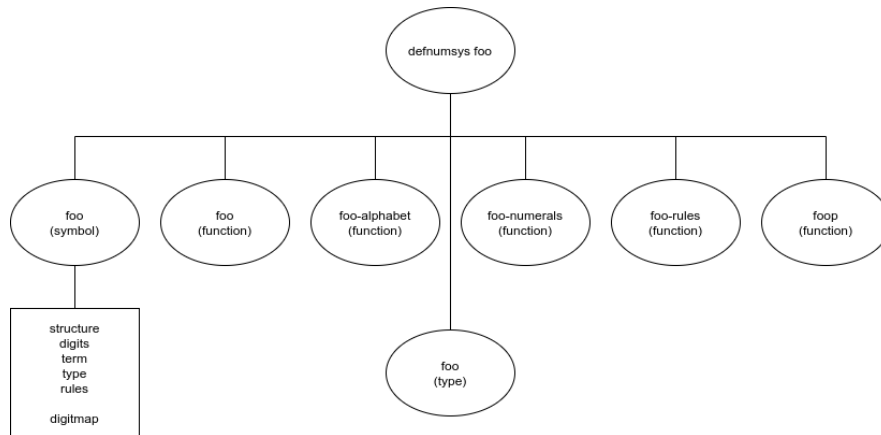


Figure 4: When `defnumsys` is called it generates a symbol, five functions, and a type. The symbol has information added to its property-list. The naming is based on the name given as an argument to `defnumsys`, here the name 'foo' is used so function names like 'foo-alphabet' and 'foop' are generated.

To facilitate treating string encoding like numeral systems we have written a macro that generates an evaluation function and a type-specifier based on three main code inputs. These are `structure`, `digits`, and `term`. The macro allows for user defined variables to be given too (such that they can be used in the main code inputs) and also allows for a supertype to be given where all inputs can be derived. The code bodies given to `defnumsys` have contextually

predefined variables `+s+` (the string encoding), `+d+` (the current digit), and `+n+` (the length of `+s+`).

The syntax for the `defnumsys` macro is shown below. Note that the arguments inside `[` and `]` are key pairs so 'body' is the code expected to be paired with key `:structure` and so on. They are also all optional, only name and supertype are required.

#### Documentation 1 (`defnumsys`)

```
defnumsys name supertypes [(structure body) (digits body) (term  
body) ...]
```

<code>name</code>	<i>The name of the numeral system.</i>
<code>supertypes</code>	<i>An empty list or list of numeral systems.</i>
<code>structure</code>	<i>Code required to evaluate terms.</i>
<code>digits</code>	<i>Code that returns an associated list of char/value.</i>
<code>term</code>	<i>Code that each digit is expanded to before evaluation.</i>
<code>...</code>	<i>Any user keywords can be defined for the above code bodies.</i>

The pseudocode below shows the step by step actions of `defnumsys`:

---

**Algorithm 2:** An overview of the `defnumsys` macro

---

```
defmacro defnumsys (name supertype [[keyword value]...]):  
  if supertype then  
    if structure empty then set to supertype structure;  
    if digits empty then set to supertype digits;  
    if term empty then set to supertype term;  
  if keywords undefined {structure, digits, term} then  
    ⊥ return error  
  Edit term to access digits in place of the symbol +d+;  
  Edit structure by replacing the symbol term with the term code;  
  Add structure code to name's property cell;  
  Add digits code to name's property cell;  
  Add term code to name's property cell;  
  Add symbol name as type in property cell;  
  evaluate defvar (name name):  
    ⊥ Bind value cell to the symbol name;  
  evaluate defun (name value):  
    ⊥ Bind function cell to call the code in structure on value;  
  evaluate defun (namep value):  
    foreach letter in value do  
      ⊥ Check it is in digitmap;  
      ⊥ Check each digit appears a valid amount of times;  
    return validity of value  
  return deftype (name):  
    ⊥ Type satisfies string and namep
```

---

In the next example we are going to define a generalised system that represents the positional numeral system. Then we will define the binary numeral system using our general supertype.

```
1 (defnumsys positional ())  
2   :structure (sum-series +s+ term)  
3   :digits (radix-alphabet r)  
4   :term (* (expt r n) +d+)  
5   :r 0)  
6  
7 (defnumsys binary (positional) :r 2)
```

In line 1 we give this new system a name, 'positional', and give an empty list for the supertype. Next on line 2 we use a function called `sum-series` as our structure value. `sum-series` takes a string, `+s+`, and mimics a positional numeral system using `term` as the definition of what each term in the sum series means. Line 3 defines the digits which is the alphabet of valid characters. `radix-alphabet` is another premade Sin library function that will return a digit list based on `r` which is the radix for the system. The format of digit lists is a list of character and value pairings i.e., (`list (#\A 10) (#\2 4)`) which would



be an alphabet where only A and 2 are valid characters and would equal 10 and 4 respectively. Line 4 defines the term and uses the inbuilt variable `+d+`. As the `sum-series` code moves across the string `+s+` it takes the current character and uses the `digits` list to get the numeric value of it and evaluates the term with `+d+` being the digit value. The value `n` in the `term` code is a variable used in `sum-series` relating to the position of the character in the string. Finally line 5 is where we have a user defined value, `:r 0`. This is the radix of our system, an important value we use in the term and structure code and one that defines our digit alphabet. An exact definition of `sum-series` and `radix-alphabet` is as follows:

```

1 (defmacro sum-series (str body)
2   `(let* ((test (position #\. ,str))
3           (terms (if test (remove #\. ,str) ,str))
4           (offset (if test (- (length terms) test) 0))
5           )
6     (loop for +n+ from (1- (length terms)) downto 0
7           sum
8           (let ((d (char terms (- (length terms) +n+ 1)))
9                 (n (- +n+ offset)))
10              ,body))))
11
12 (defun absolute (value)
13   (if (complex value)
14       (abs value)
15       (abs (float-r value))))
16
17 ; this is our function for defining the alphabet
18 ; used by some radix in a positional system
19 ; if complex return self else return float-r self
20 (defun radix-alphabet (radix)
21   (let ((active (gen-range 0 (1- (ceiling (absolute
22     radix)))) #'anu)
23         (passive #2<(#\. nil))) ; radix point can only
24         appear < 2 times
25     (cons passive active)))

```

A positional numeral system with a radix of zero is not very useful (and has an empty digit alphabet so no strings are valid). Line 7 is where we make a useful type, `binary`. To do this all we need do is add 'positional' to our supertypes list and redefine the radix keyvalue. The type validity function that is generated, `binaryp`, will check strings it is given and make sure they only have digits 0 and 1 in them.

## 4.2.2 Alphabets

The list of character-value pairs that makes up an alphabet list may also contain constraints on the character. For example `(radix-alphabet 2)` actually produces `(list (#\ . nil #\< 2) (#\0 0) (#\1 1))`. The extra values mean there must be less than 2 instances in the string for it to be a valid word in the language.

Sin has a macro shorthand for defining pairs with extra info: `#N*(character value)` here `*`  $\in \{<, >, =\}$  and `N` is the constraint value. The radix point example from above would be `#2<(#\ . nil)`.

## 4.2.3 Active Systems

We can define a static 'sin-value' as a variable initialised with a string and an associated numeral system type, and a dynamic 'sin-value' to be a string value used during an active numeral system. The latter, active systems, are numeral system types that have been put onto a global priority stack. When an operation is done on a string, if it has no type assigned, then the global stack of active systems is checked and it treats it as if it were the first matched system.

In the below example two systems are activated - binary and decimal. As binary is the primary active system and results from operations will be returned in binary but each string will be matched to the first successful active system. The lisp inputs are shown on lines starting with '>' and the result on the next line:

```
1 >(activate bin dec)
2 >(sin-truevalue "11")      ; activates bin
3 3
4 >(sin-truevalue "12")      ; activates dec
5 12
6 >(dec "11")                ; directly using dec
7 11
```

We can see here this introduced an ambiguity with 11 being valid in both binary and decimal, so it's not recommended to activate multiple overlapping systems, instead this works better with unique systems such as `(activate dec rmn)` where 'rmn' here would be the Roman Numeral system. The strength in active systems is in allowing the user to write generic string-ambiguous functions then run them with different systems activated. For this we have the loop-active macro which will run code once through for each of the systems it is given:

```
1 >(loop-active (bin oct dec)
2   (sin-truevalue "100"))
3 4
4 64
5 100
```

#### 4.2.4 Standard Library

Note that `sin-truevalue` returns the 'true' decimal value, so this code is simply printing what 100 means in the three active systems. This is one of many standard library functions Sin has for making numeral system manipulation easier. A full list can be found in the appendix, but the key functions are the generic operators discussed next and the following system related functions:

- `sin-type` - returns the system of a string if a valid system is found.
- `sin-rawstring` - returns the string without a system.
- `sin-truevalue` - evaluates a string if a valid system is found.
- `sin-cast` - evaluates string using a given system.
- `sin-alphabet` - returns digit alphabet if a valid system is found.

Here are some examples of the standard library helper functions:

```
1 >(activate dec bin rmn)
2 >(sin-type "10")
3 DEC
4 >(sin-type "VII")
5 RMN
6 >(sin-cast "10" bin)
7 2
8 >(sin-alphabet "LXX")
9 ((I 1) (V 5) (X 10) (L 50) (C 100) (D 500) (M 1000))
10 >(sin-alphabet bin)
11 ((. NIL < 2) (0 0) (1 1))
```

The `sin-type` function can take either a string, a system, or a statically defined system value and it will return the system. In the case of being passed a string it will search the global stack for active systems and if the string is not valid in any active system it returns nil. This unambiguous approach is used in all the helper functions as seen in line 8 and 10 where the alphabets are extracted using a string and a system.

#### 4.2.5 Ambiguous Operators

Operators are prefixed with 'sin' and are generated or defined when the numeral system is defined.

```
1 (sin+ [sin-values...])
2 (sin- [sin-values...])
3 (sin* [sin-values...])
```

The inductive operator function takes a radix value and generates substitution rules as a list of pairs in the for `((shiftvalue addvalue) ...)` that describe arithmetic overflow for that radix.

For example given the radix value 2 the function outputs ((0 -2) (1 1)), which we read as saying "don't move and subtract 2" then "move 1 to the left and add 1". This describes  $1_2 + 1_2 = 10_2$  (or more specifically it describes  $2_2 = 10_2$ ).

The function obviously presumes a positional numeral system and so arithmetic rules for systems like Roman Numerals are not covered by this. To have the most minimal input (only the radix value) we have to presume a positional system then construct the first few numbers in the system via the Greedy Algorithm. The amount of base cases for a numeral system is the floor of the radix, the case after that is the inductive step and how the string of digits changes between the final base case and the inductive case gives us all the string substitution rules.

For binary numeral systems (with a radix  $1 > r \leq 2$ ) the algorithm works as follows:

---



---

```

Data: r
rules := emptylist;
one := GreedyBuild(1, r);
two := GreedyBuild(2, r);
for  $i \leftarrow \text{msd}(\text{two})$  to  $\text{lsd}(\text{two})$  do
  |   addv := two[i] - one[i];
  |   rules += pair(i, addv);
return rules

```

---

In the above the msd/lsd get the most and least significant digit *index* respectively (this is the power of the base in the first and last terms). For example msd(10.01) returns 1 and lsd(10.01) returns -2.

The above is how the `generate-rules` function shown earlier works, but it is also possible to manually set rules such as we do below for the definition of negabinary:

```

1 (defnumsys negabinary (binary)
2   :r -2
3   :rules (list
4             '(+ ((0 -2) (-1 1) (-2 1))))))

```

#### 4.2.6 Arithmetic Rules & (sin+)

There are two types of rules for our systems:

- Standardisation rule: ([mask, mask, ...], (add, add, ...))
- Normalisation rule: {(offset, add), (offset, add), ...}

They are both state-machine instructions used for manipulating a digit string. For normalisation (arithmetic overflow) whenever a value is higher than the highest alphabet value in said system then a shift & add method is used.

For example in a binary system the digit string is checked for any digits of value 2 or higher. When one is found the normalisation rule set is iterated through taking the (*offset, add*) pairs and moving along the digit string by the offset amount and adding the add amount. During standardisation the mask method is used. The mask method can be seen as a 1D kernel that is checked against the number at every digit, if the mask matches the number then the corresponding add kernel is applied.

As an example here are the rules for phinary

- Phinary standardisation ruleset:  $([0,1,1], (1,-1,-1))$
- Standardisation of 10011:  $10[011] \xrightarrow{(1,-1,-1)} 10100$
- Phinary normalisation ruleset:  $\{(-1, 1), (2,1), (0,-2)\}$
- Normalisation of 2:  $[2] \xrightarrow{(-1,1)} 1[2] \xrightarrow{(2,1)} 1[2].01 \xrightarrow{(0,-2)} 10.01$ .

The (sin+) function checks that the operands are all valid in the active system type, (defnumsys <system name>) generates a function called (<system name>p) for checking validity. So if the active system is phinary (sin+) first runs (phinaryp) on all the operands. If successful it then generates the rules by calling (phinary-rules). After this it takes the raw string value of all the numbers and converts them, using phinary's alphabet, to a list of digit values - then it does unitwise addition with all the operands. Then we enter the recursive cycle of normalisation (overflow rules) and standardisation. Essentially this part is:

---

```

function normalise (value)
  oldvalue := value;
  for each digit in value do
    if digit > alphabet max then
      [ Apply shift & add rules;
  if oldvalue != value then
    [ normalise(value);
  if oldvalue == value then
    [ standardise(value);
  return value

```

---

---

---

```
function standardise (value)
  oldvalue := value;
  for each digit in value do
    if current digit and neighbouring digits == mask digits then
      | Apply add rules to each digit locally;
  if oldvalue != value then
    | standardise(value);
  if oldvalue == value then
    | normalise(value);
  return value
```

---

In the next section we will be showing the Sin library at its best by generating fractal shapes. This will be an example the ease of writing string ambiguous code that can then have meaning bestowed on it with numeral systems.

## 5 Successor Patterns

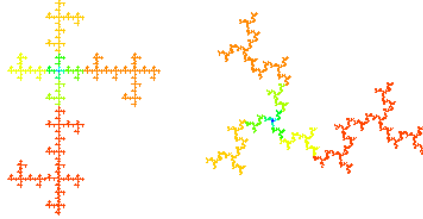


Figure 5: Phinary succession as seen in negarootbinary (left). Phinary succession as seen in Khmelnik's binary (right).

The relationship between numeral systems and fractals has been studied extensively [17] [18] [19] as has the relationship of string substitutions [20] [21].

The definition of a fractal is not as concrete as one would hope, but there are a few properties that are usually expected of them. One of these is self-similarity in which a portion of the set can be described as a scaled down version of the whole set. There is also a variant of self-similarity called self-affinity where the scaling of  $x$  and  $y$  differs [22]. Another form is statistical self-similarity such as the fractal describing of the coastline of Britain [23].

Other properties such as simple or recursive definitions are sometimes expected. Moving forward we will be using the term 'fractal' to mean self-similar tiling.

We introduce a novel model for generating fractals using numeral systems. These are called Successor patterns and they use two numeral systems to map an image onto the complex plane. They create a path using an arithmetic operation, originally it was integer succession  $succ(x) : x + 1$  hence the name but it is generalised further in [Def. 15, p42].

### 5.1 Generating Fractals

We talk in terms of the positional numeral system [Def. 1, p10] for the following definition.

**Definition 15.** Successor Pattern

Let  $c \in \mathbb{C}$  be a radix and  $p \in \mathbb{C}$  be a radix.

Let  $t$  be a digit string valid in the language  $L_c$ .

Let  $s$  range over  $L_c$ .

A successor pattern is defined by  $F_{c,p,t} := (S_c, M_p, t)$ , a 3-tuple containing inductive set  $S_c \subset L_c$ , a mapping function  $M_p : L_c \rightarrow \mathbb{C}$ , and an increment value  $t$ . We refer to  $c$  as the constructive radix and  $p$  as the projective radix.

The inductive definition of  $S_c$  and mapping of  $M_p$  are as follows

$$0 \in S_c$$

$$\forall s(s \in S_c \implies s +_c t \in S_c)$$

$$M_p(s) : \sum_{i=index(s)}^{-\infty} s_i p^i$$

For  $M_p(s)$  the function  $index(s)$  returns the position of the most significant digit relative to the radix point. The mapping of  $S_c$  by  $M_p$  gives us an image on the complex plane.

To give a working example of how one might use the System Interpreted Numbers language extension we will look at the *successor pattern function*, a function we designed. This is a program that generates a fractal image using arithmetic laws of one numeral system and the value representation of another. Whilst mainly for play, it gives us a visual example of the numeral system's traversal attributes and how greatly they can differ in a search space.

The successor pattern function works in two stages, interpret to generate and reinterpret to evaluate. Step one is to, using a specified numeral system, generate the digit representations of 0 to X. Step two is to reinterpret these numbers as if they were in a complex numeral system. Leaving us with a+bi which we can use as [x,y] coordinates for the image.

	0		0 + 0i
	1		1 + 0i
Generated using bi- nary succession	10	and reinterpreted as dragonbinary during evaluation	-1 + 1i
	11		0 + 1i
	100		0 - 2i
	101		1 - 2i

We can now use these complex values as [x,y] co-ordinates (where x=r and y=i) and draw an image that visualises digit progression in that numeral system. The algorithm is described below.

---

**function** *gen-successor-pattern* (X)

```

  digits := "0";
  output := emptylist;
  for 0 → X do
    Interpret digits as dragonbinary;
    Evaluate digits to decimal;
    Add result to output;
    Interpret digits as numsys;
    Increment digits;
  return output

```

---

## 5.2 Results

An extra step was taken in these results that limited the amount of points to be generated. This limit was done during the loop where we would +1 to the



current number, after the succession was done we would then check if the digit string was more than or equal to a limit digit string. This allowed us to collect all the digit strings that represent a tile of the fractal. A tile being for example 10000 to 11111, the tile dictated by the left most significant digit in the numeral system representation. Some of the fractals use 10x10 pixel squares to represent each point generated and some use 1 pixel per point.

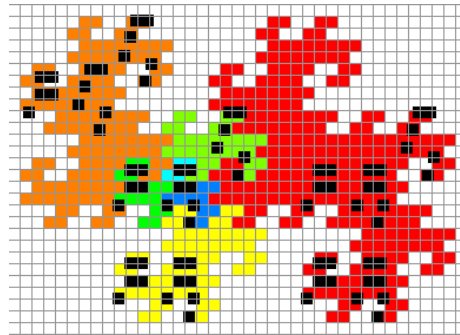


Figure 6: Here we have a coloured background fractal made from binary succession as seen in dragonbinary. On top of that is a black fractal showing phinary succession (both reinterpreted in dragonbinary). This uses 10x10 pixel squares per co-ordinate and the succession is +1 each time. Because of phinary's bidirectional expansion it requires a fractional part meaning most of the black squares are off-grid.

[Fig. 7, p44] is a comparison of six different fractals. We also have [Fig. 9, p46] which shows two perpendicular fractal paths based on the odd/even property of the unit increment (+1 being an even  $+\beta^0$ , +10 being an odd  $+\beta^1$ , and so on. There are also examples of non-binary systems generating fractals in both [Fig. 10, p47] and [Fig. 8, p45].

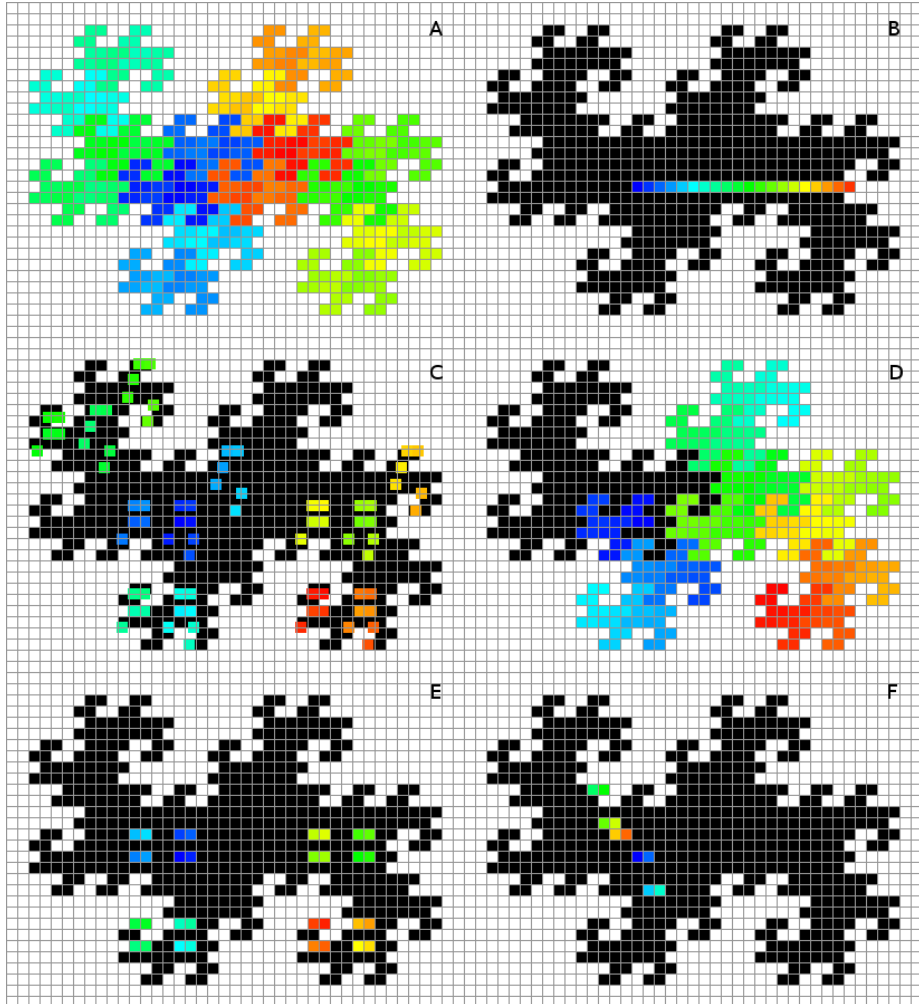


Figure 7: All these fractals are reinterpreted in dragonbinary with a bitstring limit of 1000000000. The black background fractals use standard binary succession giving full on-grid coverage of 0 to 111111111. The coloured fractals use succession of different numeral systems and are coloured blue to red based on their succession count.  $A : 2$ ,  $B : -1+i$ ,  $C : \frac{1+\sqrt{5}}{2}$ ,  $D : -2$ ,  $E : \sqrt{2}$ ,  $F : \frac{-1+\sqrt{-7}}{2}$ .



Figure 8: The code and subsequent fractal generation is not limited to only binary systems, here we have two systems using digits  $\{0,1,2,3\}$  both being reinterpreted in Donald Knuth's famous quater-imaginary system  $\langle 2i, \{0, 1, 2, 3\} \rangle$ . Left is quaternary  $\langle 4, \{0, 1, 2, 3\} \rangle$  and right is the third metallic numeral system the radix of which is sometimes called the bronze ratio  $\langle \frac{3+\sqrt{13}}{2}, \{0, 1, 2, 3\} \rangle$ .

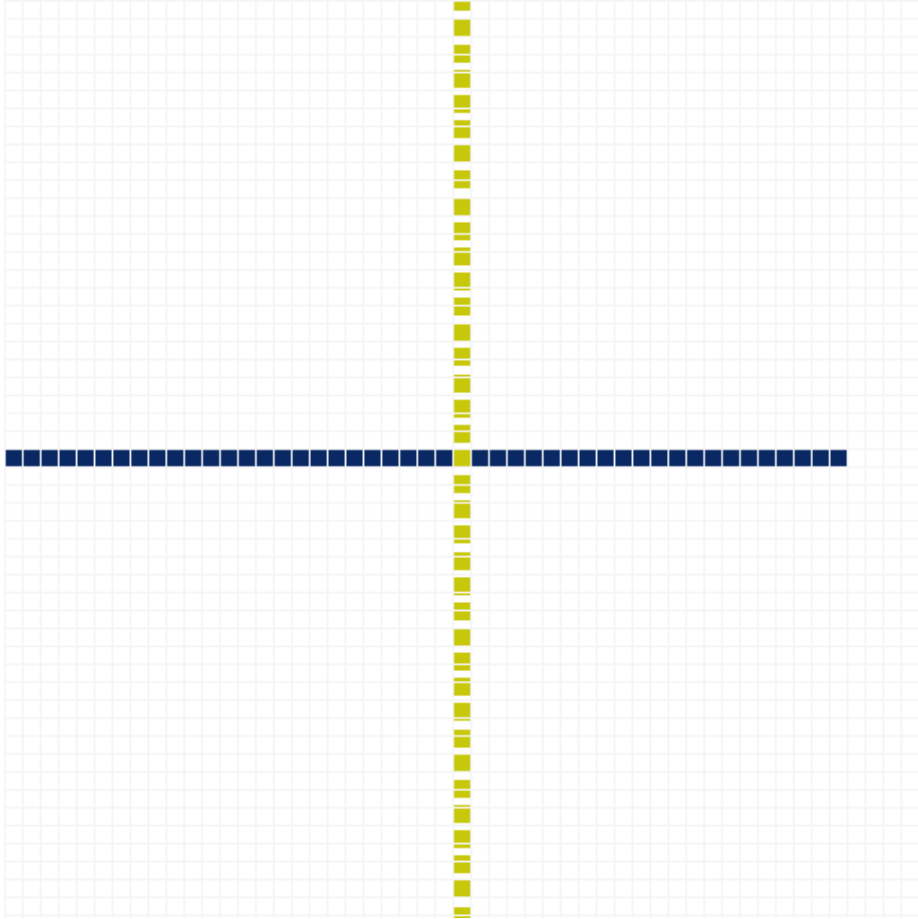


Figure 9: This is two fractals incremented in the same system (rootbinary) and evaluated in the same system (negarootbinary), but the increment for the blue fractal is +1 and the yellow fractal is +10. This setup is interesting in that whichever singular unit increment is used the resulting points are only ever on the axis lines but with differing spacing between the points. The larger the increment unit, say +1000000, the larger the gaps along the axis line. X-axis and Y-axis fractals are determined by the odd or even value of the increment unit position.

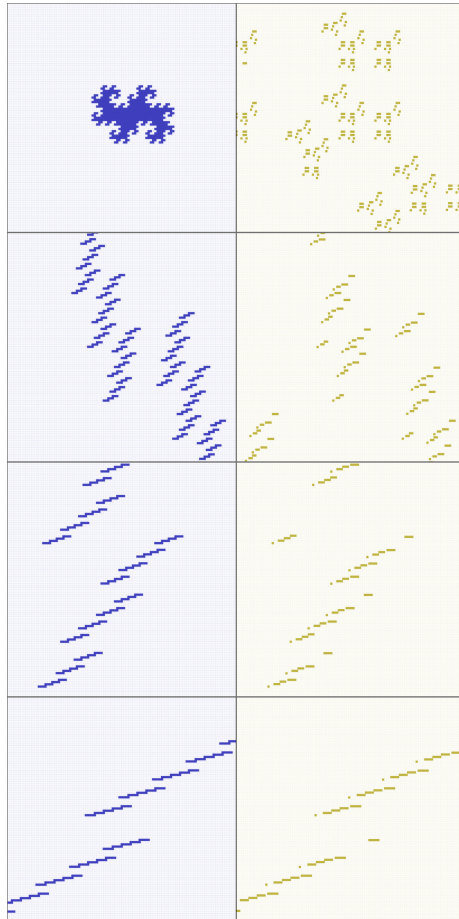


Figure 10: This is a comparison of the integer systems  $\{\beta = 2, \beta = 3, \beta = 4, \beta = 5\}$  on the left and the metallic systems  $\{\beta = \frac{1+\sqrt{5}}{2}, \beta = \sqrt{2} + 1, \beta = \frac{3+\sqrt{13}}{2}, \beta = \frac{4+\sqrt{20}}{2}\}$ . These families of systems tend closer together the higher up the radix values go.

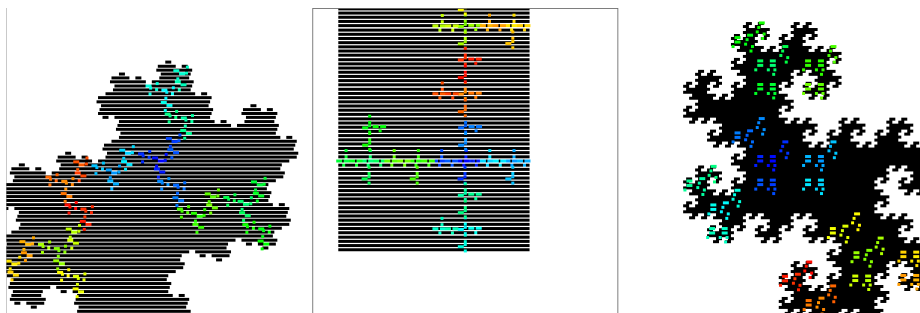


Figure 11: The three images here are made from 10x10 pixel squares for each co-ordinate. The black background image is the binary succession reinterpreted in three systems, from left to right: Khmelnik's binary, negarootbinary, and dragonbinary. The coloured fractals are phinary succession as seen reinterpreted in those same systems. The increment is +1 so it can be seen that dragonbinary has a clean coverage of the Gaussian integers.

### 5.3 The Fractal Generating Code

We shall discuss the Common Lisp code that generates the successor pattern co-ordinates. Here is a basic setup to generate the image coordinates.

```
1 (load "sin_library.lisp")
2
3 (defnumsys positional ()
4   :structure (sum-series +s+ term)
5   :digits (radix-alphabet r)
6   :term (* (expt r n) +d+)
7   :r 0
8
9   :rules (generate-rules r))
10
11 (defnumsys binary (positional) :r 2)
12 (defnumsys dragon (positional) :r #c(-1 1))
13 (defnumsys rootbinary (positional) :r (sqrt 2))
14 (defnumsys phinary (positional) :r (/ (+ 1 (sqrt 5))
15   2))
16
17 (defun gen-successor-pattern (amount)
18   (let ((value "0") (output nil))
19     (loop for i from 0 to amount do
20       (progn
21         (setq value (sin+ value "1"))
22         (setq output (append (list (dragon
23           value))))))
24     output))
```

The first line loads in all the Sin code, this is where (defnumsys) is defined and where various other helper functions such as (sum-series), (radix-alphabet), and (generate-rules) are defined.

Next we define the numeral system called 'positional'. We're using it as a template so it has a radix of  $r = 0$ . The most important thing for this definition is passing the 'rules' key the function (generate-rules r), this is stored as is so  $r$  is not evaluated to be 0 before storing this information. The call to (defnumsys positional...) generates a few different functions, one of them being (positional-rules) and when that is called it will evaluate (generate-rules r) using the stored value of  $r$ .

After the definition of the positional system we can then use it to cleanly define new positional systems by simply changing the value of  $r$ . Binary ( $r = 2$ ), dragon ( $r = -1 + i$ ), root ( $r = \sqrt{2}$ ), and phinary ( $r = \frac{1+\sqrt{5}}{2}$ ) all have the supertype positional. This now means that when (binary-rules) is called  $r$  will be 2 and when (root-rules) is called  $r$  will be  $\sqrt{2}$  and so on.

The power of the language can be seen in the last two lines of this loop. Note that (loop ... do <something>) expects one thing, so (progn ...) is used

to do more than one thing.

```
1      (loop for i from 0 to amount do
2        (progn
3          (setq value (sin+ value "1"))
4          (setq output (append (list (dragon
                                     value))))))
```

The value which starts out as "0" has a "1" added to it using the (sin+) generic operator. What this will do is check to see what the active system is and call (<system name>-rules) to generate the rules for said system, then it will perform the steps in subsection 1.3.

After this the list of results 'output' will have added to it the value interpreted as if it were in the dragon system. Whenever a numsys is defined with defnumsys it generates a function with the same name as the system which will evaluate back to decimal, for example (binary "11") will return 3. Because dragon has a complex radix this will return a complex value which will be our co-ordinates.



## 6 Genetic Algorithms

Charles Darwin first introduced the Theory of Evolution in 1858 which stated that variations occur in reproduction and are preserved through generations. In 1866 Gregor Mendel published the first probabilistic model of how the Theory of Evolution might work - this was the first concept of genetics. At the time Darwin's Natural Selection didn't rule out the possibility of Jean-Baptiste Lamarck's belief that an organism could pass on characteristics developed during its lifetime to its offspring (Lamarckism). By the end of the 19th century August Weismann, on the basis of his Germ Plasm Theory, had discarded Lamarckian views stating there was no way for the somatic cells (that make up the body) to communicate with the germ cells (used in reproduction). This is now known as the Weismann Barrier. Much later in 1953 James Watson and Francis Crick published the paper introducing the double-helix structure of DNA and its ATGC alphabet. The synthesis of these four ideas taken as a philosophy is called "Neo-Darwinism" and it is the philosophy behind genetic algorithms [24].

In biology there are three types of chromosome-structure for a cell: Haploid, Diploid and Multiploid. These are cells with one, two or multiple sets of chromosomes respectively. Humans, for example, are diploid with 23 pairs of chromosomes. Chromosomes consist of genes which hold the attributes of the phenotype, the possible states of the genes are called the alleles, the position of the gene in the chromosome is called the locus and the complete set of chromosomes is called a genotype. The most often used model for genetic algorithms is a haploid model with one chromosome per genotype [25] [26].

Genetic algorithms are metaheuristics that attempt to find a sufficiently good solution to either an optimisation or search space problem. The process begins with encoding an input into two or more parent genes and then breeding a new population of individuals by using the genetic operators 'crossover' and 'mutate'. Then each individual is judged with a fitness function and the fittest are used to populate the next generation. The encoded version of an individual is called the genotype (often encoded as bitstrings, but not always) and can be seen as the plans for a thing, the decoded version is called the phenotype and is the thing itself. The phenotype is what the fitness function tests, with the fitness of an individual being specific to the task at hand. Fitness may be how much a polynomial deviates from a set curve with the smallest deviation being the fittest, but also the fitness function could take a phenotype of character abilities then simulate that character in a situation to measure the fitness. [27].

### 6.1 Genetic Operators

The core features of a genetic algorithm are its population of possible solutions and its three 'genetic operators' which are crossover, mutation, and selection.

### 6.1.1 Crossover

Sometimes called recombination, this is the dominant feature that separates Evolutionary Algorithms (EAs) from Genetic Algorithms. The process involves taking two or more genotypes and combining them to make two or more new genotypes. Having a very high crossover rate may cause the genetic algorithm to prematurely converge by making all the chromosomes the same, though usually the crossover rate is higher than the mutation rate. Below are some types of crossover:

Single-Point	A random amount of a bit string is determined then swapped: 100 11100 and 111 00001 becomes 111 11100 and 100 00001
Multi-Point	Random sections of a bitstring are selected then swapped and not swapped alternatively
Uniform	Single bits are randomly swapped across the bitstrings

### 6.1.2 Mutation

Mutation can be seen as a conservator of diversity, although the quantity of its use is debated, for example [28] claim that with certain encodings it is better have a higher mutation rate. The usual approach is to have a low mutation rate because otherwise it becomes (in essence) a random search mechanic and will negatively affect the advantages of crossover. Below are some often used types of mutation:

Bitflip	Inverts a random bit in a string, so 1001110 0  becomes 1001110 1
Inversion	Inverts every bit in a string.

### 6.1.3 Selection

Selection is the method for picking individuals to mate and generate the new population. Of selection types, tournament can be used to control the efficiency. Here one has an increase in efficiency but a decrease in thoroughness. Below are some types of selection method:

Roulette-wheel	Random individuals are chosen and their probability of being selected is proportional to their fitness relative to the average
Tournament	Select best individual from a subset, repeat with different subsets, then breed a new population from the winners.
Boltzmann	The thermodynamic approach in simulated annealing is used. In short, the threshold for picking fitness starts large and most individuals can be picked but the threshold reduces after each picking (roughly analogous to thermal activity on rapidly cooling metal).

### 6.1.4 Arithmetic Evolutionary Algorithms

Evolutionary algorithms, as we saw in the previous section, are layered with abstraction and so can be hard to augment in ways that will certainly benefit the problem being optimised. The work in this thesis has laid down groundwork for easier empirical testing for the combination of genetic algorithms and numeral systems.

In the author's previous work a new mutation operator, *arithmutation*, was designed that used numeric overflow as a method of changing the bits in a bit string [1]. Instead of randomly flipping a bit from  $0 \rightarrow 1$  or  $1 \rightarrow 0$  the bit string was treated as a number and a random unit value was added to it. The example below uses  $\varepsilon$  as a random unit value of the radix such that  $\varepsilon \in \{1, 10_2, 100_2, 1000_2, \dots\}$ , it changes randomly for each action.

Arithmutation on an Arbitrary Bitstring

State	Action	Result
00100010	$+_2 \varepsilon$	[1]0100010
10100010	$+_2 \varepsilon$	101000[2]0
	overflow	10100[1]20
	overflow	101001[0]0
10100100	$+_2 \varepsilon$	1010010[1]
10100101	$+_2 \varepsilon$	101001[1]1

The process of arithmutation can be done using any numeral system that has a defined addition operator and that matches the possible digits in the digit string being used for the encoding. This is the deep encoding's alignment (encoding) and traversal (arithmutation) that models numeral system characteristics in an evolutionary algorithm. The mutation code itself is simply two actions: 1) generate a random unit (e.g., 001, 010, 100, etc) with as many digits as the genotype, then 2) add the unit to the genotype using the arithmetic rules of the encoded numeral system. This is the code found in our implementation:

```
;; The Arithmutation is arithmetic carry
;; applied after adding a random unit to the number.
;; This uses cl-sin functions to generate a random
   number
;; containing only 1 unit then adding it to the
   genotype.
```

```
(defun arithmutate (genotype)
  (let* ((unit (gen-number "10" (length genotype)
    ) :unit #\1)))
    (sin-fixed+ genotype unit)))
```

Note here that the function (sin-fixed+) is a special version of (sin+) where the length of the output string is determined by the length of the first input parameter (here that is the genotype).

We do not present any proofs on the validity of arithmutation, but we instead make a more formal claim and then discuss the analysis methods one might use in this endeavour.

**Claim 2.** The Doorless Hotel Hypothesis, Part 2

*A genetic algorithm can embed the alignment characteristic in the encoding  $\eta$ , and embed the traversal characteristic in the mutation  $\mu$ . Encoding  $\eta$  limits the possible phenotypes by the density of its radix representation. Mutation  $\mu$  allows the search space to be traversed from one population to the next using characteristics of arithmetic. Represented in [Eq. (26), p54].*

$$\begin{array}{ccc}
 \text{genotype}_i & \xrightarrow{\eta} & \text{phenotype}_i \\
 \downarrow \mu & & \\
 \text{genotype}_{i+1} & & 
 \end{array} \tag{26}$$

### 6.1.5 Working Implementation

A fully working Evolutionary Algorithm written in Common Lisp that uses System Interpreted Numbers (Sin) to implement the arithmutation operator and numeral system encoding in a dynamic way can be found at [29]. The Sin code is not well optimised at this stage so rigorous empirical testing was out of the scope of this project. If testing were to take place the rest of the thesis looks at how we would approach that.

## 6.2 Analysis of Genetic Algorithms

Genetic Algorithms (GAs), which are an idealised generalisation of evolution and natural selection, are hard to analyse due to the fact that their time complexity is related to characteristics of the problem that is being optimised for. The first breakthrough in trying to formalise *why* GAs seemed to perform well was found with Holland’s Schema Theory [30]. It made a probabilistic statement about the propagation of schemata, pattern-matched genotype templates, from one generation’s population to the next. It states that in genetic algorithm life cycles, short low-order schemata with above average fitness increase exponentially in successive generations. The formulation of Schema Theory was inline with the most fundamental statement in the field of Genetic Algorithms which is the Building Block Hypothesis.

**Claim 3.** The Building Block Hypothesis

*A genetic algorithm seeks optimal performance through the juxtaposition of short, low-order, high-performance schemata, called building blocks.*

There are problems with Schema Theory such as it presupposes an infinite population and can not track propagation throughout many cycles [30] [31], and while its statement on selection operator is exact, its statement on mutation

and crossover is only probability based [31]. Various more rigorous extensions to Schema Theory and alternative approaches to formalising GAs have been employed since then such as

**Markov Chain Analysis** [32] [33]: A Markov chain is a stochastic process modelled in discrete-time by some a transition matrix  $Q$ , where  $Q_{i,j}$  is the probability of moving from state  $i$  to state  $j$ . It holds the Markov Property which is that future states only depend on the current state and not previous states. For genetic algorithms this means modelling the possible states as the possible populations.

**Walsh Functions** [34] [35]: Schema Theory fails to take into account the possible stochastic deviation of a GA's population fitness. The direction of evolution may sway away from the global optimum. So a new formula for the calculation of schema fitness variance is derived by modelling schemata fitness with Walsh Polynomials.

**Exact Schema Theory** [31]: Taking into account the previous two approaches, this approach enhances Holland's model by making exact statements on mutation and crossover.

**Theoretical Run-Time Analysis** [36]: The above approaches are all inclined to make statements on infinite populations or of only one cycle of the population. With run-time analysis a more pragmatic statement can be made about real-world GAs/problems [37], albeit only on simple models. An important paper in GA Theory gave a proof that for a *real* (implementation variation allowed) GA there is an optimisation problem where crossover is essential to have polynomial run-time [38]. The mutation-only method gives exponential run-time. A phase-based analysis was used giving five phases that each had a goal state for the health of the population (the last phase goal being an absolute solution must be found). The run-time for each phase is summed up and trivial parts are removed to give the final proof statement.

### 6.3 Problem Space: Rosenbrock

We are looking at function optimisation and so for us a problem space is the set of any mathematically valid inputs to some function  $f$ . A search space is the set of all computationally feasible inputs for  $f$  and is a subset of the problem space.

The author's previous empirical results were seen on variants of the Rosenbrock function [1], which is defined as follows

$$f(x, y) = (a - x)^2 + b(y - x^2)^2. \quad (27)$$

The Rosenbrock function has a deep valley of good solutions but only one global optimum which is in the form  $f(a, a^2)$ . Usually  $a = 1$  and  $b = 100$ , for us however we change the value of  $a$  to augment the problem space.

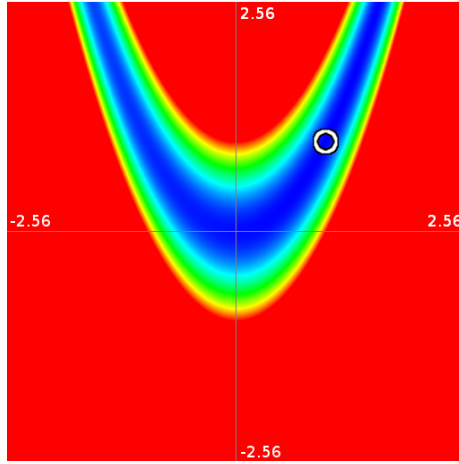


Figure 12: Plot of Rosenbrock function (R1) where the x and y axis range from  $-2.56$  to  $2.56$ . Colours are calculated from  $f(x, y)$  and are capped and normalised between  $f(x, y) = 0$  and  $f(x, y) = 100$ . Solution  $f(1, 1) = 0$  circled.

$f$	$f(x, y) = 0$	Reference name
$(1 - x)^2 + 100(y - x^2)^2$	$f(1, 1)$	R1
$(\sqrt{5} - x)^2 + 100(y - x^2)^2$	$f(\sqrt{5}, 5)$	R5
$(\varphi - x)^2 + 100(y - x^2)^2$	$f(\varphi, \varphi + 1)$	Rp

The problem space can be characterised by its fitness landscape, in [Fig. 12, p56] we map each point on the image to a colour based on  $f(x, y)$ . When  $f(x, y) \geq 100$  it is red, moving through a jet pseudocolour map towards blue at the most fit  $f(x, y) = 0$ . The global optimum is marked with a ring and is at position  $(x = 1, y = 1)$ .

The variants (R5) and (Rp) differ by a rather small visual amount, but keep the overall shape. They still retain the deep valley of local optima.

We put forward that the Rosenbrock function and it's power series related solution of  $f(a, a^2) = 0$  is a well suited test subject for deep encoding of genetic algorithms. Run-time analysis would be an interesting approach to take to this.

## 7 Conclusion

During the work of this thesis we have been able to classify a set of useful numeral systems that are known to hold specific properties and act in specific ways useful to search space representation. These were a set of positional numeral systems using an integer or Pisot radix. This section honed in on the structure of the positional numeral system and how flexible it could be.

We wrote a proof that the arithmetic operations of addition and subtraction for any metallic numeral system would complete for any two valid digit strings. Including digit strings that are numerically valid but not inside the normalised form for the system, for example phinary is a system with an alphabet of  $\{0, 1\}$  but it was proven by us that the addition operation between arbitrary numerical digit strings such as  $74385.82327457_\varphi + 1234.56789_\varphi$  will eventually terminate.

A second proof was made which states that no integer value can be represented by an irrational radix without the use of radix expansion. The proof was the result of a coding question: *Can a radix point be omitted from any encoding by using a unique radix?* of which the answer is, if the integers are required, no.

We were able to generate arithmetic laws for a subset of these systems with one universal algorithm. We then use this information to develop the main output of the thesis, System Interpreted Numbers (Sin), A Common Lisp library capable of defining numeral systems then dynamically setting the active system allowing the user to use basic strings that will then be interpreted as numbers. Sin has inbuilt functions for the specific class of numeral systems that allow it to do alphabet generation and arithmetic rule generation. The latter is for standardisation and addition overflow. Allowing for normalisation rules to be generated from the radix alone or added in manually. An evolutionary algorithm was written that uses a string encoding, this was then given a special decoding and mutation method which use the dynamic active system scope to evaluate the string encoding as representation of a numeral system and the mutation function uses the generates rules of arithmetic to randomly add a unit to the encoded number. The evolutionary algorithm is tremendously flexible and is written with binary bitstrings so by simply activating different binary systems before running we can entirely change or decode/mutate methodology. Along the way we also came up with a novel method for generating fractal shapes by using Sin's ability to interpret strings in any valid predefined numeral system.

The scope of this thesis didn't include the empirical testing of evolutionary algorithms, only the development of the algorithms and the numeral system framework on which they lay. Future work would consist of doing extensive empirical testing of arithmetic evolutionary algorithms and analysing the results through the scope of run-time analysis.

Currently the Sin language is not well optimised and for large test groups both Sin and the accompanying evolutionary algorithm will need further development to be as efficient as possible.

## 8 Appendix

### Core Functions

<b>decnumsys</b> <i>system</i>	Statically declares a variable to be of numeral system <i>system</i> .
<b>defnumsys</b> <i>name</i> <i>super-type</i> [( <i>structure body</i> ) ( <i>digits body</i> ) ( <i>term body</i> )...]	Creates a new numeral system type name <i>name</i> .
<b>activate</b> <i>systems</i>	Clears current active systems and sets it to <i>systems</i> .
<b>seta</b> <i>name</i> <i>value</i> [ <i>numsys</i> ]	Sets the string value of a statically declared numeral system variable.
<b>loop-active-systems</b> <i>systems</i> <i>body</i>	For each system in <i>systems</i> the system is activated and code <i>body</i> is executed.
<b>loop-active-systems-multi</b> <i>systems</i> <i>body</i>	For each list of systems in <i>systems</i> the list of systems is activated and code <i>body</i> is executed.
<b>sin+</b> <i>operands</i> ...	If all operands are of matching type and there exists addition rules then addition takes place.
<b>sin-fixed+</b> <i>operands</i> ...	If all operands are of matching type and there exists addition rules then addition takes place. If the output length does not match the length of the first operand, it is truncated or padded with zeros where needed.



## Helper Functions

<b>sin-type</b> <i>numsys</i>	Returns the numeral system type.
<b>sin-rawstring</b> <i>numsys</i>	Returns the string value or nil.
<b>sin-truevalue</b> <i>numsys</i>	Evaluates the string using the static or active numeral system and returns the decimal value.
<b>sin-cast</b> <i>value numsys</i>	Evaluates value as if it were of type numsys and returns the decimal value.
<b>sin-numeral</b> <i>numeral numsys</i>	Returns the numeral value from the alphabet of numsys.
<b>sin-alphabet</b> <i>numsys [(digits-only value)]</i>	returns the alphabet of numsys, value pairs can be ignored.
<b>sin-value-list</b> <i>value</i>	Converts a string into a decimal list based on an alphabet.
<b>sin-rules</b> <i>numsys (pick value)</i>	Returns the rule sets for numsys. Specific rules may be chosen.
<b>sin-description</b> <i>numsys</i>	Prints a description of numsys.
<b>anu</b> <i>number</i>	Returns an (A)lpha (N)umeric (U)ppercase character where (anu 0) ==> #\0 and (anu 10) ==> #\A.
<b>anl</b> <i>number</i>	Returns an (A)lpha (N)umeric (L)owercase character where (anu 0) ==> #\0 and (anu 10) ==> #\a.
<b>gen-number</b> <i>numsys length (unit value) (unit value)</i>	Generates a random number in system numsys.
<b>gen-range</b> <i>start finish digit-func</i>	Generates a range.
<b>sum-series</b> <i>str body</i>	Loops across string str and performs code body summing the value of each loop.
<b>radix-alphabet</b> <i>radix</i>	Returns an alphabet valid for radix.
<b>generate-rules</b> <i>radix</i>	Return set of rules valid for radix.

## References

- [1] D. Ward-Williams, “Comparing binary systems in genetic algorithm encoding,” 10 2013.
- [2] D. Wolpert and W. Macready, “Coevolutionary free lunches,” *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 721–735, 2005.
- [3] N. A. I. et al, “Understanding color models: A review,” *ARPJ Journal of Science and Technology*, 2012.
- [4] SharkD, “Color solid comparison,” 2008. [Online; accessed 2021].
- [5] W. F. Anderson, “Arithmetic in maya numerals,” *American Antiquity*, pp. 54–63, 1971.
- [6] Patrick Gray, “Inca - quipu,” 2016. [Online; accessed 2021].
- [7] O. R. Keister, “The incan quipu,” *The Accounting Review*, vol. 39, no. 2, pp. 414–416, 1964.
- [8] A. Stakhov, “Brousentsov’s ternary principle, bergman’s number system and ternary mirror-symmetrical arithmetic,” *Comput. J.*, vol. 45, pp. 221–236, 02 2002.
- [9] D. E. Knuth, “A imaginary number system,” *Communications of the ACM*, vol. 3, no. 4, pp. 245–247, 1960.
- [10] I. Solomon, “Positional codes of complex numbers and vectors,” *viXra*, 2008.
- [11] A. Rényi, “Representations for real numbers and their ergodic properties,” *Acta Mathematica Academiae Scientiarum Hungarica*, 1957.
- [12] W. Parry, “On the  $\beta$ -expansions of real numbers,” *Acta Mathematica Academiae Scientiarum Hungarica*, 1960.
- [13] C. Frougny and B. Solomyak, “Finite beta-expansions,” *Ergodic Theory and Dynamical Systems*, vol. 12, no. 4, p. 713–723, 1992.
- [14] E. W. Weisstein, “Algebraic integer. From MathWorld—A Wolfram Web Resource.”
- [15] K. Schmidt, “On Periodic Expansions of Pisot Numbers and Salem Numbers,” *Bulletin of the London Mathematical Society*, vol. 12, no. 4, pp. 269–278, 1980.
- [16] D. Ward-Williams, “System interpreted numbers.” <https://bitbucket.org/NeuralOutlet/cl-sin>, 2021.
- [17] W. Gilbert, “Fractal geometry derived from complex bases,” *The Mathematical Intelligencer*, vol. 4, pp. 78–86, 06 1982.

- [18] W. Gilbert, “The fractal dimension of sets derived from complex bases,” *Canadian Mathematical Bulletin*, vol. 29, 12 1986.
- [19] D. Goffinet, “Number systems with a complex base: a fractal tool for teaching topology,” *The American Mathematical Monthly*, vol. 98, no. 3, pp. 249–255, 1991.
- [20] K. Scheicher and J. Thuswaldner, “Canonical number systems, counting automata and fractals,” *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 133, 10 2001.
- [21] P. Arnoux and E. Harriss, “What is a rauzy fractal,” *Notices of the AMS*, vol. 61, no. 7, 2014.
- [22] B. B. Mandelbrot, “Self-affine fractals and fractal dimension,” *Physica Scripta*, vol. 32, pp. 257–260, oct 1985.
- [23] B. Mandelbrot, “How long is the coast of britain? statistical self-similarity and fractional dimension,” *science*, vol. 156, no. 3775, pp. 636–638, 1967.
- [24] C. Reeves and J. Rowe, “Genetic algorithms: Principles and perspectives: A guide to ga theory,” *Operations Research/ Computer Science Interfaces Series*, vol. 20, pp. 1–3, 01 2002.
- [25] R. Kumar and J. Yotishree, “Haploid vs diploid genome in genetic algorithmsfor tsp,” *International Journal of Computer Science and Information Security*, 2010.
- [26] Y. e. a. Wang, “Improved genetic algorithm based complex-valued encoding,” *International Journal of Computer Science and Network Security*, 2009.
- [27] R. C. Charkaborty, “Fundamentals of genetic algorithms,” 2010.
- [28] A. E. Smith and D. M. Tate, “Genetic optimization using a penalty function,” in *Proceedings of the 5th International Conference on Genetic Algorithms*, (San Francisco, CA, USA), p. 499–505, Morgan Kaufmann Publishers Inc., 1993.
- [29] D. Ward-Williams, “Arithmetic evolutionary algorithm.” <https://bitbucket.org/NeuralOutlet/evolutionary-algorithm>, 2021.
- [30] D. White, “An overview of schema theory,” 2014.
- [31] A. Wright, “The exact schema theorem,” *Computing Research Repository - CORR*, 05 2011.
- [32] A. H. Wright and Y. Zhao, “Markov chain models of genetic algorithms,” in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, GECCO’99, (San Francisco, CA, USA), p. 734–741, Morgan Kaufmann Publishers Inc., 1999.

- [33] H. Dawid, “A markov chain analysis of genetic algorithms with a state dependent fitness function,” *Complex Systems*, vol. 8, 01 1994.
- [34] D. E. Goldberg and M. Rudnick, “Genetic algorithms and the variance of fitness,” *Complex Systems*, vol. 5, 1991.
- [35] D. E. Goldberg, “Genetic algorithms and walsh functions: Part i, a gentle introduction,” *Complex Systems*, vol. 3, 1989.
- [36] F. N. Benjamin Doerr, “The benefits of population diversity in evolutionary algorithms: A survey of rigorous runtime analyses,” in *Theory of Evolutionary Computation* (D. Sudholt, ed.), pp. 359–404, 2019.
- [37] P. S. Oliveto, J. He, and X. Yao, “Time complexity of evolutionary algorithms for combinatorial optimization: A decade of results,” *International Journal of Automation and Computing*, 2007.
- [38] T. Jansen and I. Wegener, “Real royal road functions - where crossover provably is essential,” *Discrete Applied Mathematics*, vol. 149, pp. 111–125, 08 2005.