Generating Geometric Sequences Through Convolution

Stefan Gerber

March 28, 2026

Abstract

The goal for this paper is to be able to demonstrate the full construction of several novel algorithms for the calculation of geometric sequences. The algorithms proposed are proved for correctness using techniques in counting, number theory, algebra, and Fourier Analysis. These methods manipulate both the base and exponent to an expanded form requiring only simple binary operations to generate the resultant sequences. Expanding the exponential expression in this way attempts to achieve efficiency by using many simple operations to compute what would otherwise be an expensive calculation. This is initially shown by tracing the construction in base two which is further generalized to higher bases. These techniques were shown to be ineffective at reducing the practical execution time for geometric sequence generation.

Contents

List of Figures

1 Introduction

1.1 Geometric Sequences

1.1.1 Geometric Sequence Definition

A Geometric Sequence will be defined as an ordered set of numbers of the form $A = \{a * b^i : a, b \in \mathbb{R}, i, k \in \mathbb{N}, 0 \leq i \leq k\}$, where i increments by 1, $[a * b^0, a * b^1, a * b^2, ..., a * b^k]$. This sequence features three main components that makes it unique. a, b, k.

'a' is the scale factor of the sequence. This is able to shift the exponent as well.

'b' represents the base of the sequence. This is the common ratio arises when dividing two neighboring elements ab^{i+1} by ab^i is $\frac{ab^{i+1}}{ab^i}$.

'k' is the highest value of i within the sequence. This may correspond to the highest exponent of the sequence if the sequence is not shifted. The length of the sequence is also dictated as |A| = k + 1.

Values of the sequence may be expressed as A_i or a_i which equates to the i^{th} element of the 0 indexed sequence, or simply ab^i.

1.1.2 The Uses of Geometric Sequence

Within mathematics and computing, there are several uses for geometric sequences. Any sort of iterative method or incremental series might be a candidate for geometric sequences. Some examples where these types of sequences are utilized are the following.

- Taylor/Maclaurin Series function approximations
- Taylor Method Initial Value Problem Differential Equation Solver
- Various physical processes involving energy decay or growth for use within pendulum motion, bouncing ball, or radioactive decay problems.

These methods do not specifically call for the use of geometric sequences, but the elements of the sequence are used within the calculations. The calculations generally scale individual elements of the sequence by some scale factor dependent upon the index. This is how the rebound height of a bouncing ball would be calculated. Each bounce decays the energy of the system by the common ratio or its square. Therefore each element in the sequence would represent the rebound height of the ball after that index's count of bounces. For usages within series, the entire scaled sequence might be summed together. The k value would then be increased to yield a potentially more accurate approximation as is done within The Taylor Series.

To be able to effectively apply these methods within a computer program, a geometric sequence solver must be constructed that not only produces the correct sequence elements, but also produces them in an efficient manner when compared to similar methods.

1.1.3 Geometric Sequence Generation Methods

There are some techniques that can be used in the generation of these geometric sequences of length k + 1.

- Generating the sequence can be done by simply evaluating exponentiation of the base and multiplication of the scale for every element in the sequence.
- A slightly more efficient technique utilizing the previous element's calculation as a step towards the current element's value. There is the recursive technique that involves multiplying the common ratio with the previously calculated sequence element. This recursive strategy is equivalent to $A_{i+1} = A_i b$ where $A_0 = a$
- The first simple evaluation technique can be modified to be parallelized in which the evaluations are split across multiple compute units to be done at once.

It is important to note that the problem of generating geometric sequences is not equivalent to the problem of calculating exponents. Calculating exponents is a component of geometric sequences, but it does not represent the entire problem. Any exponent calculation algorithm can be used to generate geometric sequences.

- The exponent by squaring algorithm is an example of an optimized algorithm that is much more efficient at calculating exponents than simply using iterative multiplication which is analogous to the first example of geometric sequence solver proposed above. Exponentiation by squaring takes advantage of a decomposition of the exponent breaking it down into squaring and multiplications by x using the binary representation of the exponent. For example $x^{17} = x^{10001_2} = x^1(x^0(x^0(x^0(x^1)^2)^2)^2)^2$. The total geometric sequence elements that are calculated at any time through this process are $[1, x, x^2, x^4, x^8, x^{16}, x^{17}]$. This method leaves increasingly large gaps in the sequence that must be filled in as the exponent increases. These gaps would need to be filled in by manually multiplying leading this method to be no better than the recursive multiplication approach when generating an entire geometric sequence.
- A more generalized version of this algorithm is the m-ary exponentiation algorithm which "reduces to the binary method discussed earlier, when m = 2" [1]. This m-ary version replaces the squaring operation with the exponentiation by the m value. The instruction that each potential unique value of the m-ary string represents is $x^{d \mod m} * (c)^m$ where c is the current number. By following the m-ary string and this instruction set, the exponent will be reached. This requires a precomputed window of exponential values to be calculated $x^0, x^1, ..., x^{m-1}$, which is in itself a geometric sequence. This has the same problem as the m = 2, binary case since there would still b gaps that need to be filled in.

These methods all operate upon the same principle of recursively multiplying the common ratio with a neighboring element to yield a full sequence.

1.1.4 Operations Upon Powers of 2

All of the current methods for generating a geometric sequence on a single compute unit rely on this recursive multiplication method. The multiplication of increasingly large numbers, as outlines within the previous geometric sequence methods might lead to the overall execution time increasing as well. This body of work defines new decompositions of geometric sequences that use simple operations such as bit shifting (multiplication or division by powers of 2) and addition or subtraction to form a sequence of any common ratio. These decompositions would require additional steps to represent the sequence, but these more efficient power of 2 sub calculations might provide a practical reduction in execution time.

1.1.5 Introduction To Binomials

Throughout this work, binomial coefficients are utilized extensively as solutions or components of proofs. These coefficients constitute the rows of Pascal's Triangle. This triangle is built in such a way that each number is the sum of the two elements from the previous row directly above it. The start of the triangle is shown below.

$k = 0$:				1				
$k = 1$:			1		1			
$k = 2$:		1		2		1		
$k = 3$:	1		3		3		1	
$k = 4$:	1	4		6		4		1

There are four properties of these numbers that are known to be true and used throughout the proofs of this document.

- The sum of each 0 indexed row corresponds to a non negative integer power of 2 $(2^0, 2^1, 2^2...)$ where the row number is the exponent.
- Each row also corresponds to the coefficients the would be present by fully multiplying out the binomial $(x+y)^k$, where k represents the row number.
- Each row is symmetric.
- Each element within the triangle can be expressed as $\binom{k}{r}$ 'k choose r' or $\frac{k!}{r!(k-r)!}$ where k is the row number and r is the index of the element within the 0 indexed row. It does not matter if the element's index within the row is chosen right to left or left to right due to the symmetry of the row.

The numbers of the triangle are often called binomial coefficients and are used to solve counting problems. $\binom{k}{r}$ represents the number of ways to choose r item(s) from a set of k unique items where the ordering of the items does not matter. For example with a set of $\{a, b, c\}$, k = 3 so the Pascal's Triangle row would be [1,3,3,1]. There then exists 1 ways to pick no items from the set $\{\}$, 3 ways to pick 1 item $\{a, b, c\}$, 3 ways to pick 2 items $\{ab, ac, bc\}$, and 1 way to pick 3 items $\{abc\}$.

1.2 Array functions

In order to logically decompose a geometric sequence in terms of these bit shifts, the concept of an array function must be defined. For the purposes of this work, an Array Function will be defined as a function that is performed upon an array. Although quite simple, the mean or averaging function is an example of one such function. The mean array function is defined as $MEAN(A) = \frac{1}{|A|}\sum_{i=0}^{|A|-1} A_i$, |A| being the length of A, and A_i, being the zero indexed ith member of A. It is important to establish these naming conventions as this forms the basis for the later sections. These array functions, will be defined such that both their input and output are both arrays. It can be argued that a function who's output is just a number does architecturally output an array of size 1. For the purposes of this paper, we will use this assumption and all array functions will be defined as a $Function : [\mathbb{R}] \rightarrow [\mathbb{R}]$, where the function takes a real valued array to a real valued array.

1.3 Patterns

When observing an array of numbers, it is only natural to start and begin to ascertain some sort of pattern amongst the elements. For some random ordered set of real numbers, contained within an array, there must be some pattern amongst them. This is of course a bold claim, but it can quickly be realized when introducing functions to the array. For example the array A = [-3, 54, 0.5, -9, 2.5], one might immediately attempt to scan the array for some semblance of a pattern. There might in fact be some sort of pattern or closed form solution that explains the sequence such as a Lagrange polynomial, however there is an easier explanation involving functions. These randomly chosen numbers represents one of the combinations of numbers that average to a value of [9]. This therefore gives meaning to A which is simply a random array of numbers.

Throughout this paper we will be conducting a process similar to this for a different array function in the forwards approach (as was shown above for MEAN(A) = [9]), but also in its backwards, inverse (MEAN^{-1}([9]) = A). This method requires a different approach as it involves manufacturing some sort of array who's application of the given array function is the given input value. For the case of the MEAN function, clearly there are an infinite amount of possible values that A might take on in the inverse application. Some of the examples are [9], [9, 9, 9, 9], [8,9,10], or [0,18]. Each of those arrays are valid inputs to yield an average of [9], but by imposing some kind of constraint, the domain

can be reduced to a potential unique solution. For example, forcing A to only consist of single instances of powers of 2 would perhaps pose some challenge to find a unique array similar to The Subset Sum Problem. For example, an input of 9 has value of $A = [2^2, 2^4]$, while a value of 25 has no solution, checked for 10^7.

These inputs were checked using the python program found at Section_1/inverse_mean.py. Accompanying this document, there is a command line utility tool that accompanies the sections which can be run at main.py. This tool breaks down each of the functions and solvers created within the following sections into the following groupings. The grouping names will become more clear throughout the sections.

1. Scalar Functions
2. Array Functions
3. Matrix Functions
4. Leading Edge Functions
5. Memoization Functions
6. Rational Approximation Functions
7. Geometric Sequence Solvers
8. Analysis
9. Applications
10. Utility Functions

These ten groupings do not correspond directly to the sections of the paper, however the actual file system does group the functions based by section. The inverse mean solver can be accessed at Scalar Functions/Inverse Mean.

This main.py file requires the following libraries to be installed: pydantic, scipy, matplotlib, pandas, numpy, sympy.

Before generating geometric sequences, some basic definitions must be proven. Sections 2 - 3 demonstrate the functions required to properly generate these sequences.

2 The Midpoint Function

2.1 Midpoint Array Function

2.1.1 Element wise Midpoints

The main array function that will be used within this section is the successive midpoint which will be defined as follows.

$$Midpoint(A) = \begin{cases} [\frac{A_i + A_{i+1}}{2} : i \in \mathbb{Z}, 0 \leq i \leq |A| - 2] & |A| > 1 \\ A & |A| \leq 1 \end{cases} \quad (1)$$

This function is similar to the traditional midpoint formula for two elements, but is extended to be operable upon an array. The function takes the midpoint of each neighboring pair in order, outputting the result to a new array. Assuming there are k elements within the array (k $>$ 1), clearly the resultant array from Midpoint will have k - 1 elements since the last element has no neighboring pair to its right Midpoint($[A_0, A_1, A_2, \ldots, A_{k-3}, A_{k-2}, A_{k-1}]$) $=[\frac{A_0+A_1}{2}, \frac{A_1+A_2}{2}, ..., \frac{A_{k-3}+A_{k-2}}{2}, \frac{A_{k-2}+A_{k-1}}{2}]$. This array wise function accesses the end values (A_0, A_{k-1}) once (quantity of 2), and each of the interior values twice (quantity of - 2), for k - 1 additions and divisions by 2. The total array accesses is 2(k - 1) for k - 1 additions and k - 1 divisions, resulting in a total amount of operations to be 4k - 4 when k $>$ 1. This leads to a total worst case running time complexity of O(k) when k $>$ 1 or a best case running time of $\Omega(1)$ when k $\leq$ 1. Additionally, this algorithm can actually be implemented to be in place by assigning each new i^{th} individually calculated midpoint pair to the i^{th} slot. This is possible since that value of the original i^{th} array element no longer needs to be accessed after its second access. The ending slots could be filled with INF to denote their non access which yields an auxiliary space complexity of O(1).

The scalar midpoint function is available at 'Section_2/midpoint.py' or within the command line program at 'Scalar Functions/Midpoint'.

The midpoint array function is available at 'Section_2/forwards_midpoint.py' or within the command line program at 'Array Functions/Forwards Midpoint'.

2.1.2 Successive Applications of The Midpoint Array Function

Since the Midpoint function has been defined to input and output arrays, the resultant array could be infinity passed to another Midpoint function. This recursion would possess a maximum depth before it outputs the same singleton array. Therefore this recursive structure could generate only a maximum of k - 1 unique output arrays. Since the input contains n elements, each successive iteration reduces the output array size by 1, k - 1 passes through the Midpoint function would reduce the output array size to 1 (k - (k - 1) = 1). This successive midpoint iterates k - 1 times over an O(k) Midpoint array function, which results in a final running time complexity of $O(k^2)$.

The successive nature of taking this array function has been described in Discrete-Time Signal Processing as a cascade connection system.

> "In a cascade connection of systems, the output of the first system is the input to the second, the output of the second is the input to the third, etc. The output of the last system is the overall output." [2]
>
> This recursive Midpoint(Midpoint(... A)), can be explored at different sizes of array A. In the following equations three different array sizes are tested k = 2, 3, 4. Additionally, A_i represents the output of the ith iteration of the Midpoint function and a$_i$represents the ith element of A.

k = 2, $A_0 = [a_0, a_1], A_1 = [\frac{a_0+a_1}{2}]$.

k = 3, $A_0 = [a_0, a_1, a_2], A_1 = [\frac{a_0+a_1}{2}, \frac{a_1+a_2}{2}], A_2 = [\frac{\frac{a_0+a_1}{2}+\frac{a_1+a_2}{2}}{2}] = [\frac{a_0+2a_1+a_2}{4}]$
.

k = 4, $A_0 = [a_0, a_1, a_2, a_3]$, $A_1 = [\frac{a_0+a_1}{2}, \frac{a_1+a_2}{2}, \frac{a_2+a_3}{2}]$, $A_2 = [\frac{a_0+2a_1+a_2}{4}, \frac{a_1+2a_2+a_3}{4}]$
$A_3 = [\frac{\frac{a_0+2a_1+a_2}{4}+\frac{a_1+2a_2+a_3}{4}}{2}] = [\frac{a_0+3a_1+3a_2+a_3}{8}]$.

It appears that some sort of pattern has emerged when examining the coefficients and the denominator. It seems as though the singleton that is produced is the weighted average of the initial input array's elements scaled with binary coefficients.

$$SuccessiveMidpoint(A) = [\frac{1}{2^{|A|-1}} \sum_{i=0}^{|A|-1} \binom{|A|-1}{i} a_i] \tag{2}$$

The proof of Formula 2 is the following.

Proof. Successive applications of Formula 1, yields Formula 2.

Taking these successive midpoints on a general array, B, results in a final scalar $m = \frac{1}{2^n} \sum_{i=0}^{k} \binom{k}{i} b_i$ where b_i is the i-th element of the 0-indexed B and $k = |B| - 1$.

Base Case

Let there be a base B of length 2: $B = [b_0, b_1]$. The successive midpoint of this array only has one iteration, $m = (b_0 + b_1)/2$. Therefore

$$\begin{aligned} m &= \frac{1}{2^1} \sum_{i=0}^{1} \binom{1}{i} b_i \\ &= \frac{1}{2} \left(\binom{1}{0} b_0 + \binom{1}{1} b_1 \right) \\ &= (b_0 + b_1)/2 = m \end{aligned}$$

Inductive Step k = p -1

Now there is an array B of length $k - 1$. Since the array is indexed it can be split into two different bases.

$B_0 = B - b_k = [b_0, b_1, \ldots, b_{k-1}]$. $B_1 = B - b_0 = [b_1, b_2, \ldots, b_k]$.

Assuming that the successive midpoint of B_0 is $m_0 = \frac{1}{2^{p-1}} \sum_{i=0}^{p-1} \binom{p-1}{i} b_i$. The successive midpoint of B_1 is $m_1 = \frac{1}{2^{p-1}} \sum_{i=0}^{p-1} \binom{p-1}{i} b_{i+1}$.

These two points m_0, m_1 can have a midpoint applied to them to find the final midpoint of B which should be equal to $\frac{1}{2^p} \sum_{i=0}^{p} b_i \binom{p}{i}$.

This final midpoint m should be equivalent to $\frac{1}{2^p} \sum_{i=0}^{p} \binom{p}{i} b_i$.

$$
\begin{aligned}
m &= (m_0 + m_1)/2 \\
&= \frac{\left(\frac{1}{2^{p-1}} \sum_{i=0}^{p-1} \binom{p-1}{i} b_i + \frac{1}{2^{p-1}} \sum_{i=0}^{p-1} \binom{p-1}{i} b_{i+1}\right)}{2} \\
&= \frac{1}{2^p} \left(\sum_{i=0}^{p-1} \binom{p-1}{i} b_i + \sum_{i=0}^{p-1} \binom{p-1}{i} b_{i+1} \right) \\
&= \frac{1}{2^k} \left(\sum_{i=0}^{p-1} \binom{p-1}{i} b_i + \sum_{i=1}^{p} \binom{p-1}{i-1} b_i \right) \\
&= \frac{1}{2^p} \left(\binom{p-1}{0} b_0 + \sum_{i=1}^{p-1} \binom{p-1}{i} b_i + \sum_{i=1}^{p-1} \left(\binom{p-1}{i-1} b_i \right) + \binom{p-1}{p-1} b_p \right) \\
&= \frac{1}{2^p} \left(b_0 + \sum_{i=1}^{p-1} \left(\binom{p-1}{i} b_i + \binom{p-1}{i-1} b_i \right) + b_p \right) \\
&= \frac{1}{2^p} \left(b_0 + \sum_{i=1}^{p-1} b_i \left(\binom{p-1}{i} + \binom{p-1}{i-1} \right) + b_p \right)
\end{aligned}
$$

Pascal's identity: $\binom{n}{r} + \binom{n}{r-1} = \binom{n+1}{r}$

Let $n = p - 1$ and $r = i \implies \binom{p-1}{i} + \binom{p-1}{i-1} = \binom{p}{i}$.

Therefore

$$
\begin{aligned}
&= \frac{1}{2^p} \left(b_0 + \sum_{i=1}^{p-1} b_i \binom{p}{i} + b_p \right) \\
&= \frac{1}{2^p} \sum_{i=0}^{p} b_i \binom{p}{i}.
\end{aligned}
$$

This means that the successive midpoint returns a binomial weighted arithmetic mean of the input array. This makes the Successive Midpoint function and the MEAN array function demonstrated in Section 1.1, to be of the same class of function. This new function achieves a normalized weighted average by dividing by the sum of the weights (as shown by the binomial theorem) much like any other averaging function.

Example 1. Assume an input array of A = [3,9,7]. The successive midpoint could be easily computed through the usage of Formula 2.

SuccesiveMidpoint([3,9,7]) = $\frac{1*3+2*9+1*7}{2^2} = 7$. Although verification of this can be achieved by manually computing the Midpoint for each successive row. The following matrix, M, represents each of the rows produced throughout the calculation.

n = 3, $A_0 = [3, 9, 7], A_1 = [\frac{3+9}{2}, \frac{9+7}{2}] = [6, 8], A_2 = [\frac{6+8}{2}] = [7]$. M =

3	9	7
6	8	0
7	0	0

Observing row 0 in this 0 indexed matrix results in the original array A_0, [3,9,7], which was input into the Successive Midpoint function. The elements comprising the anti diagonal [7,8,7] represent the terminal elements within each step of midpoint succession (A_0, A_1, A_2). Finally, the 0^{th} column, denoted as A' = [3,6,7], shows the first element from each of the progressions (A_0, A_1, A_2). A' being of course a column vector as opposed to A, being a row vector. It is important to note that A' does not note the transpose of A, but rather the first column of the matrix achieved when combining together the steps of successive applications of the midpoint. Since both the anti diagonal and A' are arrays, they can be passed back as arguments to additional rounds of successive midpoints. Section 2.2, discusses the case of using the first column as the input to another round of successive midpoint applications.

The successive midpoint function is available at 'Section_2/successive_midpoint.py' or within the command line program at 'Matrix Functions/Successive Midpoint'.

2.2 The Anti Midpoint Array Function

This section formalizes how A' or the first column, can be transformed back to the original input array A or first row. This A' will be known as the leading edge of the matrix, which is equivalent to the first column.

2.2.1 Leading Edge Definitions

First, the notion of this A' column must be formalized. A leading edge can be be created by augmenting the process of performing a successive midpoint by simply appending the first element of each created array including the input to a new array. This allows the leading edge to be defined in an iterative manner.

Theorem 1. *A weighted sum upon an array can be defined as* $S_m = \sum_{i=0}^{m} a_i w_i$ *where* a_i *are the elements of the input array A and* w_i *is the weight attributed to that element. The leading edge is defined in the following way. A' =* $[S_0, S_1, ..., S_k]$. *Each element of A' represents the linear combination of the first k weights with the first k elements.*

Corollary 1. *Leading Edge Generation via Theorem 1*

The successive midpoint taken upon a continuous subset of elements from array A taken from index i to index j will result in the i_{th} *element of the (j - i)*th *successive midpoint row* A_{j-i}.

The Proof of SuccessiveMidpoint demonstrates that the successive applications of the midpoint formula will yield a binary coefficient weighted sum (Formula 2).

Assuming an input array A of length n, clearly the leading edge would be composed of the concatenation of the singleton arrays created from the successive midpoints of increasingly lengthened slices of A. When A[i:j], i, j$\in \mathbb{Z}, 0 \leq i, j \leq k-1, i \leq j$ *represents a subsection slice of A including the elements of A with indices between and including i, j.*

A' = [sucessiveMidpoint(A[0:0]),sucessiveMidpoint(A[0:1]),...,sucessiveMidpoint(A[0:k-2],sucessiveMidpoint(A[0:k-1])]

$=\lfloor \frac{a_0}{2^0}, \frac{a_0+a_1}{2^1}, \frac{a_0+2a_1+a_2}{2^3}, \ldots, \frac{1}{2^{k-2}} \sum_{i=0}^{k-2} a_i \binom{k-2}{i}, \frac{1}{2^{n-1}} \sum_{i=0}^{k-1} a_i \binom{k-1}{i} \rfloor$

Assuming A has a length of k.

Base case.

The first element of A', $a'_0 = \frac{a_0}{2^0} = a_0$.

Induction Step.

Assuming the k - 1th row $A_{k-1} = B$ *is comprised of two elements [*b_0,b_1*] where presumably*

$b_0 = \frac{1}{2^{k-2}} \sum_{i=0}^{k-2} a_i \binom{k-2}{i}$,$b_1 = \frac{1}{2^{k-2}} \sum_{i=0}^{k-2} a_{i+1} \binom{k-2}{i}$, *the resultant midpoint of this should be equivalent to* $\frac{1}{2^{k-1}} \sum_{i=0}^{k-1} a_i \binom{k-1}{i}$. *In a similar fashion to Proof 1.*

$\frac{1}{2}\left(\frac{1}{2^{k-2}} \sum_{i=0}^{k-2} a_i \binom{k-2}{i} + \frac{1}{2^{k-2}} \sum_{i=0}^{k-2} a_{i+1} \binom{k-2}{i}\right)$

$=\frac{1}{2^{k-2}}(a_0 + \sum_{i=1}^{k-2} a_i \binom{k-2}{i} + \sum_{i=1}^{k-2} a_i \binom{k-1}{i-1} + a_{k-1})$ *(Pascal's Identity)*

$=\frac{1}{2^{k-1}} \sum_{i=0}^{k-1} a_i \binom{k-1}{i}$

It has now been demonstrated that the 0^{th} column created in a successive midpoint matrix is equivalent to the definition of a leading edge in Theorem 1.

The iterative leading edge function is available at 'Section_2/midpoint_leading_edge.py' or within the command line program at 'Leading Edge Functions/Midpoint Leading Edge.

2.2.2 Defining The Anti Midpoint Function

Now that A' is properly defined in terms of A, A must also be able to be defined in terms of A'. Rather, from some leading edge, how can we return the initial starting array?

Let us assume some small general 2x2 and 3x3 matrices M that follows the properties from Corollary 1.

k = 2 A = [a_0,a_1], M = $\begin{matrix} a_0 & a_1 \\ \frac{a_0+a_1}{2} & 0 \end{matrix}$ A'=[$a_0, \frac{a_0+a_1}{2}$].

k = 3 A = [a_0,a_1,a_2], M = $\begin{matrix} a_0 & a_1 & a_2 \\ \frac{a_0+a_1}{2} & \frac{a_1+a_2}{2} & 0 \\ \frac{a_0+2a_1+a_2}{4} & 0 & 0 \end{matrix}$ A'=[$a_0, \frac{a_0+a_1}{2}, \frac{a_0+2a_1+a_2}{4}$].

It might not be immediately apparent what function could transform A' to A. Since successive midpoints were able to create these A' arrays potentially some sort of "inverse" midpoint function could undo the application of the successive midpoint function.

Observing the simple scalar midpoint to be $a'_1 = \frac{a_0+a_1}{2}$, solving for a_1, $2a'_1 - a_0 = a_1$. Since the first column and first rows share their first element, $a_0 = a'_0$, $2a'_1 - a'_0 = a_1$. This function serves as the inverse of the midpoint when applied in the scalar case, but it must be proven to be able to work in the successive case.

As was done in Section 2.1.1 this anti midpoint must be transformed to be able to be an array function rather than just upon two points. In order to formalize this, the anti midpoint array function will be defined as follows.

$$AntiMidpoint(A') = \begin{cases} [2A'_{i+1} - A'_i : i \in \mathbb{Z}, 0 \leq i \leq |A'| - 2] & |A'| > 1 \\ A' & |A'| \leq 1 \end{cases} \quad (3)$$

Where it inputs a leading edge A' of length k, outputting an array of length n - 1 as before if k $>$ 1 or A' if k $\leq$ 1.

It is important to note that the anti midpoint is not the inverse of the midpoint. The anti midpoint is only the inverse of the midpoint when operating upon some number x and 1. $2\frac{1+x}{2} - 1 = x$

The scalar anti midpoint function is available at 'Section_2/anti_midpoint.py' or within the command line program at 'Scalar Functions/Anti Midpoint'.

The anti midpoint array function is available at 'Section_2/backwards_anti_midpoint.py' or within the command line program at 'Array Functions/Backwards Anti Midpoint'.

2.2.3 Closed Form Formula For Successive Applications of The Anti Midpoint

As was done in Section 2.1.2, increasing general lengths of A' can be tested to attempt to glean a hint of a pattern when performing successive applications of the anti midpoint.

k = 2, $A'_0 = [a'_0, a'_1], A'_1 = [2a'_1 - a'_0]$.

k = 3, $A'_0 = [a'_0, a'_1, a'_2], A'_1 = [2a'_1 - a'_0, 2a'_2 - a'_1], A'_2 = [2(2a'_2 - a'_1) - (2a'_1 - a'_0)] = [4a'_2 - 4a'_1 + a'_0]$.

k = 4, $A'_0 = [a'_0, a'_1, a'_2, a'_3]$, $A'_1 = [2a'_1 - a'_0, 2a'_2 - a'_1, 2a'_3 - a'_2]$, $A'_2 = [4a'_2 - 4a'_1 + a'_0, 4a'_3 - 4a'_2 + a'_1]$ $A_3 = [2(4a'_3 - 4a'_2 + a'_1) - (4a'_2 - 4a'_1 + a'_0)] = [8a'_3 - 12_2a' + 6a'_1 - a'_0]$.

This pattern might be a little bit more difficult to spot than the successive midpoint case, but it appears to be the following.

$$SuccessiveAntiMidpoint(A') = \sum_{i=0}^{|A'|-1} (-1)^{(|A'|-1-i)} 2^i \binom{|A'|-1}{i} a'_i \quad (4)$$

The proof of Formula 4 is the following.

Proof. Successive Applications of Formula 3 yields Formula 4.

This proof follows a similar procedure to Proof 1. We aim to prove that for a basis $B' = [b'_0, b'_1, \ldots, b'_k]$, the recovered element b_k of the original basis B is given by:

$$b_k = \sum_{i=0}^{k} (-1)^{(k-i)} \, 2^i \binom{k}{k-i} b'_i$$

Base Case

For some basis of 2 elements $B' = [b'_0, b'_1]$, the resulting b_1 from the Anti Midpoint should be equal to $2b'_1 - b'_0$. Since $b_0 = b'_0$. We check the formula for $k = 1$:

$$\begin{aligned} b_1 &= \sum_{i=0}^{1} (-1)^{(1-i)} \, 2^i \binom{1}{1-i} b'_i \\ &= (-1)^{(1-0)} \, 2^0 \binom{1}{1-0} b'_0 + (-1)^{(1-1)} \, 2^1 \binom{1}{1-1} b'_1 \\ &= -b'_0 + 2^1 b'_1 \end{aligned}$$

The base case holds.

Inductive Step k = p + 1

For some new basis of $p+1$ elements, $B' = [b'_0, b'_1, \ldots, b'_k]$, the Anti Midpoint (recovered element b_k) should be:

$$b_p = \sum_{i=0}^{p} (-1)^{(p-i)} \, 2^i \binom{p}{p-i} b'_i$$

This basis B' can be split into two sub-bases. $B'_0 = [b'_0, b'_1, \ldots, b'_{k-1}]$ and $B'_1 = [b'_1, b'_2, \ldots, b'_k]$. Assuming that these bases can have the successive Anti Midpoints applied to them to achieve m'_0 and m'_1 respectively where they equal the following:

$$m'_0 = \sum_{i=0}^{p-1} (-1)^{(p-1-i)} \, 2^i \binom{p-1}{p-1-i} b'_i$$

$$m'_1 = \sum_{i=0}^{p-1} (-1)^{(p-1-i)} \, 2^i \binom{p-1}{p-1-i} b'_{(i+1)}$$

So therefore, $b_p = 2m'_1 - m'_0$. We substitute the expressions for m'_0 and m'_1:

$$
\begin{aligned}
b_p &= 2\left(\sum_{i=0}^{p-1}(-1)^{(p-1-i)}\cdot 2^{(i)}\binom{p-1}{p-1-i}b'_{(i+1)}\right) \\
&\quad -\left(\sum_{i=0}^{p-1}(-1)^{(p-1-i)}\cdot 2^{(i)}\binom{p-1}{p-1-i}b'_{i}\right) \\
&= \sum_{i=1}^{p}(-1)^{(p-1-(i-1))}\cdot 2^{(i)}\binom{p-1}{p-1-(i-1)}b'_{i} \\
&\quad -\sum_{i=0}^{p-1}(-1)^{(p-1-i)}\cdot 2^{(i)}\binom{p-1}{p-1-i}b'_{i} \\
&= \sum_{i=1}^{k}(-1)^{(p-i)}\cdot 2^{i}\binom{p-1}{p-i}b'_{i} + \sum_{i=0}^{p-1}(-1)^{(p-i)}\cdot 2^{(i)}\binom{p-1}{p-1-i}b'_{i} \\
&= (-1)^{(p-p)}\cdot 2^{p}\binom{p-1}{p-k}b'_{p} \\
&\quad +\sum_{i=1}^{p-1}(-1)^{(p-i)}\cdot 2^{i}\binom{p-1}{p-i}b'_{i} \\
&\quad +\sum_{i=1}^{p-1}(-1)^{(p-i)}\cdot 2^{(i)}\binom{p-1}{p-1-i}b'_{i} \\
&\quad +(-1)^{(p-0)}\cdot 2^{(0)}\binom{p-1}{p-1-0}b'_{0} \\
&= \left((-1)^{(p-p)}\cdot 2^{p}\binom{p-1}{p-p}b'_{p}\right) \\
&\quad +\left(\sum_{i=1}^{p-1}\left((-1)^{(p-i)}\cdot 2^{i}b'_{i}\right)\left(\binom{p-1}{p-i}+\binom{p-1}{p-1-i}\right)\right) \\
&\quad +\left((-1)^{(p-0)}\cdot 2^{(0)}\binom{p-1}{p-1-0}b'_{0}\right)
\end{aligned}
$$

Pascal's Identity: $\binom{N}{R} + \binom{N}{R-1} = \binom{N+1}{R}$. Let $N = p-1$ and $R = p-i$.

$$
\begin{aligned}
&= (-1)^{(p-p)} \cdot 2^p \binom{p-1}{p-p} b'_p \\
&\quad + \left(\sum_{i=1}^{p-1} (-1)^{(p-i)} \cdot 2^i \binom{p}{p-i} b'_i \right) \\
&\quad + (-1)^{(p-0)} \cdot 2^{(0)} \binom{p-1}{p-1-0} b'_0 \\
&= \sum_{i=0}^{p} \left((-1)^{(p-i)} \cdot 2^i \binom{p}{p-i} b'_i \right) \quad \text{QED.}
\end{aligned}
$$

Assume the input array of A = [3,9,7] from Example 1. The successive midpoint matrix was determined to be M = $\begin{matrix} 3 & 9 & 7 \\ 6 & 8 & 0 \\ 7 & 0 & 0 \end{matrix}$, with a leading edge of A' = [3, 6, 7].

Example 2. Applying the successive anti midpoint formula on A' = [3, 6, 7]

SuccesiveAntiMidpoint([3,6,7]) $= 4 * 7 - 4 * 6 + 3 = 7$, which corresponds to final element within the original A array. Verification of this can be checked manually, placed column by column into an array to demonstrate that the leading edge formed upon the 0^{th} row is equivalent to A in its entirety. It should be noted that the successive anti midpoints are placed into the matrix as columns as opposed to rows within the matrix to align with the property demonstrated within Theorem 1.

k = 3, $A'_0 = [3,6,7]', A'_1 = [2*6-3, 2*7-6] = [9,8], A'_2 = [2*8-9] = [7]$

. M = $\begin{matrix} 3 & 9 & 7 \\ 6 & 8 & 0 \\ 7 & 0 & 0 \end{matrix}$

Interestingly, the sum of the weights used for each calculation of the Anti-SuccesiveMidpoint function sums to be 1 and the sum of their magnitudes is the $3^{|A'|-1}$. This is verified in the following corollary.

Corollary 2. *Anti Successive Midpoint is a properly weighted arithmetic mean.*

By formula 4, the weights of the mean are corespondent to the following sum. Assuming |A'| = k + 1

$\sum_{i=0}^{|A'|-1} (-1)^{\left(|A'|-1-i\right)} 2^i \binom{|A'|-1}{|A'|-1-i}$

$= \sum_{i=0}^{k} (-1)^{(-i)} 2^i \left(\frac{(k)!}{(k-i)!(k-(k-i))!} \right)$

$= \sum_{i=0}^{k} (-1)^{(k-i)} 2^i \frac{(k)!}{(k-i)!i!}$

$= \sum_{i=0}^{k} (-1)^{(k-i)} 2^i \binom{k}{i}$

$= (2 + (-1))^k$ *(Binomial Theorem)*

$= 1$

Additionally, taking the magnitude of the weights is equivalent to the following.

$\sum_{i=0}^{|A'|-1} |\,(-1)^{(|A'|-1-i)}\, 2^i \binom{|A'|-1}{|A'|-1-i}| = \sum_{i=0}^{k} 2^i \binom{k}{i}$

$=(2+1)^k$ *(Binomial Theorem)* $= 3^k$

Therefore the Anti Successive Midpoint is a proper weighted mean, with the added fact that the summed magnitude of the weights is equivalent to $3^{|A'|-1}$.

The successive anti midpoint function is available at 'Section_2/successive_anti_midpoint.py or within the command line program at Matrix Functions/Successive Anti Midpoint'.

2.2.4 The Inverse of Midpoint Leading Edge

We must first formalize two proposed inverse functions.

MidpointLeadingEdge and MidpointLeadingEdge^{-1}. By the definition of the leading edge formalized by Theorem 1, the leading edges generated by the functions are the following. Assume that the big cup indexed union operation creates the ordered union of these array elements. $\bigcup_{i=0}^{|A|-1}[a_i] = A$

$$MidpointLeadingEdge(A) = \bigcup_{i=0}^{|A|-1} [\frac{1}{2^i} \sum_{j=0}^{i} \binom{i}{j} a_j] \tag{5}$$

$$MidpointLeadingEdge^{-1}(A') = \bigcup_{i=0}^{|A'|-1} [\sum_{j=0}^{i} (-1)^{(i-j)}\, 2^j \binom{i}{j} a'_j] \tag{6}$$

For these to be be truly a well defined inverse, $MidpointLeadingEdge^{-1}(MidpointLeadingEdge(A)) = A$

For some input array A, MidpointLeadingEdge(A) will create an array of the general form $[a_0, \frac{a_0+a_1}{2}, \frac{a_0+2a_1+a_2}{4}, ..., \frac{1}{2^{|A|-1}} \sum_{i=0}^{|A|-1} \binom{|A|-1}{i} a_i]$ as demonstrated in Formula 5. This then will be used as the A' input within Formula 6 to output the general array form of.

$[a_0, -a_0+2\frac{a_0+a_1}{2}, a_0-4(\frac{a_0+a_1}{2})+4\frac{a_0+2a_1+a_2}{4}, ..., \sum_{i=0}^{|A|-1} (-1)^{(|A|-1-i)}\, 2^i \binom{|A|-1}{i} a'_i]$

$= [a_0, a_1, a_2, ...]$

It is clear that the formula describing the ith element of this array generated by $MidpointLeadingEdge^{-1}(MidpointLeadingEdge(A)) = A$ is of the following form.$a_i = \sum_{j=0}^{i} \left((-1)^{(i-j)}\, 2^j \binom{i}{j} \left(\frac{1}{2^j}\right) \left(\sum_{k=0}^{j} \binom{j}{k} a_k \right) \right)$

Which represents the weights of the anti midpoint Formula 4 of length i, being applied to the corresponding element of the A' array found by Formula 2.

Lemma 1. *Inclusion Exclusion*

$=\sum_{s=0}^{m} (-1)^{(m-s)} \binom{m}{s}$ *(m = i - k, s = j - k)*

$=(1+(-1))^{i-k} = 0^{i-k}$ *(so clearly when i != k the sum is 0)*

$=\sum_{j=i}^{i} (-1)^{(i-j)} \binom{0}{j-i}$ *(setting k := i, collapses the sum to a single term where j = i)*

$=(-1)^{(i-i)}\binom{0}{i-i} = 1$ *(i=k, sum is 1)*

$\sum_{j=k}^{i}(-1)^{(i-j)}\binom{i-k}{j-k} = \begin{cases}1 & i=k\\ 0 & else\end{cases}$

Proof. Formula 6 Is The Inverse of Formula 5

$a_i = \sum_{j=0}^{i}\left((-1)^{(i-j)}2^j\binom{i}{j}\left(\frac{1}{2^j}\right)\left(\sum_{k=0}^{j}\binom{j}{k}a_k\right)\right)$

$=\sum_{j=0}^{i}\sum_{k=0}^{j}(-1)^{(i-j)}\binom{i}{j}\binom{j}{k}a_k$

$=\sum_{k=0}^{i}\sum_{j=k}^{i}(-1)^{(i-j)}\binom{i}{j}\binom{j}{k}a_k$ (Reindexing)

$=\sum_{k=0}^{i}\binom{i}{k}\sum_{j=k}^{i}(-1)^{(i-j)}\binom{i}{k}\binom{i-k}{j-k}a_k$ (Vandermonde's identity)

$=\sum_{k=0}^{i}a_k\binom{i}{k}\sum_{j=k}^{i}(-1)^{(i-j)}\binom{i-k}{j-k}$

$\sum_{k=i}^{i}a_k\binom{i}{i}$ (The interior value of the outer sum is equal to 1 when k = i else 0 by Lemma 1)

$=a_i$

Therefore MidpointLeadingEdge and the inverse operation are properly defined in Formula 5 and Formula 6.

The successive anti midpoint function is available at 'Section_2/inverse_midpoint_leading_e or within the command line program at 'Leading Edge Functions/Inverse Successive Leading Edge'.

2.2.5 Successive Midpoint Property.

Finally, the successive Midpoint Property can be defined for a matrix.

Theorem 2. *The Successive Midpoint Property as applied to a matrix*

Let M be a square matrix of size n by n, initialized to zeros. M possesses the successive midpoint property if any of the two conditions are met.

1. Each row of M besides the first row is the array defined midpoint of the previous row, with null values denoted as 0.
2. Each column of M besides the first column is the array defined anti midpoint of the previous column, with null values denoted as 0.

The matrix that is created to follow these properties is defined as the Cascade Matrix. The elements of this matrix are solved to be the following where M is a one indexed array. This means that for any input array, the cascade matrix created from

$$M_{i,j} = SuccesiveMidpoint(A[j : i+j-1]) = SuccessiveAntiMidpoint(A'[i : j+i-1]) \tag{7}$$

These two properties have been shown to be equivalent through showing the inverse property holds for Formula 5 and Formula 6. A matrix generated by either a successive midpoint of A as an input row or the matching A' column

as a successive anti midpoint will be equivalent. Both A and A' are unique to the initial column or row that generates them.

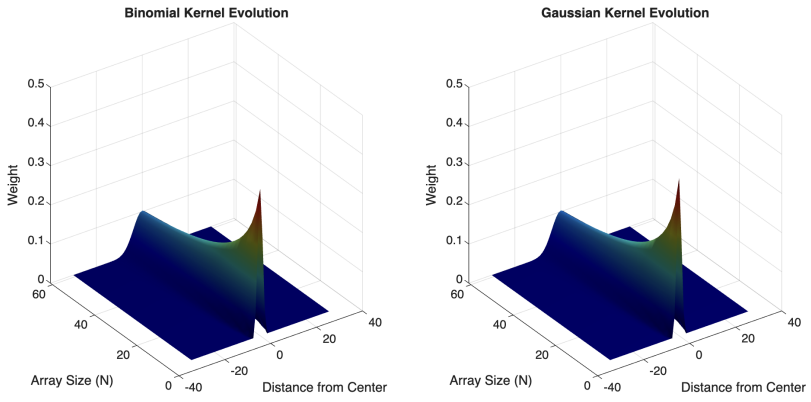
It is important to note that the inverse of the SuccessiveMidpoint function is not indeed the SuccesiveAntiMidpoint function. This clearly can be demonstrated through the fact that SuccessiveMidpoint outputs an array with a singular value, rendering SuccesiveAntiMidpoint to default to the base case returning its input value in AntiSuccesiveMidpoint(SuccessiveMidpoint(A)). The inverses are only defined for the leading edges that are generated by taking the first element of each successive iteration of the midpoint or anti midpoint upon the inputs.

3 Applications of The Successive Midpoint Property

Section 2 has formalized the notion of the cascade matrix for an arbitrary input row A or column A'. Using this arbitrary input has only the apparent benefit of creating a weighted arithmetic mean upon the elements. This could be useful in situations in which the inputs have an some order where elements around the center require an increased weight than those further from the center. The effect of this binomial filter is very similar to what would be a discrete gaussian kernel or even a triangular kernel.

The following image, visualizes the binomial kernel and gaussian kernel for different lengths.

Figure 1: Binomial Kernel vs Gaussian Kernel Comparison



This is a widely used technique when applied to an arbitrary input. It can be applied to two dimensional kernels for imaging or one dimensional kernels for signal processing. However, the remainder of this discussion will make use of the formulae proven within Section 2 applied onto a class of specific input arrays to generate geometric sequences. Input arrays (rows and columns) will be denoted using augmented big cup notation to denote the union of arrays. This $\bigcup_{i=0}^{k}[]$ will not result in the union of elements of a set, but in fact the union of the singleton arrays within the operator. This is equivalent to simply concatenating or appending the current array to the running array preserving the order. Input rows will continue to be defined as A while output columns will be defined as A' .

3.1 Constant Input Array

3.1.1 Input Constant Row Leading Edge

Let the entire input row be defined as such $A = \bigcup_{i=0}^{k}[c]$ where c is some constant and k is the desired length. Clearly the successive midpoint applied to this input will result in an cascade matrix completely comprised of c. The midpoint between c and c results in itself. Therefore A' = A.

By Theorem 1, the pth 0 indexed element of A' should be $\frac{1}{2^p}\sum_{i=0}^{p} a_i\binom{p}{i}$ (Formula 2)

Since $a_i = c$, this can be substituted,

$a'_p = \frac{1}{2^p}\sum_{i=0}^{p} c\binom{p}{i}$

$= \frac{c2^p}{2^p} = c$

$A' = \bigcup_{i=0}^{k}[c]$

3.1.2 Input Constant Row Inverse Leading Edge

Now the resultant leading edge for the anti midpoint case can be found when using an initial constant input.

$A' = \bigcup_{i=0}^{k}[c]$

By Theorem 1, the pth 0 indexed element of A should be $\sum_{i=0}^{p}(-1)^{(p-i)}\,2^i\binom{p}{i}a'_i$ (Formula 4)

a'_i has been demonstrated to be c for all i.

$\sum_{i=0}^{p}(-1)^{(p-i)}\,2^i\binom{p}{i}c$

$= c(2+(-1))^p$ (Binomial Theorem)

= c

$A = \bigcup_{i=0}^{k}[c]$

Within both the leading edge and the inverse leading edge, the constant input is unchanged by the operator. This makes the constant input to be the identity of this system.

3.2 Linear

3.2.1 Input Linear Defined Row Leading Edge

$A = \bigcup_{i=0}^{k}[ai+b]$

By Theorem 1, the pth 0 indexed element of A' should be $\frac{1}{2^p}\sum_{i=0}^{p} a_i\binom{p}{i}$ (Formula 2)

Since $a_i = ai + b$, this can be substituted,

$= \frac{1}{2^p}\sum_{i=0}^{p}(ai+b)\binom{p}{i}$

$= \frac{a}{2^p}\sum_{i=0}^{p} i\binom{p}{i} + \frac{b}{2^p}\sum_{i=0}^{p}\binom{p}{i}$

$= \frac{ap2^{p-1}}{2^p} + b$

$= \frac{ap}{2} + b$

$A' = \bigcup_{i=0}^{k}[\frac{ai}{2} + b]$

3.2.2 Input Linear Defined Row Inverse Leading Edge

$A' = \bigcup_{i=0}[ai + b]$

To express A in terms of elements of A', the pth 0 indexed element of A is expressed as $\sum_{i=0}^{p} (-1)^{(p-i)} 2^i \binom{p}{i} a'_i$ by (Formula 4)

$a_p = \sum_{i=0}^{p} (-1)^{(p-i)} 2^i \binom{p}{i} (ai + b)$
$\sum_{i=0}^{p} (-1)^{(p-i)} 2^i \binom{p}{i} (ai + b)$
Let $f(z) = (2z - 1)^p$
$f(z) = \sum_{i=0}^{p} (-1)^{(p-i)} (2z)^i \binom{p}{i}$
$f'(z) = \sum_{i=0}^{p} i\,(-1)^{(p-i)} 2^i z^{i-1} \binom{p}{i}$
$\frac{d}{dz}(2z - 1)^p = 2p(2z - 1)^{p-1}$(Chain rule)
$\sum_{i=0}^{p} (-1)^{(p-i)} 2^i \binom{p}{i} (ai + b)$
$= (az \sum_{i=0}^{p} i\,(-1)^{(p-i)} z^{i-1} 2^i \binom{p}{i}) + b$ (z = 1) (Section 3.1.2)
$= azf'(z) + b$
$=2apz(2z - 1)^{p-1} + b$
$= 2ap + b$
$A = \bigcup_{i=0}^{k}[2ai + b]$

3.3 Quadratic

3.3.1 Input Quadratic Defined Row Leading Edge

$A = \bigcup_{i=0}^{k}[ai^2 + bi + c]$

By Theorem 1, the pth 0 indexed element of A' should be $\frac{1}{2^p} \sum_{i=0}^{p} a_i \binom{p}{i}$ (Formula 2)

Since $a_i = ai^2 + bi + c$, this can be substituted,
$a'_p = \frac{1}{2^p} \sum_{i=0}^{p} (ai^2 + bi + c) \binom{p}{i}$
$= \frac{a}{2^p} \sum_{i=0}^{p} i^2 \binom{p}{i} + \frac{b}{2^p} \sum_{i=0}^{p} i \binom{p}{i} + \frac{c}{2^p} \sum_{i=0}^{p} \binom{p}{i}$
$= \frac{ap(p+1)2^{p-2}}{2^p} + \frac{b(p+1)2^{p-1}}{2^p} + \frac{c2^p}{2^p}$
$= \frac{ap(p+1)}{2^2} + \frac{b(p+1)}{2^1} + c$
$A' = \bigcup_{i=0}^{k}[\frac{ai^2+i(a+2b)+2b+4c}{4}]$

3.3.2 Input Quadratic Defined Row Inverse Leading Edge

$A' = \bigcup_{i=0}^{k}[ai^2 + bi + c]$

$a_p = \sum_{i=0}^{p} (-1)^{(p-i)} 2^i \binom{p}{i} (ai^2 + bi + c)$ (Formula 4)
It has been shown that when
$f(z) = (2z - 1)^p, f'(z) = 2k(2z - 1)^{p-1}$
Then $f''(z) = 4p(p - 1)(2z - 1)^{p-2}$
$f(z) = (\sum_{i=0}^{p} (-1)^{(p-i)} z^i 2^i \binom{p}{i})$
$f'(z) = (\sum_{i=0}^{p} i\,(-1)^{(p-i)} z^{i-1} 2^i \binom{p}{i})$
$f''(z) = (\sum_{i=0}^{p} i(i - 1)\,(-1)^{(p-i)} z^{i-2} 2^i \binom{p}{i})$
$f''(z) = (\sum_{i=0}^{p} i^2\,(-1)^{(p-i)} z^{i-2} 2^i \binom{p}{i}) - \sum_{i=0}^{p} i\,(-1)^{(p-i)} z^{i-2} 2^i \binom{p}{i})$

$f''(z) = (\sum_{i=0}^{p} i^2 (-1)^{(p-i)} z^{i-2} 2^i \binom{p}{i}) - 2p$ (Section 3.2.2)
$z^2(f''(z) + 2p) = (\sum_{i=0}^{p} i^2 (-1)^{(p-i)} z^i 2^i \binom{p}{i})$
$= z^2(4p(p-1)(2z-1)^{p-2} + 2p)$
$= (4p(p-1) + 2p)$ (z = 1)
$= 4p^2 - 2p$
$a_p = a(4p^2 - 2p) + 2bp + c$ (Section 3.2.2)
$a_p = 4ap^2 + 2p(b-a) + c$
$A = \bigcup_{i=0}^{k} [4ai^2 + 2i(b-a) + c]$

3.4 Exponential

3.4.1 Input Exponential Row Midpoint

$A = \bigcup_{i=0}^{k} [ab^i]$.
$a'_p = \frac{1}{2^p} \sum_{i=0}^{p} ab^i \binom{p}{i}$ (Formula 2)
$= \frac{a}{2^p} \sum_{i=0}^{p} b^i \binom{p}{i}$
$= \frac{a}{2^p} (b+1)^p$ (Binomial Theorem)
$= a(\frac{b+1}{2})^p$
$A' = \bigcup_{i=0}^{k} [a(\frac{b+1}{2})^i]$.

3.4.2 Input Exponential Row Anti Midpoint

Now lets assume that the A' is in the exponential form
$A' = \bigcup_{i=0}^{k} [ab^i]$
To express A in terms of A',
$a_p = \sum_{i=0}^{p} (-1)^{(p-i)} 2^i \binom{p}{i} ab^i$ (Formula 4)
$= \sum_{i=0}^{p} (-1)^{(p-i)} 2^i \binom{p}{i} ab^i$
$= a \sum_{i=0}^{p} (-1)^{(p-i)} (2b)^i \binom{p}{i}$
$= a((2b) + (-1))^p$ (Binomial Theorem)
$= a(2b-1)^p$
$A = \bigcup_{i=0}^{k} [a(2b-1)^i]$

This exponential input array forms the basis for the rest of the sections. For some input array A that is a geometric sequence, the resultant A' will also be a geometric sequence, but with a shifted base.

4 Positive Integer Based Geometric Sequences

Section 3.4 demonstrates that taking the successive midpoint or successive anti midpoints will create a leading edge that is a entirely comprised of sequentially increasing exponentials. This leads to the creation of two additional formulas specifying the initial exponential input in Formula 5 and Formula 6. The following Formula 8 and Formula 9 collapse down the summation logic from the original formulas. This again uses the assumption that the big cup operator creates the ordered union of an array as defined within Section 3. The inputs and outputs of both of these formulas are geometric sequences. Using this methodology, geometric sequences will be created from some arbitrary starting b and will result in an ending base of b'.

$$MidpointLeadingEdge(\bigcup_{i=0}^{n-1}[ab^i]) = \bigcup_{i=0}^{n-1}[a(\frac{b+1}{2})^i] \tag{8}$$

$$MidpointLeadingEdge^{-1}(\bigcup_{i=0}^{n-1}[ab^i]) = \bigcup_{i=0}^{n-1}[a(2b-1)^i] \tag{9}$$

Formula 8 states that taking the successive midpoint upon a geometric sequence of base b will result in a new leading edge series of base $\frac{b+1}{2}$. Formula 9 dictates that taking the successive anti midpoint will create a new leading edge series of base 2b-1. Recursive usages of Formula 8 and Formula 9 could potentially map any starting base b, to a desired resultant base b'. Section 4 will define the techniques that will facilitate this goal.

Yet another successive function must be created, which is the successive application of the successive midpoint and anti midpoints. The relationship between the input base of the geometric series array and the output must be defined. Using Formula 8, it is apparent that for an initial base of b, the resultant base will be $\frac{b+1}{2}$ upon the leading edge when successive midpoints are applied. Via Formula 9, the resultant base from an initial base of b is 2b-1 upon the leading edge for an application of the successive anti midpoint. For both of these relationships, the coefficient 'a' can be neglected as it is unchanged as a linear scalar. The following formulas simply represent distributing the 'a' scalar across the resultant vector.

$$a * MidpointLeadingEdge(\bigcup_{i=0}^{n-1}[b^i]) = \bigcup_{i=0}^{n-1}[a(\frac{b+1}{2})^i] \tag{10}$$

$$a * MidpointLeadingEdge^{-1}(\bigcup_{i=0}^{n-1}[b^i]) = \bigcup_{i=0}^{n-1}[a(2b-1)^i] \tag{11}$$

4.1 Successive Applications and Base 2 Memoization

Both Formulas 10 and 11 demonstrate the distribution of the 'a' scalar when applying MidpointLeadingEdge and its inverse upon a generic geometric series

(ab^k). Since the coefficient can be distributed out of the array, 'a' will be just assumed to be disregarded for the calculations within this section without loss of generality a = 1.

4.1.1 Successive Application of Midpoint Leading Edge and Inverse Midpoint Leading Edge

Since a single application of MidpointLeadingEdge transforms the geometric sequence base $f(b) = \frac{b+1}{2} = b'$, assuming f to be a generic function defined only for the scope of this sub section. How would the transformation of additional applications of this function correspond with respect to the initial base 'b'?

n = 2, representing a two successive applications of the transformation.

$f(f(b)) = f(\frac{b+1}{2}) = \frac{\frac{b+1}{2}+1}{2} = \frac{b+1+2}{2*2}$

n = 3

$f(f(f(b))) = f(\frac{b+1+2}{2*2}) = \frac{\frac{b+1+2}{2*2}+1}{2} = \frac{b+1+2+4}{2*2*2}$

Clearly there is a pattern emerging of the following form, where n represents the recursion depth.

$f(b,n) = \frac{b+\sum_{i=0}^{n-1}(2^i)}{2^n} = \frac{b+(2^n-1)}{2^n} = \frac{b-1}{2^n} + 1$

$$MidpointBaseChange(b,n) = \frac{b-1}{2^n} + 1 \tag{12}$$

Proof. Formula 12

Proof by induction.

Base case.

n = 1, equivalent to simply taking a single execution of the base change $f(b) = \frac{b+1}{2}$

$MidpointBaseChange(b,1)$

$= \frac{b-1}{2^1} + 1$

$= \frac{b-1+2}{2}$

$= \frac{b+1}{2} = f(b)$

Inductive step.

Assuming that when n = k, up the the k^{th} iteration, $MidpointBaseChange(b,k) = \frac{b-1}{2^{k+1}} + 1$.

The k+1^{th} iteration should be equivalent to $MidpointBaseChange(b, k+1) = \frac{b-1}{2^{k+2}} + 1$

This can be calculated by applying f to the k^{th} iteration.

$f(MidpointBaseChange(b,k))$

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

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

$= \frac{\frac{b-1+2*2^{k+1}}{2^{k+1}}}{2}$

$= \frac{b-1+2^{k+2}}{2^{k+2}}$

$= \frac{b-1}{2^{k+2}} + 1$

For the base change of the MidpointLeadingEdge inverse, a similar approach can be taken to find the resultant pattern of the base change function $g(b) = 2b - 1$. However this is unnecessary since it is already known that the base resulting from f can be undone by applying g onto it. This means that $g = f^{-1}$ from the inverse proof.

Since $MidpointBaseChange(b, n) = \frac{b-1}{2^n} + 1 = b'$, simply solving for b should produce the inverse function.

$b' = \frac{b-1}{2^n} + 1$

$2^n(b' - 1) + 1 = b$

This means that $MidpointBaseChange^{-1}(b, n) = 2^n(b-1)+1$. This appears to be a new formulation although upon inspecting the terms it appears to be almost identical to its inverse MidpointBaseChange with the only difference being the power of the n within the 2^n.

Therefore it can be concluded that the inverse of MidpointBaseChange is equivalent to itself with an inverse sign n.

$MidpointBaseChange^{-1}(b, n) = MidpointBaseChange(b, -n) = b'$.

This makes perfect intuitive sense since if it takes n steps to reach b' from b, then it will take n steps backwards (-n) to reach b from b'.

4.1.2 Successive Application of Midpoint Leading Edge and Inverse Midpoint Leading Edge Geometric Sequences

Using the formal constructions from these past sections, a simple set of geometric sequences can be created. Using MidpointBaseChange, it is possible to generate these exponential sequences by defining only two free variables n, b, or b'.

n := integer defining the desired number of steps

b := real value defining the initial base for the input geometric sequence array

b' := the resultant base for the output geometric sequence array after n applications of the leading edge formula.

The following value k, must always be defined.

k := positive integer defining the highest desired exponent to be calculated

The following formula unifies both the leading edge formula and its inverse. This is done by utilizing the extensions created within Section 4.1.1 to form this generalized formula of geometric sequences.

$$SuccessiveLeadingEdge(b, n, k) = \bigcup_{i=0}^{k} [(\frac{b-1}{2^n} + 1)^i] \tag{13}$$

Example 3. Random application of Formula 13

Let b= -4, n=3, k=2.

This will form the following geometric sequence. $[1, \frac{-4-1}{2^3} + 1, (\frac{-4-1}{2^3} + 1)^2]$ $=[1, \frac{3}{8}, \frac{9}{64}]$.

Using the formula to find the resultant geometric sequence is the easy way. It can still be calculated manually by using successive implementations of the leading edge formulas.

Since n is positive, it is equivalent to using the successive midpoints upon the inputs.

This will require 3 applications of the successive midpoint formula, the 0^{th} application corresponds to generating the inputs, k=2 b=-4 $[(-4)^0,(-4)^1,(-4)^2]$=[1,-4,16].

n=1

$$\begin{matrix} 1 & -4 & 16 \\ -1.5 & 6 & 0 \\ 2.25 & 0 & 0 \end{matrix}$$

n=2

$$\begin{matrix} 1 & -1.5 & 2.25 \\ -.25 & .375 & 0 \\ .0625 & 0 & 0 \end{matrix}$$

n=3

$$\begin{matrix} 1 & -.25 & .0625 \\ .375 & -.09375 & 0 \\ .140625 & 0 & 0 \end{matrix}$$

Which correctly generated the geometric sequence on the final leading edge.

However interesting it may be to generate random geometric sequences by just taking a few midpoints, defining the resultant basis b' is more useful. This frees up either the initial base b or n number of iterations. For now, the base b will be set, leaving n to be free.

$b' = \frac{b-1}{2^n} + 1$ (Solving for n)

$2^n = \frac{(b-1)}{(b'-1)}$

$n = log_2(\frac{(b-1)}{(b'-1)})$

Since the logarithm has been introduced, n might not be an integer for all input values of b and b'. This is problematic since the algorithm must be able to be reliable in generating all bases. The variable n requires an integer constraint so it might be more helpful to have n be fixed while b is free to vary.

$b' = \frac{b-1}{2^n} + 1$ (Solving for b)

$2^n(b' - 1) + 1 = b$

Example 4. n = 10, b' = 10000

$b = 2^{10}(10000 - 1) + 1$

=10238977

It is theoretically practical to pursue a base generation such as this, but it fails to provide any real benefits. The geometric sequence of the initial base would have to be calculated for each new generation. The starting base might even be larger than the desired final base as shown in Example 4. Arbitrarily defining a number of iterations is not intuitive and can be set to any integer value which will shift the starting base. A new strategy must be devised in order to use an easily calculable geometric sequence that will produce a fixed number of integer iterations.

The successive leading edge function is available at 'Section_4/successive_leading_edge.py' or within the command line program at 'Leading Edge Functions/-Successive Leading Edge'.

The inverse successive leading edge function is available at 'Section_4/inverse_successive_leading_edge.py' or within the command line program at 'Leading Edge Functions/Inverse Successive Leading Edge'.

The formulaic leading edge function (Formula 13) is available at 'Section_4/successive_leading_edge_formulaic.py' or within the command line program at 'Leading Edge Functions/Successive Leading Edge Formulaic'.

4.2 Logarithmic Time Memoization

This section will define an encoding technique that standardizes the initial base to 2 while providing a fixed memoization instruction set used to calculate a desired positive integer basis b' greater than 2. This base of 2 is selected as the initial geometric sequence input as an easily calculable sequence of bit shifts that can be reused for multiple calculations.

Since the desired output basis b' is a positive integer greater than 2, the leading edge generated by successive iterations of the anti midpoint function should be utilized. However it is not enough to simply utilize Formula 13 by setting b to 2. A new function must be devised to be able to utilize the successive anti midpoint function to yield a generic basis b'. Since the anti midpoint function when applied to a base b and 1 results in 2b - 1 = c, this function will be used to build up from 2. The anti midpoint of b and 1 is utilized to define the b' since the first element of a geometric sequence of this nature will always be 1. However since the inverse of this function is exactly equivalent to the midpoint of c and 1, $\frac{c+1}{2} = b$. Since c is a positive integer, b will always be either an integer or and integer added to one half. This behavior is resultant from the parity of c. Noting this parity of c as a boolean value bool(c mod 2) which is equivalent to finding the parity of b, $f(c) = \lfloor \frac{c+1}{2} \rfloor = b$, 1 for even 0 for odd. The function allows for c to be recovered either by performing an anti midpoint between b and 1, or simply multiplying by 2. This will form a complete instruction set that will allow for the full recover of the desired basis b' from an initial basis of 2. It now must be proven that successive iterations of this function will result in 2 for any input. This is to ensure that for any input b', there exists a valid sequence of steps that can be result from 2 back to the original b'.

Lemma 2. $\lfloor \frac{c+1}{2} \rfloor = \lceil \frac{c}{2} \rceil$

This is a basic analytical proof. Since c is a positive integer its parity is either even or odd.

Case 1: c = 2k, k is an integer, c is even.

$\lfloor \frac{c+1}{2} \rfloor$

$= \lfloor \frac{2k+1}{2} \rfloor$

$= \lfloor k + \frac{1}{2} \rfloor$

$= k$

This is equivalent to $\lceil \frac{c}{2} \rceil = \lceil \frac{2k}{2} \rceil = k$

Case 2: c = 2k+1, k is an integer, c is odd.
$\lfloor \frac{c+1}{2} \rfloor$
$= \lfloor \frac{2k+1+1}{2} \rfloor$
$= \lfloor \frac{2k+2}{2} \rfloor$
$= k+1$
This is equivalent to $\lceil \frac{c}{2} \rceil = \lceil \frac{2k+1}{2} \rceil = \lceil k + \frac{1}{2} \rceil = k+1$
Therefore $\forall c \in \mathbb{N}, \lfloor \frac{c+1}{2} \rfloor = \lceil \frac{c}{2} \rceil$

Proof. Successive Applications $\lfloor \frac{c+1}{2} \rfloor$ will eventually equal 2.

It will be more computationally efficient to perform an equivalent statement $\lfloor \frac{c+1}{2} \rfloor = \lceil \frac{c}{2} \rceil$ (Lemma 2)

Let $f(x) = \lceil \frac{x}{2} \rceil$. $f_n(x)$, equates to the composition of f with itself n time.

$f_1(x) = f(x), f_2(x) = \lceil \frac{\lceil \frac{x}{2} \rceil}{2} \rceil$

Claim 2: $f_n(x) = \lceil \frac{x}{2^n} \rceil$

Proof By Induction

Base Case n = $1 \lceil \lceil \frac{x}{2^1} \rceil \rceil = \lceil \frac{x}{2} \rceil$

Inductive Case Assume n = k is true, $f_k(x) = \lceil \frac{x}{2^k} \rceil$ then $f_{k+1}(x) = \lceil \frac{x}{2^{k+1}} \rceil$, must be show to be true.

$\lceil \frac{x+m}{n} \rceil = \lceil \frac{\lceil x \rceil + m}{n} \rceil$

By utilizing Theorem 3.11 from Graham, Knuth and Patashnick, p.72.

[3]

$\lceil \frac{x+m}{n} \rceil = \lceil \frac{\lceil x \rceil + m}{n} \rceil$

Let m = 0 and n = 2^k

$f_k(x) = \lceil \frac{x}{2^k} \rceil = \lceil \frac{\lceil x \rceil}{2^k} \rceil$

$f_k(\frac{x}{2}) = f_{k+1}(x) = \lceil \frac{\lceil \frac{x}{2} \rceil}{2^k} \rceil$

Since 2^k is a positive integer,

$\lceil \frac{\lceil \frac{x}{2} \rceil}{2^k} \rceil = \lceil \frac{x}{2^{k+1}} \rceil$

Since Claim 2 has been shown to be true Claim 1,$f_n(x)$ will always equate to 2, can be shown to be true.

$\lim_{n \to \infty^+} f_n(x)$
$= \lim_{n \to \infty^+} \lceil \frac{x}{2^n} \rceil$

When approaching from the positive side, clearly the inner $\frac{x}{2^n}$ will converge to 0. However since the inner $\frac{x}{2^n}$ will always be slightly above 0 when approaching from this side, the outer ceiling will always round up to 1 once it reaches this point.

This means that the memoization must always terminate once 1 is reached.

The only way to get to a b = 1 is from a c = 1 or 2. $\lceil \frac{c}{2} \rceil = b$. Therefore a value of 2 must will always be a step within every memoization. Additionally, the memoization will always start with 0. Denoting the initial scale from 1 to 2.

his allows for two inverse algorithms to be created. First an initial memoization function that is able to take in a b' and output the memoization bit string that will correspond to the anti midpoint and scale by two operations that are necessary to recover b'. The inverse of this function can also be created

which will take a memoization bit string as input and return the desired b', by following the recovery functions. Both of these algorithms run in logarithmic time, $O(lg(b))$ since the repeated division by two is equivalent to lg(b). This is due to $\lceil \frac{b}{2^{n+1}} \rceil \approx \frac{b}{2^{n+1}}$

$\frac{b'}{2^{n+1}} = 1$

$b' = 2^{n+1}$

$log_2(b') - 1 = n$

Since n represents the number of iterations of f or the number of elements within the memoization, $O(log_2(b'))$. This means that the length of the output bit string will be $\lceil log_2(b') \rceil$ which rounds to the nearest inclusive integer.

The log time memoization function is available at 'Section_4/positive_integer_log_memoizat or within the command line program at 'Memoization Functions/Positive Integer Log Memoization'.

The log time memoization recovery function is available at 'Section_4/positive_integer_log_memoization_base_reconstruction.py' or within the command line program at 'Memoization Functions/Positive Integer Log Memoization Base Reconstruction'.

4.3 Constant Time Memoization

The running time of the memoization technique from Section 4.2 does run in logarithmic time which does not represent a significant bottleneck within the overall running time complexity of a proposed universal geometric sequence generation algorithm where a single iteration would be in $O(k^2)$. This unified algorithm would have an running time of O($k^2 log_2(b)$)since the memoization has been show to be of length $\lceil log_2(b') \rceil$. There is an easy substitution that can be made to alter this log time memoization technique to run in constant time. This would not change the running time of the full solver algorithm since the memoization execution would still be of the same length, but it does solidify this memoization into a unique base encoding.

4.3.1 Unique Base Encoding

By analyzing the resultant memoizations produced by the log time memoization technique, it is apparent that these bit strings uniquely encode the input base b. If the encoding transformation can be identified then a simple function can be produced to to replace the log time memoization.

$f(s, i) = 2s - m_i$ where m_i represents the ith element within the memoization and s represents the initial starting point of the progression will be used to represent the result from each stage of the memoization. Since the 0 bit represent multiplication by 2 and the 1 bit represents multiplication by 2 followed by subtraction by 1, this formula correctly outputs the stage. The succession of this algorithm proceeds as follows.

$f(s, 0) = 2s - m_0$

$f(f(s, 0), 1) = 2(2s - m_0) - m_1 = 4s - 2m_0 - m_1$

$f(f(f(s, 0), 1), 2) = 2(4s - 2m_0 - m_1) - m_2 = 8s - 4m_0 - 2m_1 - m_2$

Clearly another pattern is emerging involving powers of 2.

Since the input array for this section's progression begins with powers of 2, s = 1 with $m_0 = 0$. s = 1 aligns $\lim_{n\to\infty}\lceil b'/2^n\rceil = 1$. Additionally, each memoization will begin with a 0 as its first element as shown within Section 4.2.

It appears that the recursive applications of f, is represented by. $b = 2^{\lceil log_2(b)\rceil} - \sum_{i=0}^{\lceil log_2(b)\rceil - 1} 2^{\lceil log_2(b)\rceil - 1 - i} m_i$

Solving for the summation transforms the formula into $\sum_{i=0}^{\lceil log_2(b)\rceil - 1} 2^{\lceil log_2(b)\rceil - 1 - i} m_i = 2^{\lceil log_2(b)\rceil} - b$.

The summation term clearly is the decimal representation of the memo table where m_0 is the most significant bit. This means that converting $2^{\lceil log_2(b)\rceil} - b$ to base 2 will result in the binary encoding of the memo table with the most significant bit needing to be read first.

$$ConstantTimeMemoization(b) = (2^{\lceil log_2(b)\rceil} - b)_2 \tag{14}$$

Proof. Formula 14

Proving Formula 14 is equivalent to proving the recursive relation $f(s,n) = b = s2^{n+1} - \sum_{i=0}^{n} 2^{n-i} m_i$. where $\lceil log_2(b)\rceil$ = n+1 which is an argument to f.

Proof by induction.

Base Case. First iteration of f, n = 0,

$f(s,0) = 2s - m_0$

$= s2^1 - \sum_{i=0}^{0} 2^{0-i} m_i$

$= 2s - m_0$

Inductive case. Assuming the kth iteration of f = $f(f(f(...f(s,0)...)), k-1), k) = s2^{k+1} - \sum_{i=0}^{k} 2^{k-i} m_i$

So the $k+1^{th}$ iteration should be equivalent to $s2^{k+2} - \sum_{i=0}^{k+1} 2^{k+1-i} m_i$.

Therefore $f(f(f(...f(s,0)...)), k), k+1) = 2(s2^{k+1} - \sum_{i=0}^{k} 2^{k-i} m_i) - m_{k+1}$

$= s2^{k+2} - \sum_{i=0}^{k} (2^{k+1-i} m_i) - m_{k+1}$

$s2^{k+2} - \sum_{i=0}^{k+1} 2^{k+1-i} m_i$ (Absorbing term)

This allows for memoization to be completed in constant time using Formula 14. This constant time complexity does depend on the efficiency of the underlying implementation of $\lceil log_2(b)\rceil$ and the ease of access of the binary representation of the difference $2^{\lceil log_2(b)\rceil} - b$.

Putting the memoization, together with the successive anti midpoint leading edge generation allows for a complete positive integer geometric sequence solver to be constructed. Starting from only array of 1's, the geometric sequence of any positive integer base can be constructed through only multiplying by 2 and subtractions.

The constant time memoization encoding is available at 'Section_4/positive_integer_constant or within the command line program at 'Memoization Functions/Positive Integer Constant Memoization'.

The full positive integer geometric sequence solver is available at 'Section_4/positive_integer_sequence.py' or within the command line program at 'Geometric Sequence Solvers/Positive Integer Sequence'.

5 Binomial Convolution on a 1D exponential signal

It has been shown that applying the successive midpoint operations is equivalent to a weighted average of scaled binomial coefficients. For the case of an exponential input the following represents the exact solution for the successive midpoint applied k times across an input of length k $\frac{1}{2^k}\sum_{i=0}^{k} b^i\binom{k}{i}$. The initial input k is simply a geometric sequence of base b that has been transformed by the cumulative sum of binomial coefficients. This section will view this input geometric sequence not as simply an array, but as a one dimensional signal in the time domain. This will be specified for signals of exponentials in both the discrete and continuous spaces.

5.1 Weighted Sum to Convolution

Taking the simplified successive midpoint formula $\frac{1}{2^k}\sum_{i=0}^{k} b^i\binom{k}{i}$ is the following construction $\frac{1}{2^k}[b^0, b^1, b^2..., b^k][\binom{k}{0}, \binom{k}{1}...\binom{k}{k}]'$. This linear combination is not immediately in the form of a convolution, but the application is equivalent.

Proof. Representing the successive midpoint formula as a convolution
$\frac{1}{2^k}\sum_{i=0}^{k} b^i\binom{k}{i}$
$=\frac{1}{2^k}\sum_{i=0}^{k} b^i\frac{k!}{i!(k-i)!}$
$=k!\sum_{i=0}^{k}\frac{b^i}{2^k i!(k-i)!}$
$=k!\sum_{i=0}^{k}\frac{b^i}{2^i 2^{k-i} i!(k-i)!}$
$=k!\sum_{i=0}^{k}(\frac{1}{i!}(\frac{b}{2})^i)(\frac{1}{(k-i)!}(\frac{1}{2})^{k-i})$

Now this can be represented by the following Cauchy product (removing the scale factor k!).

$\sum_{i=0}^{k}(\frac{1}{i!}(\frac{b}{2})^i)\sum_{i=0}^{k}(\frac{1}{(k-i)!}(\frac{1}{2})^{k-i})$

Let $f(x) = (\frac{1}{x!}(\frac{b}{2})^x)$ and $g(x) = (\frac{1}{x!}(\frac{1}{2})^x)$ which are discrete functions over the interval $[0, k]$.

Therefore the convolution between these function, $f * g = \sum_{i=0}^{k} f(i)g(k-i)$

$$\frac{1}{2^k}\sum_{i=0}^{k} b^i\binom{k}{i} = k!(f*g) = k!\sum_{i=0}^{k} f(i)g(k-i), f(x) = (\frac{1}{x!}(\frac{b}{2})^x), g(x) = (\frac{1}{x!}(\frac{1}{2})^x) \tag{15}$$

An interesting result from this follows from the f and g functions that are being convolved. Both of these functions are almost identical and follow this relationship $f(x) = b^x g(x)$. Now before running through the calculations for the anti midpoint convolution from this initial sum $\sum_{i=0}^{k}(-1)^{(k-i)}\,2^i\binom{k}{i}b^i$, it is immediate to tell that the $(-1)^{k-i}$ would end up within the g(x) function for the resultant convolution while the 2^i might end up within the f with what might be an even more straight forward proof than the previous.

Proof. Anti midpoint convolution
$\sum_{i=0}^{k} (-1)^{(k-i)} 2^i \binom{k}{i} b^i$
$=\sum_{i=0}^{k} (-1)^{(k-i)} 2^i \frac{k!}{i!(k-i)!} b^i$
$=k! \sum_{i=0}^{k} (\frac{(2b)^i}{i!})(\frac{(-1)^{k-i}}{(k-i)!})$
For this function $f(x) = \frac{(2b)^x}{x!}$, $g(x) = \frac{(-1)^x}{x!}$

This raises the question of some sort of generic class of transformations that would not be represented by a midpoint or anti midpoint, but rather something with generalized weights c_0, c_1. These weights would be applied to the input in opposite order since, in the anti midpoint case the 2^i is applied to the second element when the $(-1)^i$ is applied to the first element, $-1a_0 + 2a_1$.

$$GeneralizedBinomialTransformation(A, c_0, c_1) = \sum_{i=0}^{k} c_0^{k-i} c_1^i \binom{k}{i} a_i, k = |A|-1 \tag{16}$$

This would represent essentially the successive application of this generic element wise transformation $c_1a_0 + c_0a_1$ similarly to how the array wise midpoint was constructed.

Proof. Successive applications of general linear transformations $c_0a_0 + c_1a_1$ of two elements results in formula 16

Proof by induction.

Base case k=1, should result in $c_1a_0 + c_0a_1$
$\sum_{i=0}^{k} c_0^i c_1^{k-i} \binom{k}{i} a_i$
$=c_0^0 c_1^{1-0} \binom{1}{0} a_0 + c_0^1 c_1^{1-1} \binom{1}{1} a_1$
$=c_1a_0 + c_0a_1$

Induction step, |A| = k+1 then assuming the generic transformation applied to A[0:k-1] = $\sum_{i=0}^{k-1} c_0^i c_1^{k-1-i} \binom{k-1}{i} a_i$ and the generic transformation applied to A[1:k] = $\sum_{i=1}^{k} c_0^{i-1} c_1^{k-i} \binom{k-1}{i-1} a_i$, the it must be shown that the generic transformation applied to A[0:k] =$\sum_{i=0}^{k} c_0^i c_1^{k-i} \binom{k}{i} a_i$

Applying the generic transformation upon the successive generic transformations of the array slices $c_1(\sum_{i=0}^{k-1} c_0^i c_1^{k-1-i} \binom{k-1}{i} a_i) + c_0(\sum_{i=1}^{k} c_0^{i-1} c_1^{k-i} \binom{k-1}{i-1} a_i)$
$=\sum_{i=0}^{k-1} (c_0^i c_1^{k-i} \binom{k-1}{i} a_i) + \sum_{i=1}^{k} (c_0^i c_1^{k-i} \binom{k-1}{i-1} a_i)$
$c_0^0 c_1^{k-0} \binom{k-1}{0} a_0 + \sum_{i=1}^{k-1} (c_0^i c_1^{k-i} \binom{k-1}{i} a_i) + \sum_{i=1}^{k-1} (c_0^i c_1^{k-i} \binom{k-1}{i-1} a_i) + c_0^k c_1^{k-k} \binom{k-1}{k-1} a_k$(Reindexing)
$=c_1^k a_0 + \sum_{i=1}^{k-1} (c_0^i c_1^{k-i} \binom{k-1}{i} a_i) + \sum_{i=1}^{k-1} (c_0^i c_1^{k-i} \binom{k-1}{i-1} a_i) + c_0^k a_k$
$=c_1^k a_0 + \sum_{i=1}^{k-1} ((\binom{k-1}{i} + \binom{k-1}{i-1})(c_0^i c_1^{k-i} a_i)) + c_0^k a_k$
$=c_1^k a_0 + \sum_{i=1}^{k-1} ((\binom{k-1}{i} + \binom{k-1}{i-1})(c_0^i c_1^{k-i} a_i)) + c_0^k a_k$
$=\sum_{i=0}^{k} \binom{k}{i} c_0^i c_1^{k-i} a_i$ (Pascal's Identity)

Now for the convolution transformation of this generic transformation is of the following form applied to a generic input.

Proof. Generic binomial transformation expressed as a convolution applied to a generic input.

$\sum_{i=0}^{k} c_0^{k-i} c_1^i \binom{k}{i} a_i$

$=\sum_{i=0}^{k} c_0^{k-i} c_1^i \frac{k!}{i!(k-i)!} a_i$

$=k! \sum_{i=0}^{k} (\frac{c_1^i a_i}{i!})(\frac{c_0^{k-i}}{(k-i)!})$

Let $f(x) = \frac{c_1^x a_x}{x!}$, g(x)$=\frac{c_0^x}{x!}$

So the convolution is expressed as

$$\sum_{i=0}^{k} c_0^{k-i} c_1^i \binom{k}{i} a_i = k!(f * g) = k! \sum_{i=0}^{k} f(i)g(k-i), f(x) = (\frac{c_1^x a_x}{x!}), g(x) = (\frac{c_0^x}{x!}) \tag{17}$$

Therefore the generic binomial transformation has been shown to be true so clearly the successive midpoint and anti midpoint functions are just specific elements of this function class. For the successive midpoint, $c_0 = c_1 = \frac{1}{2}$ and for the successive anti midpoint $c_0 = -1, c_1 = 2$.

5.2 Discrete Fourier Transform To Generate Geometric Sequences

5.2.1 Spatial To Frequency Domain

Now that the generic binomial transformation formula has be converted into a convolution expression between two function, a Fourier transformation can be attempted to gain efficiency in this algorithm.

Proof. Applying the Discrete Fourier Transform DFT $X(n) = \sum_{h=0}^{N-1} x(h) e^{\frac{-2\pi i h n}{N}}$

$F(n) = \sum_{h=0}^{N-1} (\frac{c_1^h a_h}{h!}) e^{\frac{-2\pi i h n}{N}}$,$G(n) = \sum_{h=0}^{N-1} (\frac{c_0^h}{h!}) e^{\frac{-2\pi i h n}{N}}$ in the generic a_n case.

For an input signal of a geometric sequence of base b however it can be written as follows.

$F(n) = \sum_{h=0}^{N-1} \frac{(c_1 b e^{\frac{-2\pi i n}{N}})^h}{h!}$,$G(n) = \sum_{h=0}^{N-1} \frac{(c_0 e^{\frac{-2\pi i n}{N}})^h}{h!}$ within the frequency domain.

Therefore the convolution $k!(f * g)$, $\mathcal{F}(k!(f * g)) = k!F(n)G(n)$

$=k!(\sum_{h=0}^{N-1} \frac{(c_1 b e^{\frac{-2\pi i n}{N}})^h}{h!})(\sum_{h=0}^{N-1} \frac{(c_0 e^{\frac{-2\pi i n}{N}})^h}{h!})$

$=k!(\sum_{h=0}^{N-1} \frac{(c_1 b z)^h}{h!})(\sum_{h=0}^{N-1} \frac{(c_0 z)^h}{h!})$ (Let $z = e^{\frac{-2\pi i n}{N}}$)

Clearly by taking $N \to \infty$, with a large enough sampling rate, the sums approach what would be the Maclaurin series of some shifted exponential.

$e^{qx} = \sum_{h=0}^{\infty} \frac{g x^h}{h!}$

So $\sum_{h=0}^{\infty} \frac{(c_1 b z)^h}{h!} = e^{c_1 b z}$ $(q = c_1 b)$.

$\sum_{h=0}^{\infty} \frac{(c_0 z)^h}{h!} = e^{c_0 z}$ $(q = c_0)$

Therefore $\lim_{N \to \infty} k!(\sum_{h=0}^{N-1} \frac{(c_1 b z)^h}{h!})(\sum_{h=0}^{N-1} \frac{(c_0 z)^h}{h!}) = k! e^{c_1 b z} e^{c_0 z}$

$=k!e^{z(c_1b+c_0)}$

$X(n) = k!e^{(c_1b+c_0)e^{\frac{-2\pi in}{N}}}$ which is calculated over some sampling range N.

5.2.2 Frequency To Time Domain

Proof. Applying the Discrete Fourier Transform DFT $x(n) = \frac{1}{N}\sum_{h=0}^{N-1} X(h)e^{\frac{2\pi ihn}{N}}$

$X(n) = k!e^{(c_1b+c_0)e^{\frac{-2\pi ihn}{N}}}$

$= X(n) = k!\frac{1}{N}\sum_{j=0}^{N-1} \frac{(c_1b+c_0)^j}{j!}e^{\frac{-2\pi ihn}{N}}$ since $(e^{qx} = \sum_{h=0}^{\infty} \frac{qx^h}{h!})$

$\frac{k!}{N}\sum_{h=0}^{N-1}\sum_{j=0}^{N-1} \frac{(c_1b+c_0)^j}{j!}e^{\frac{-2\pi ihn}{N}}$(IDFT)

$=\frac{k!}{N}\sum_{j=0}^{N-1} \frac{(c_1b+c_0)^j}{j!}\sum_{h=0}^{N-1} e^{\frac{-2\pi ih(n-j)}{N}}$ (Swapping Sums)

$\sum_{h=0}^{N-1} e^{\frac{-2\pi ih(n-j)}{N}} = \begin{cases} N & n=j \\ 0 & else \end{cases}$ Due to the orthogonality of the roots of unity.

$x(n) = k!\frac{(c_1b+c_0)^n}{n!}$

Since when n=k, satisfies this condition.

$x(n) = (c_1b + c_0)^n$

Therefore this shows that for the successive midpoint where $c_1 = c_0 = \frac{1}{2}$, $x(n) = (\frac{b+1}{2})^n$ which falls in line with the expected value.

This full loop does represent a time complexity increase from $O(n^2)$ where every single row of an n x n cascade matrix would have to be operated upon to $O(nlnn)$ by using Fourier Transforms to represent the multiplication of the functions in the frequency domain. Although this method is quite useful for finding a midpoint leading edge. Essentially in order to calculate that final exponent value N - 1, you need to iterate k though the other values up to N - 1as well. This therefore generates the leading edge in a similar fashion to how the matrix in the time domain would find the leading edge upon the 0^{th}column.

5.3 Discrete Fourier Transform To Generate Successive Leading Edges

The methods defined within 7.2 have shown that it is possible to generate a leading edge in a more efficient time complexity than simply taking successive midpoints, successive anti midpoints, or even a general linear transformation of two terms. However this only is able to generate one iteration, which is not sufficient for generating any arbitrary geometric series using only midpoints and anti midpoint constructions as has been previously defined. The Discrete Fourier methods as proposed in Section 5.2 must be extended to involve the usage of the memoization supplied by ConstantTimeMemoization or LogTimeMemoization.

It can be naively assumed that the approach to take would be simply take these following steps:

1. Take in some sort of input array of length n

2. Perform the Discrete Fourier Transform upon it
3. Retrieve the leading edge
4. Transform it back into the time domain
5. Either multiple the entire leading edge by increasing powers of 2 or return to step one until the memoization is complete

Since it has been shown that the length of the memoization is $log_2 b$ where b is the desired base of the final output geometric series (b^n), this would yield an overall complexity of $O(lg(b)nlg(n))$. This still provides an efficient optimization from the overall previous running time of $O(lg(b)n^2)$, but the memoization can be used within the frequency domain to provide an even further optimization.

To recall, the memoization is simply a bit string in which 1 corresponds to repeating the main successive $c_0a_1 + c_1a_0$ transformation again to create a leading edge and 0 corresponds to multiplying each element in the leading edge element wise by an increasing geometric sequence of powers of 2.

5.3.1 Memoization 0 Operation

Within the time domain, a memoization code of 0 corresponds to the following. $b^k \to (2b)^k$ forming a new sequence

Corollary. *Finding the frequency operation corresponding to the spatial* $b^k \to (2b)^k$

As was discussed in Section 5.1, the function involved in the convolution involving the b term was $f(x) = (\frac{(bc_1)^x}{x!})$, $g(x) = (\frac{c_0^x}{x!})$. *However for this function, clearly* $c_1 = 2, c_0 = 0$ *since this operation simply involves multiplying by a power of two. This is however similar to the anti midpoint which is* $b^k \to (2b-1)^k$ *where* $c_0 = -1, c_1 = 2$

Using the proof found within Section 5.2.1,$X(n) = k!e^{(c_1b+c_0)e^{\frac{-2\pi in}{N}}}$

Therefore the frequency domain representation of the multiplication by an increasing power of two is the following.

$X_{(2b)^k}(n) = k!e^{2be^{\frac{-2\pi in}{N}}}$

This however does differ from what simply b^k *would be within the frequency domain.*

The transformation $b^k \to (1b+0)^k$ *quite obviously has* $c_1 = 1, c_0 = 0$.

Plugging this into the formula for X(n),

$X_b(n) = k!e^{be^{\frac{-2\pi in}{N}}}$

The k! can be added back late. Without loss of generality, $\Phi(n) = e^{(c_1b+c_0)e^{\frac{-2\pi in}{N}}} = \frac{X(n)}{k!}$ *represents a normalized base, but for each successive operation upon it a new k! scaler will be added retroactively in the end. If the k! is ignored then therefore the transformation in the frequency domain corresponding to that of the multiplying by a power of two is simply squaring.*

$\Phi_{(2b)^k}(n) = (\Phi_b(n))^2$

This is an interesting conclusion and allows for the '0' instruction of the memoization to be applied in the frequency domain simply by squaring the frequency.

5.3.2 Memoization 1 Operation

The construction for the anti midpoint within the frequency domain has already been defined within the previous section however by using $c_0 = -1, c_1 = 2$,

$\Phi_{2b-1}(n) = e^{(2b-1)e^{\frac{-2\pi in}{N}}}$
$= e^{2be^{\frac{-2\pi in}{N}}} e^{-e^{\frac{-2\pi in}{N}}}$
$= \Phi_{2b}(n)\Phi_{-1}(n)$
$= (\Phi_b(n))^2\Phi_{-1}(n)$
$= \frac{(\Phi_b(n))^2}{\Phi_1(n)}$

This clearly demonstrates the efficiency that can be yielded by using the frequency domain rather than the time domain. The successive anti midpoint is the same as the memoization operation, but with a scale factor.

As a potential point of interest, determining the frequency domain operation that will correspond to the successive midpoint $c_0 = c_1 = \frac{1}{2}$,

$\Phi_{.5b+.5}(n) = e^{(.5b+.5)e^{\frac{-2\pi in}{N}}}$
$=e^{\frac{b}{2}e^{\frac{-2\pi in}{N}}} e^{\frac{1}{2}e^{\frac{-2\pi in}{N}}}$
$=\sqrt{\Phi_b(n)\Phi_1(n)}$

This is slightly more tricky due to the presence of the branching square root and the complex numbers, however it still does represent clearly what would be the inverse to the above anti midpoint spectral formula. Plugging in the anti midpoint construction for $\Phi_b(n)$ yields $\sqrt{\frac{(\Phi_b(n))^2}{\Phi_1(n)}\Phi_1(n)}$, which simplifies to what would be expected to be the unit within this system $\Phi_b(n)$.

5.3.3 Encoding full memoization within the frequency domain

It has been defined what the analog of both memoization operations utilizing the spectral representations of these functions. A new construction can be created that processes the entire memoization into what would be a single operation using the following formulation.

$$RecursiveSpectralBaseBuilder(b,n) = \begin{cases} (\Phi_b(n))^2 & 0 \\ \frac{(\Phi_b(n))^2}{\Phi_1(n)} & 1 \end{cases} \tag{18}$$

This allows for a bit string memoization to be built up in a recursive fashion.

Example 5. Bit string 01100
bit = 0
$(\Phi_b(n))^2$
bit = 1
$\frac{((\Phi_b(n))^2)^2}{\Phi_1(n)} = \frac{(\Phi_b(n))^4}{\Phi_1(n)}$

bit = 1
$\frac{(\frac{(\Phi_b(n))^4}{\Phi_1(n)})^2}{\Phi_1(n)} = \frac{(\Phi_b(n))^8}{(\Phi_1(n))^3}$
bit = 0
$(\frac{(\Phi_b(n))^8}{(\Phi_1(n))^3})^2 = \frac{(\Phi_b(n))^{16}}{(\Phi_1(n))^6}$
$=\Phi_{16b-6}(n)$
Clearly there appears to be a pattern here which appears to actually be quite interesting and elegant if true.

$$SpectralBaseBuilder(memo, b) = \Phi_{2^{|memo|}b-memo_{10}}(n) \tag{19}$$

The memo parameter represents the memoization bit string. |memo| represents the length of the bit string. $memo_{10}$ is the representation of the memo bit string in decimal and b is the input base.

Proof. Formula 19

The actual definition of the memoization in Formula 14, is the following $(2^{\lceil log_2(b')\rceil} - b')_2$ which is dependent on b' the output base. So this spectral function can be reconfigured to incorporate this feature. For this construction the base case input b has been assumed to be b = 1 which falls in line with the memoization initial input of 1's in the time domain.

$\Phi_{2^{|memo|}b-memo_{10}}(n)$
$=\Phi_{2^{|memo|}-(2^{\lceil log_2(b')\rceil}-b')}(n)$
$=\Phi_{b'}(n)$

Therefore this memoization can properly be used to build up the spectral function of the desired base from the initial b = 1.

Although it can be concluded from this construction that it might just be easier to simply use the spectral function $\Phi_{b'}(n)$ initially without recursing through the memoization. This however would directly represent the frequency domain analog of just finding the sequence b'^k by just doing the following $\prod_{i=0}^{k} b'$. Finding the sequence in this way would be quite wasteful as the transformation into the frequency domain would be quite overkill. This does show that following the recursive pattern of the memoization will in fact determine the mathematically correct geometric sequence. This algorithm provides what would be inefficient when compared to a simple product, but it does provide great optimizations when compared to the time domain approach. The fully time domain approach operates in the time of $O(lg(b')n^2)$. The translation to the frequency domain required $O(klg(k))$ combined with the actual internal recursive memoization utilization of length lg(b') that each iterates over the n samplings for a final running time of $O(klg(k) + nlg(b'))$ where n is the exponent and b; is the desired base. This is clearly less desirable than simply using the $\Phi_{b'}(n)$ spectral function which would have a running time of only $O(klg(k))$, since the frequency domain calculation would only be of O(n) due to the memoization being completely removed. Of course simply performing the successive product is more optimal only occupying O(n). On top of this time complexity, the frequency domain representation creates the opportunity for errors to accumulate over the

successive iterations. The table described the descending time complexities of the various methods as described to calculate the geometric sequences.

The referenced naive product geometric sequence solver is available at 'Section_5/naive_sequence.py' or within the command line program at 'Geometric Sequence Solvers/Naive Sequence'.

The spectral geometric sequence solver is available at 'Section_5/spectral_geometric_sequen or within the command line program at 'Geometric Sequence Solvers/Spectral Geometric Sequence'.

6 Rational bases

To expand these constructions to be able to handle geometric sequences of all rational bases, two more extensions need to be created. First, geometric sequences of bases of negative integers must be handled. This has already pretty much been totally handled in the positive integer base algorithms although a more nuanced version can be created as well. Additionally, geometric sequences of a rational between 0 and 1 must be able to be created. This will allow for all rational numbers to be represented as geometric sequences constructed by successive midpoints. This section will remain in the time domain as it is more intuitive to understand midpoint transformations, although it can easily be applied to the frequency domain using the constructions of Section 5.

6.1 Expand bases to negative

There are two different ways in which geometric sequences of a negative integer base can be handled. First, there is the naive way that simply uses a positive integer proxy base that is scaled by alternating -1. This does work, but it leaves out the opportunity to use a more nuanced solution, actually constructing the negative base through midpoints and anti midpoints.

6.1.1 Naive

A negative base really is technically just a positive base that is scaled by -1. It is not equivalent to $(-1)b^i$, but rather $(-b)^i$. This means that Section 4's statements regarding taking out the scale factor would not work for this case, since the -1 is not a scale factor it is the parity of the base. Regardless of this, $(-b)^i = (-1)^i(b)^i$, so finding the geometric sequence is equivalent to using some leading edge method as described and simply alternating the sign. Formula 13, can be modified to be the following.

$$NaiveNegativeBase(b, n, k) = \bigcup_{i=0}^{k}[(sign(b))^i(\frac{|b|-1}{2^n}+1)^i] \tag{20}$$

However immediately following Formula 13, Example 3, provides as example where the provided base actually is negative, with some alternating leading edges. Even further than that, the final answer actually contains rational results between 0 and 1 as well. Clearly it is possible to achieve these results through the use of the same original unmodified construction. The only difference would be the starting base b = 1 no longer applies, but it would have to be a starting base of b = -1. The memoization functions also would have to be modified since the anti midpoint applied to b and -b are not equivalent, not even in their magnitudes. $(2b-1) \neq (2(-b)-1) = -(2b+1)$. Therefore a modified memoization will have to be made to accommodate this, which will hopefully be a natural extension of the existing memoization techniques.

6.1.2 Negative Log Memoization

The log time complexity memoization technique can be adapted and used almost in its exact state, but with some minor changes. These changes can best be illustrated through the use of an example.

Example 6. Log Memoization Comparison

First the memoization construction behind positive initial base will be shown
Let b' = 11
$\lceil \frac{11}{2} \rceil = 5, 11 \mod 2 = 1$
$\lceil \frac{5}{2} \rceil = 3, 5 \mod 2 = 1$
$\lceil \frac{3}{2} \rceil = 2, 3 \mod 2 = 1$
$\lceil \frac{2}{2} \rceil = 1, 2 \mod 2 = 0$ (Stop since 1 is reached)
Which results in a memoization of '1110' from b = 1.
Now the negative case can be demonstrated.
Let b' = -11
$\lceil \frac{-11}{2} \rceil = -5, -11 \mod 2 = 1$
$\lceil \frac{-5}{2} \rceil = -2, -5 \mod 2 = 1$
$\lceil \frac{-2}{2} \rceil = -1, -2 \mod 2 = 0$ (Stop since -1 is reached)
This results in a memoization from -1 of '110'

It should also be noted that it is simply not possible for continual memoizations from -1 to result in positive 1. Since $\lceil \frac{-1}{2} \rceil = 0$. This was shown in the Proof of Successive Memoization Functions.

Corollary 3. *Successive Negative Memoization Functions*

The successive applications of the division function $f(x) = \lceil x/2 \rceil$, has been to be defined as , $f^n(x) = \lceil x/2^n \rceil$, it is clear that for any input, $f^n(x)$ tends towards 1 for a positive b'.

'$\lim_{n\to\infty} \lceil b'/2^n \rceil = 1$'. Since for all natural numbers n, $b'/2^n$ will always be slightly larger than 0 (for $b' > 0$), which rounds up to 1. However, for the case of the negative b', b' < 0. $\lim_{n\to\infty} \lceil b'/2^n \rceil = 0$. This is due to the fact that the $\frac{-b'}{2^n}$ will result in an increasingly decreasing value delta, that for any small epsilon chosen, some n can be found that will make delta less than epsilon. However since $\frac{b'}{2^n} < 0 \forall n$ its ceiling will eventually always round up to 0.

Due to the conclusion of this Corollary, a more natural starting point b, might be b = 0 rather than b = -1. This can be encoded within the first bit of the memoization and would fit well with the existing schema.

For the new memoization, the first bit will represent the value of the initial b. b' > 0, b = 1 and b' < 0, b = 0. After reading this bit, the memoization would carry out as usual. This leading bit is analogous to what would be a representation of a parity bit.

Using this new starting point of b = 0, would add one final step to the memoization of -11 of $\lceil \frac{-1}{2} \rceil = 0, -1 \mod 2 = 1$.This means the final bit in the memoization of any negative base integer would result in a 1. However when looking at the memoization of a positive base, it can be noted that the final bit of the memoization always terminates in a 0. This solves the issue of needing to

artificially encode that starting point into a sort of parity bit. This is already naturally encoded within the memoization. Therefore in order to properly use the memoization for positive and negative base integers, the binary inverse of the last bit of the memoization must be read as the starting point.

6.1.3 Negative Constant Memoization

Now that the Log memoization schema has been expanded to include that of the negative final base b'. There must be some way to construct a constant memoization of b' as well. Examining the Example 6, can however give quite the hint as to what the memoization encoding might be. The memoization for b' = -11, when adding the final 1 bit results in a memoization of '1101' which when read in little endian is |-b| in base 2.

Corollary 4. *Constant memoization of negative*

The proof of Formula 14 determined that the memoization is equivalent to the following. $b' = s2^{n+1} - \sum_{i=0}^{n} 2^{n-i} m_i$. *where* $\lceil log_2(b') \rceil = n+1$ *and* m_i *represents that memoization. This was shown to be true regardless of the input of b', but was primarily intended for use with a positive base b'. However by augmenting the definition of n to be,* $\lceil log_2(|b'|) \rceil$ *-1= n, the formula may be able to include all bases.*

$b' = s2^{n+1} - \sum_{i=0}^{n} 2^{n-i} m_i$

$b' - s2^{n+1} = -\sum_{i=0}^{n} 2^{n-i} m_i$

$s2^{n+1} - b' = \sum_{i=0}^{n} 2^{n-i} m_i$

This formulation works for any base, augmenting Formula 14 to be the following.

$$ConstantTimeMemoization(b') = (b * 2^{\lceil log_2(|b'|) \rceil} - b')_2 \tag{21}$$

b is the initial starting point, 0 for negative bases, 1 for positive bases. b' is the target base. This will result in a binary string that should be read with its most significant bit being read first.

This confirms the suspicions that the constant time memoization for a negative base is simply the binary representation of the magnitude of the desired base, b'. Since for a negative base the memoization simplifies to $0 * 2^{\lceil log_2(|b'|) \rceil} - (-1)|b'|$ because of the b = 0 and the parity of b. Now this constant time construction of the memoization formula is working for all integer bases. There is one caveat that should be discussed before diving straight into the program representation of this scheme.

The initial base b = 0 and b = 1, technically means that the input array is the geometric sequence with a base of b with some sort of arbitrary length. This however brings up the issue in the negative b' case when b = 0 where the sequence would be $[0^0, 0^1, ..., 0^k]$, clearly the first element has an issue. The 0^0 element will just be defined to be equal to 1, which falls in line with the general floating point representation of this indeterminate form. It also falls in line with every other geometric sequence and it also makes the anti midpoint

between these first two element ($2*0^1-0^0=-1$), to actually output the correct value.

The integer constant memoization is available at 'Section_6/integer_constant_memoization.p or within the command line program at 'Memoization Functions/Integer Constant Memoization'.

The all integer base solver is available at 'Section_6/all_integer_bases.py' or within the command line program at 'Geometric Sequence Solvers/All Integer Bases'.

6.2 Expand bases to rational less than 1 greater than 0

Now that it is possible to construct a geometric sequence of any integer base, the next step is to expand this to include the rest of the rational numbers. This however is not as simple as the constructions found within Section 6.1. Due to the nature of floating point representations, a compromise will immediately be made in regards to the precision of the solution. Since for these rational numbers of the form $b' = \frac{i}{j}, i, j \in \mathbb{Z} : i < j$, a contraction must occur from 1 using midpoints rather than the anti midpoint. A naive approach can also be concluded as well.

6.2.1 Naive Rational Approach

Since the definition of the rational number $\frac{i}{j}$ is defined by the ratio of two integers, the memoization of these two integers can be found separately. The geometric series, $(\frac{i}{j})^k = \frac{i^k}{j^k}$can be then found in three parts.

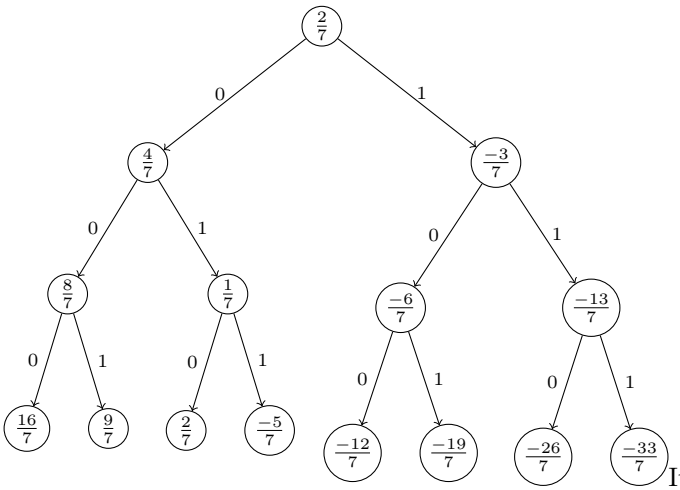
1. Find the geometric series of the numerator
2. Find the geometric series of the denominator
3. Perform an element wise division between these two arrays

6.2.2 Rational Approximations

We will begin with an immediate example.

Example 7. $\frac{2}{7}$

How would we retrieve the memoization for this simple rational number? The memoization would need to be constructed from anti midpoints and multiplications by 2 to result in a b = 1. The branches of each memoization will be shown as follows.



It might seem that, eventually some sort of finite combination of transformations will reach 1. Although this is not in fact the case.

Proof. The recursive transformations of 2b'-1 and 2b when operated upon the rational $b' = \frac{i}{j}$, clearly only operate upon the numerator i.

$2\frac{i}{j} - 1 = \frac{2i-j}{j}$,$2\frac{i}{j} = \frac{2i}{j}$. In both of the operations, j is unaffected. Therefore for the result to equal to 1, the numerator must be equal to j. The numerator must not be equal to some integer c*j where $c \neq 1$ since this would result in a divergence since these operations are equivalent to an expansion, not a contraction. So this problem can be reduced to simply looking at the numerator.

A general memoization can be applied to the transformation, similarly to how the Constant Time Memoization was proven.

n=1, $2\frac{i}{j} - m_0 = \frac{2i-jm_0}{j}$

n=2, $2\frac{2i-m_0}{j} - m_1 = \frac{4i-2jm_0-jm_1}{j}$

This appears to be in the form $\frac{2^n i - j\sum_{h=0}^{n-1} 2^{n-1-h} m_h}{j}$

Lemma 3. *Proof of the form*

Proof by induction $\frac{2^n i - j\sum_{h=0}^{n-1} 2^{n-1-h} m_h}{j}$

Base case n = 1 which should be equal to $\frac{2i-jm_0}{j}$

$\underline{\frac{2^n i - j\sum_{h=0}^{n-1} 2^{n-1-h} m_h}{j}}$

$=\underline{\frac{2^1 i - j\sum_{h=0}^{1-1} 2^{1-1-h} m_h}{j}}$

$=\underline{\frac{2i-jm_0}{j}}$

Inductive Step

Assuming when n = k is defined as the following $\frac{2^k i - j\sum_{i=0}^{k-1} 2^{k-1-i} m_i}{j}$*, therefore when n = k + 1 the result and the transformation should be* $\frac{2^{k+1} i - j\sum_{h=0}^{k} 2^{k-h} m_h}{j}$

$2(\frac{2^k i - j\sum_{h=0}^{k-1} 2^{k-1-h} m_h}{j}) - m_{k+1}$

$=\frac{2^{k+1} i - (j\sum_{h=0}^{k-1} 2^{k-h} m_h) - jm_{k+1}}{j}$

$=\frac{2^{k+1}i-(j\sum_{h=0}^{k}2^{k-h}m_h)}{j}$ *(Recollection of that final term)*

Proof. There must exist a finite memoization to represent any rational number, $\frac{i}{j}$ where i, j are coprime integers that evaluates to 1.

Proof by contradiction.

We are trying to find if there exists a memoization in which there is a convergence to 1. Therefore by using Lemma 3, the following equality can be set up.

$1 = \frac{2^n i-(j\sum_{h=0}^{n-1}2^{n-1-h}m_h)}{j}$

There must be some finite n in which this equality holds with a finite number of memoizations which are either 1 or 0.

$j = 2^n i - (j\sum_{h=0}^{n-1}2^{n-1-h}m_h)$

$j + j\sum_{h=0}^{n-1}2^{n-1-h}m_h = 2^n i$

$j(1 + \sum_{h=0}^{n-1}2^{n-1-h}m_h) = 2^n i$

Since each of the memoization bits are either 0 or 1, the following can be simplified to be a base 10 integer $c = \sum_{h=0}^{n-1}2^{n-1-h}m_h$

$j(1+c) = 2^n i$

$\frac{1+c}{2^n} = \frac{i}{j}$

This would mean that $i = 1 + c$, and $j = 2^n$ where n is a positive integer representing the length of the memoization.

The assumption that i and j are any coprime integers is violated by the 2^n limit upon j. This shows that every rational number is not representable within a finite memoization.

This proof demonstrates that in a finite memoization, not every rational number can be constructed purely from successive midpoints or divisions by 2. However what is possible is constructing a rational number in the form $\frac{a}{2^n} = a * 2^{-n}$. The problem has now transformed from finding an exact solution, but instead to finding an acceptable a and n that satisfies some error bound E. The minimization of this error can be expressed as $|\frac{i}{j} - \frac{a}{2^n}| < E$, $|\frac{i*2^n-aj}{j*2^n}| < E$ or simply $|\frac{c*2^n-a}{2^n}| < E$ if the rational number is not expressed directly as a ratio.

Proof. Expressing c' as an integer scaled by a power of 2

From this, bounds we can have two inequalities that the minimum acceptable solution will be within.

$|\frac{c*2^n-a}{2^n}| < E$

$\frac{c*2^n-a}{2^n} < E, -E < \frac{c*2^n-a}{2^n}$

Case 1

$\frac{c*2^n-a}{2^n} < E$

$c2^n - a < 2^n E$

$2^n(c-E) < a$

Case 2

$-E < \frac{c*2^n-a}{2^n}$

$-2^n E < c2^n - a$

$a < 2^n(c+E)$

Therefore n can be an independently varied integer value that must be within these two bounds. The problem has now been reduced to finding the first integer solution that falls within these bounds. This limits the range of possibilities to search for a.

6.2.3 Rational Compromise

There exist several approximation algorithms that can be used to find minimize the error, but a greedy approach will be taken to determine the minimal values of n and a that would lead to an error below the acceptable boundary. The algorithm behind this approximation is actually quite simple and is able to work for any rational number that can be expressed as a floating point. It simply iterates through values of n to find the first integer solution to the boundary condition. The reason why n was chosen as the independent variable is that values of a might be arbitrarily large however since the denominator is 2^n, only checking a relatively small subset of n will yield a sufficient answer within the error bounds. In general $n \ll a$. Also logarithms would be introduced if solving for a which would result in the potential for even less precise results.

This remarkably simple algorithm will find the minimum integer upper bound of a. This is possible by comparing the minimum upper bound with the maximum lower bound that is an integer. If there is any overlap between these two bounds then the maximum integer lower bound would represent the minimum value of a and n that corresponds to the acceptable error tolerance.

The following python program is able to output some interesting results.

This very simple algorithm works extremely well. For all rational inputs, there exists some approximation within the error bound. This unsurprisingly works with integer inputs with an error of 0, in its first iteration n = 0. Trying the input $\frac{2}{7}$ from Example 7 with an error of .0001 results in an output of $\frac{585}{2^{11}}$ with an actual error of approximately $7 * 10^{-5}$. This even works for numbers that are technically not rational in the real numbers, but are represented by rational floating point approximations such as algebraic and transcendental numbers. Using the same error tolerance of 10^{-4}, $\sqrt{2} \approx \frac{5793}{2^{12}}$ error $9 * 10^{-5}$, $\pi \approx \frac{3217}{2^{10}}$ error $9 * 10^{-6}$. Using an error of 10^{-4} results in the first three decimal places to be accurate which is nice to see, but much greater tolerances might be needed to be used when calculating geometric sequences since the error will be accumulating on every successive midpoint iteration similar to the Fourier Transform representation.

The rational approximation function is available at 'Section_6/base_2_rational_approximato or within the command line program at 'Rational Approximation Functions/Base 2 Rational Approximator'.

6.2.4 Rational Memoization

The memoization has already been mostly solved within Section 6.2.2 resulting in the following formula.

$$\frac{1+\sum_{h=0}^{n-1} 2^{n-1-h} m_h}{2^n} = \frac{i}{j}, (a-1)_2 \tag{22}$$

In conjunction with the rational_approximation algorithm, reduces the problem of the memoization quite significantly. Since the approximation is in the form of $\frac{a}{2^n}$, it is apparent that $1+\sum_{h=0}^{n-1} 2^{n-1-h} m_h = a$, $\sum_{h=0}^{n-1} 2^{n-1-h} m_h = a-1$. This means $(a-1)_2$ read from the most significant bit first serves as the memoization for the rational number in n-1 bits. This clearly does not work when n = 0 for any numerator other than 1, but that memoization has already been solved.

This memoization, defined for any i and j integers unfortunately does not work for i or in this case 'a' less than or equal to 0. Since the memoization relies on the binary representation of what would be a - 1, these values of a less than or equal to 0. The negative representation of binary numbers does not directly translate into this definition on the left hand side of the equation $\sum_{h=0}^{n-1} 2^{n-1-h} m_h = a-1$. Additionally, using midpoints between b and 1 it is not possible to reach some sort of small negative rational number since the additional of the +.5 added to each term will clearly make the recursive operations tend towards the positive end of the number line.

Proof. Midpoints fail to memoize any negative rational numbers

Assuming taking the successive midpoint and divide by 2 operations from some fixed point b will return the desired b'.

n=1 $\frac{b+m_0}{2}$

n=2 $\frac{\frac{b+m_0}{2}+m_1}{2} = \frac{b+m_0+2m_1}{4}$

Proof by induction $b' = \frac{b+\sum_{i=0}^{n-1} 2^i m_i}{2^n}$

Base case

n = 1

$\frac{b+\sum_{i=0}^{1-1} 2^i m_i}{2^1} = \frac{b+2^0 m_0}{2^1}$

Inductive Step

Assuming n = k is $\frac{b+\sum_{i=0}^{k-1} 2^i m_i}{2^k}$, then n = k +1 is $\frac{b+\sum_{i=0}^{k} 2^i m_i}{2^{k+1}}$

$\frac{\frac{b+\sum_{i=0}^{k-1} 2^i m_i}{2^k}+m_k}{2}$ (Applying midpoint with m_k)

$\frac{\frac{b+(\sum_{i=0}^{k-1} 2^i m_i)+m_k 2^k}{2^k}}{2}$

$\frac{b+(\sum_{i=0}^{k-1} 2^i m_i)+m_k 2^k}{2^{k+1}}$

$\frac{b+(\sum_{i=0}^{k} 2^i m_i)}{2^{k+1}}$ (Absorbing trailing term)

So if this is assumed to be equal to an eventual $b' = \frac{b+\sum_{i=0}^{n-1} 2^i m_i}{2^n}$.

This is simply the following $\frac{b+c}{2^n}$, where c must be a positive integer. Since b' is less than 0, b+c < 0 as well. This means that -b $>$ c. Due to the memoization not being fixed, a standardized starting point cannot be achieved. The memoization is potentially infinity long depending on the n required or requested for the sequence. This means that a standard fixed b is not achievable. For this to

work, b would clearly have to be equivalent to the limit as b approached negative infinity or a non standard base for each problem, contradicting the initial assumption.

For negative rational numbers, there are no fixed starting b points which mean there is no memoization that will work to recover the required output base. This however can be handled using the naive approach discussed in Section 6.1.1, simply finding the absolute value of the rational input then alternating the final signs.

Finally the condition for rational numbers greater than 1 can be discussed. These numbers would have to use both the contraction and the expansion modes, which is not possible as both are inverse of each other. Therefore these numbers must endure a normalization before being operated upon to be within 0 and 1. The scaling by 2^n, will be undone within the final leading edge. For this same reason, the geometric sequence of rational numbers less than 0 must be taken to be of their positive normalized counterpart since the contraction will naturally lead them to be positive. There also is the case for geometric sequences of negative power, this case can be handled easily by considering the reciprocal of the desired base which is possible due to the rational domain expansion and the absolute value of the exponent.

The rational memoization function is available at 'Section_6/rational_memoization.py' or within the command line program at 'Memoization Functions/Rational Memoization'.

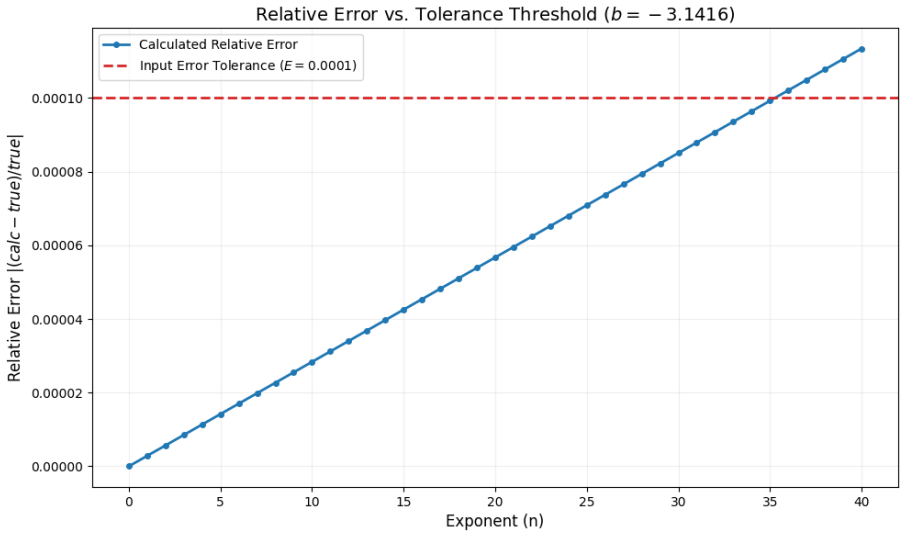
The rational input base geometric sequence solver is available at 'Section_6/rational_sequence.py' or within the command line program at 'Geometric Sequence Solvers/Rational Sequence'.

6.3 Augmentation of Error Tolerance

6.3.1 On The Topic of Error

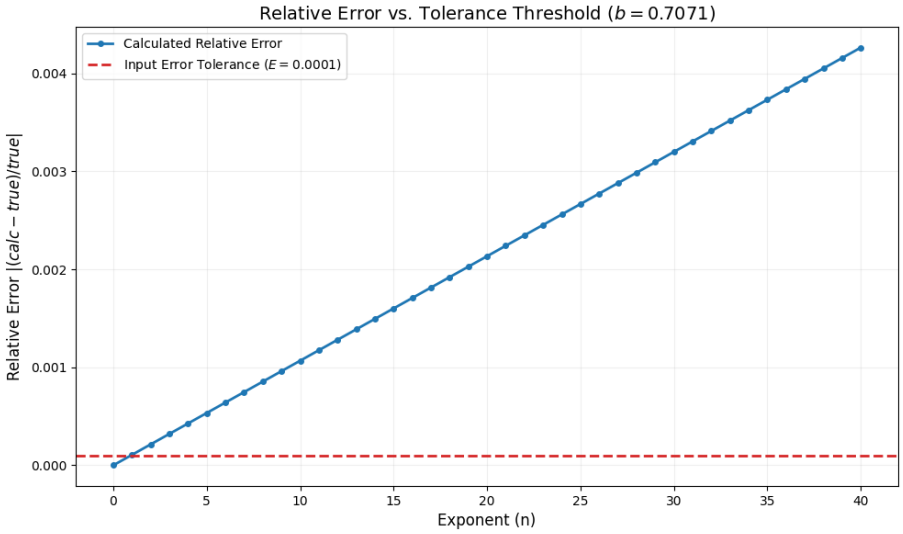
There was an initial suggestion within Section 6.2.3 in which the error would essentially compound for the rational approximation inputs. Observing the data, this does seem to be taking place, although the increments of change are quite small. The following figures represent the actual change of percent error as more terms are added to the sequence. The figures use $-\pi$ and $\frac{1}{\sqrt{2}}$ as b'.

Figure 2: Relative Error of $-\pi$ Using Absolute Error Approximation



This shows the error with a tolerance of .0001 to approximate $-\pi$

Figure 3: Relative Error of $\frac{1}{\sqrt{2}}$ Using Absolute Error Approximation



This demonstrates the error when approximating $\frac{1}{\sqrt{2}}$ with a tolerance of .0001.

Clearly both of these figures demonstrate a crossing of the tolerance threshold. This means that the set tolerance only is valid for very few elements in the

sequence. A new formulation in approximation needs to be created in order to maintain below tolerance for all desired elements of the sequence. This can easily be remedied by essentially making the approximation to be more accurate. The actual error as was initially defined is as follows, $|\frac{a}{2^n} - b'| = \mathrm{X}$, which is set to be bounded by some threshold E. However this only guarantees that the second term in the geometric sequence, (k = 1), to be within this bound. It would be assumed that the error would be growing by a factor of X, for each new element of the sequence. However that absolute error defined for each iteration of k can be defined as $|(\frac{a}{2^n})^k - b'^k| = X_k$ which is absolutely not X^k. This is due to the fact that exact geometric sequence of $\frac{a}{2^n}$ will be recovered by the leading edge methods which approximates the b^k progression. What is essentially an element wise exponentiation of can be used to find new values of 'a' and 'n' for the given maximum exponent k, which still satisfied the boundary condition E.

Proof. Finding the new conditions of $|(\frac{a}{2^n})^k - b'^k| = X_k < \mathrm{E}$

Previously, the boundaries were shown to be $2^n(c - E) < a$, $a < 2^n(c + E)$, when k = 1. This will be more generalized to any k. Where b' = c.

$|(\frac{a}{2^n})^k - c^k| = X_k < \mathrm{E}$

$-E < (\frac{a}{2^n})^k - c^k < E$

Case 1

$(\frac{a}{2^n})^k - c^k < E$

$\frac{a^k}{2^{nk}} - c^k < E$

$a^k < 2^{nk}(E + c^k)$

$a < (2^{nk}(E + c^k))^{\frac{1}{k}}$

$a < 2^n(E + c^k)^{\frac{1}{k}}$

Case 2

$-E < (\frac{a}{2^n})^k - c^k$

$c^k - E < \frac{a^k}{2^{nk}}$

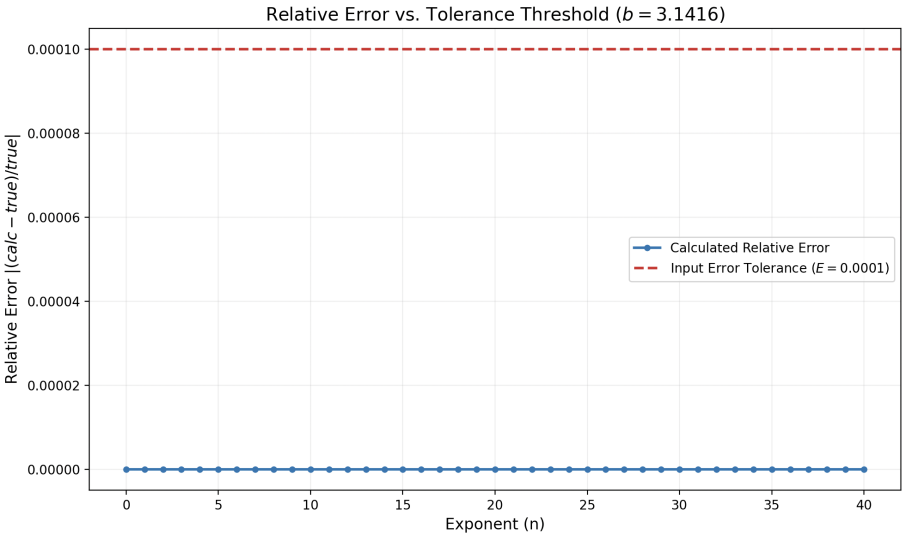
$2^{nk}(c^k - E) < a^k$

$2^n(c^k - E)^{\frac{1}{k}} < a$

These boundaries can be substituted into the main approximate boundary conditions.

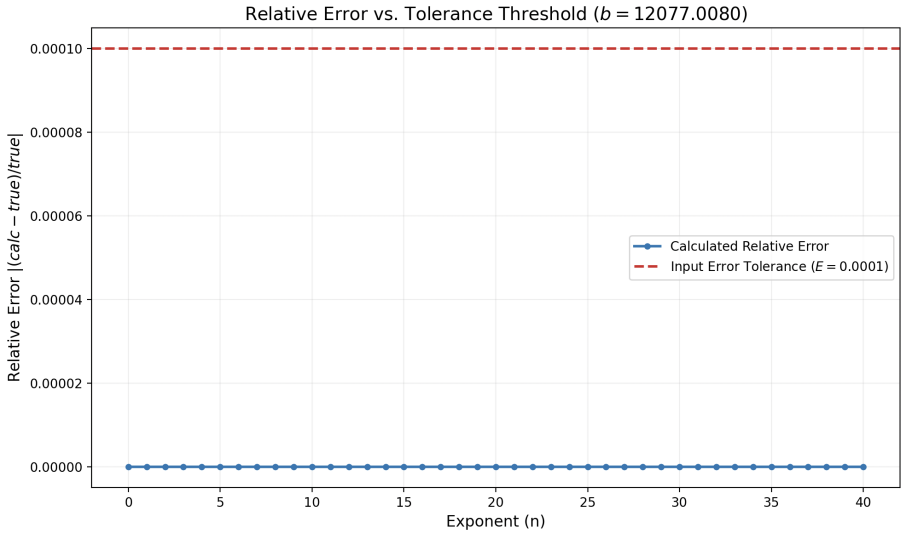
This does have greatly improved error, staying well within the bounds. For the numbers tested, the relative error is either 0 or very near to 0 in the floating point representation. The improvised bounds, essentially allow for a more precise and accurate approximation to be found with almost nearly all of the represented decimal places to be the same. It estimates $\pi \approx \frac{884279719003555}{2^{50}} * 4$ and $\sqrt{2} \approx \frac{388736063997}{2^{39}}$.

Figure 4: Relative Error of π Using Relative Approximation k = 1



The improved error of π can be seen in the figure above. The relative error is far below the tolerance.

Figure 5: Relative Error of $1000 * e * \pi * \sqrt{2}$ Using Relative Approximation k = 1



Now this figure shows a larger number, $1000 * e * \pi * \sqrt{2}$, and it still manages to provide a relative error of nearly zero for all elements in the sequence.

This solver and section is defined as the rational solver, however these clearly are not rational numbers but are transcendental and algebraic numbers that possess no rational representation. However, within Python, these numbers are represented generally as 64 bit double precision floating point numbers, meaning to a precision of around decimal places. In this representation, π could be represented only as 3.141592653589793, which quite evidently can be represented as a rational number, namely $\frac{3141592653589793}{10^{15}}$ which is nearly the approximation that was found $\frac{884279719003555}{2^{50}}$. This can be demonstrated by, $\frac{3141592653589793}{10^{15}} = \frac{3141592653589793}{2^{log_2 10^{15}}} = \frac{3141592653589793}{2^{15log_2 10}} \approx \frac{3141592653589793}{2^{50}}$. The approximation method might appear to be getting an error value that would be considered to be 'overkill'. For the limited amount of precision possible within these floating point numbers, the number is being precisely represented within the floating point domain.

6.3.2 Relative Error Approximation

It might also be beneficial to find the boundary conditions in term of relative error, not absolute error. The error of the values was measured in relative error for the above figure, but the actual boundary conditions were represented in absolute terms. This technically is incorrect as the units of these measures of error are different.

Proof. Finding the new boundary conditions for 'a' when, $|\frac{(\frac{a}{2^n})^k - c^k}{c^k}| < E$

$-E < \frac{(\frac{a}{2^n})^k - c^k}{c^k} < E$

Case 1

$\frac{(\frac{a}{2^n})^k - c^k}{c^k} = (\frac{a}{c2^n})^k - 1 < E$

$(\frac{a}{2^n})^k - c^k < c^k E$

$\frac{a^k}{2^{nk}} - c^k < c^k E$

$a^k < 2^{nk}(c^k E + c^k)$

$a < (2^{nk}(c^k E + c^k))^{\frac{1}{k}}$

$a < c2^n(E+1)^{\frac{1}{k}}$

Case 2

$-c^k E < (\frac{a}{2^n})^k - c^k$

$c^k - c^k E < \frac{a^k}{2^{nk}}$

$c^k 2^{nk}(1-E) < a^k$

$c2^n(1-E)^{\frac{1}{k}} < a$

This is assuming c > 0 or k is even, $c2^n(1-E)^{\frac{1}{k}} < a < c2^n(1+E)^{\frac{1}{k}}$

For c < 0 and is odd, $c2^n(1+E)^{\frac{1}{k}} < a < c2^n(1-E)^{\frac{1}{k}}$

This provides the new bounds for a that can be used to determine an appropriate approximation with respect to the relative error condition.

Using this rational approximation boundary yields what would be the same results since the previous method reaches the minimum error allowed by floating point representations. This methodology is just more mathematically correct as the error for small values would already be quite small below the threshold using only absolute error.

In this representation there is essentially no difference between the numbers approximated and the true values, which means the approximation is essentially as good as it possibly could get. This naturally leads to a solver that no longer uses an error tolerance, but instead finds the rational approximation for all of the decimal places. An approximate ratio of this nature would be one where k = 1, with a tolerance of about 10^{-16}. This would create an approximation in which $\frac{a}{2^n}$ has most if not all of the decimal places correct for b'. The methodologies shown this far are able to generate the geometric sequence of any floating point representation real number by only using successive midpoints or anti midpoints upon a starting array of all ones. In the domain of real numbers, this method fails to produce exact solutions, however in the domain of floating point real number representations, this method can actually handle all real inputs.

The error chart creation function is available at 'Utilities/error_chart.py' or within the command line program at 'Analysis/Error Chart'.

7 Additional Midpoint Property Matrix Facts

7.1 Cascade matrix Representations

7.1.1 Separable Vectors

Observing the well known matrix created by an input B = $[3^0, 3^1, 3^2, 3^3]$, with weights $[\frac{1}{2}, \frac{1}{2}]$,the following cascade matrix is generated.

1	3	9	27
2	6	18	0
4	12	0	0
8	0	0	0

The input b yields the leading edge B' of powers of 2 as expected. The interior elements of the matrix are actually defined as the linear combination between B'*B with the 0 elements placed to make it upper diagonal. If this is true than the matrix can be defined as separable and be easily calculated by simply taking linear combinations.

In this case b'*b equals.

1	3	9	27
2	6	18	54
4	12	36	108
8	24	72	216

This seems to follow the midpoint rule in which $\frac{a_{(i,j)}+a_{(i+1,j)}}{2} = a_{(i,j+1)}$. This means that the separable matrix created could actually be a sub matrix of a larger upper diagonal matrix of the usual upper diagonal form containing powers of 3 with a greater terminal exponent.

Proof. The cascade matrix is an augmented separable matrix.

In the general case the matrix is defined with an input row vector $[b^0, b^1, ..., b^k]$ $(1^{st}row)$, using a midpoint function the weights matrix kernel is equivalent to $[\frac{1}{2}, \frac{1}{2}]$. It has been shown that the leading edge $(1^{st}column)$ is defined as $[(\frac{b+1}{2})^0,(\frac{b+1}{2})^1,...,(\frac{b+1}{2})^k]$. This means that the outer product between these two vectors B' and B would yield the a matrix of the following form in terms of b.

$$M = \begin{matrix} b^0(\frac{b+1}{2})^0 & b^1(\frac{b+1}{2})^0 & ... & b^k(\frac{b+1}{2})^0 \\ b^0(\frac{b+1}{2})^1 & b^1(\frac{b+1}{2})^1 & ... & b^k(\frac{b+1}{2})^1 \\ ... & ... & \ddots & \vdots \\ b^0(\frac{b+1}{2})^3 & b^1(\frac{b+1}{2})^3 & ... & b^k(\frac{b+1}{2})^k \end{matrix}$$

This matrix can more simply be defined as $M_{(i,j)} = b^{j-1}(\frac{b+1}{2})^{i-1}$. Now it must be shown that the cascade matrix generated upon a larger input B with a larger k, contains all of the elements within this matrix M. Since the cascade creation of the matrix creates a new anti diagonal slice each time k increases by one, the $b^k(\frac{b+1}{2})^k$ element would be apart of the last anti diagonal slice. The final row of the M matrix (row k = M[k]), can essentially be treated as a new input row for starting a new cascade matrix. Even though the input

is not a geometric sequence, a cascade matrix still can be constructed in the general sense as defined in Section 2 with MidpointLeadingEdge, which takes in a general input vector. This formula was assumed to iterate |M[k]|-1 times producing a |M[k]| x |M[k]|, k + 1 x k + 1, matrix P, when starting with the initial basis. Placing the M matrix on top of the P matrix where they over lap. A similar construction can be applied to the k^{th} column of M, M[:][k] when using the inverse of MidpointLeadingEdge, which will create a matrix N, of the same dimensions due to the same summation bounds. When all combined together, the following matrix M' is created which will have dimensions of,

|M[k]| + |P[1]| - 1 x |M[:][k]| + |N[:][1]| - 1 (subtracting one shared row and column instance)

2(k + 1) - 1 x 2(k + 1) - 1

2k+1 x 2k+1.

$$M' = \begin{matrix} M[1:k][1:k] & N \\ P & 0 \end{matrix}$$

Proof by Induction

k = 1 M has dimension k + 1 x k + 1, 2 x 2

$$M = \begin{matrix} (\frac{b+1}{2})^0 b^0 & (\frac{b+1}{2})^0 b^1 \\ (\frac{b+1}{2})^1 b^0 & (\frac{b+1}{2})^1 b^1 \end{matrix}$$

M' would have dimension 2 * 1 + 1 x 2 * 1 + 1, 3 x 3.

$$M' = \begin{matrix} (\frac{b+1}{2})^0 b^0 & (\frac{b+1}{2})^0 b^1 & (\frac{b+1}{2})^0 b^2 \\ (\frac{b+1}{2})^1 b^0 & (\frac{b+1}{2})^1 b^1 & 0 \\ (\frac{b+1}{2})^2 b^0 & 0 & 0 \end{matrix}$$

For this to be true, taking the midpoint between the last row of M, should yield $(\frac{b+1}{2})^2$ by the midpoint leading edge formula with the input of $[(\frac{b+1}{2})^0 b^0 (\frac{b+1}{2})^0 b^1 (\frac{b+1}{2})^0 b^2]$.

$\frac{\frac{(\frac{b+1}{2})^1 b^0 + (\frac{b+1}{2})^1 b^1}{2}}{2}$

$\frac{\frac{\frac{b+1}{2} + \frac{b+1}{2} b}{2}}{2}$

$\frac{\frac{b^2+b+b+1}{2}}{2}$

$\frac{b^2+2b+1}{4}$

$\frac{(b+1)^2}{2^2}$(Binomial Theorem)

$(\frac{b+1}{2})^2$

Inductive step

k = n

Assume that the M_n matrix, which is the outer product of $[b^0, b^1, ..., b^n]$ and $[(\frac{b+1}{2})^0, (\frac{b+1}{2})^1, ..., (\frac{b+1}{2})^n]$, produces an entirely valid M'_n matrix of dimension 2n + 1 x 2n + 1. The theoretical M_{n+1}will have an additional two rows and columns that need to be justified since 2(n+1) + 1 = 2n + 3 with a first column final row element equivalent to $(\frac{b+1}{2})^{2n+2}$

$$M'_n = \begin{matrix} (\frac{b+1}{2})^0 b^0 & ... & (\frac{b+1}{2})^0 b^n & \cdots & (\frac{b+1}{2})^0 b^{2n} \\ \vdots & \ddots & \vdots & \iddots & 0 \\ (\frac{b+1}{2})^n b^0 & \cdots & (\frac{b+1}{2})^n b^n & 0 & 0 \\ \vdots & \iddots & 0 & 0 & 0 \\ (\frac{b+1}{2})^{2n} b^0 & 0 & 0 & 0 & 0 \end{matrix}$$

This M'_n is assumed to be correct, which can be expanded to include M'_{n+1} which is matrix that will also obey the successive midpoint property.

$$M'_{n+1} = \begin{matrix} (\frac{b+1}{2})^0 b^0 & ... & (\frac{b+1}{2})^0 b^n & (\frac{b+1}{2})^0 b^{n+1} & \cdots & (\frac{b+1}{2})^0 b^{2n} & (\frac{b+1}{2})^0 b^{2n+1} & (\frac{b+1}{2})^0 b \\ \vdots & \ddots & \vdots & \vdots & \cdots & (\frac{b+1}{2})^1 b^{2n} & (\frac{b+1}{2})^1 b^{2n+1} & 0 \\ (\frac{b+1}{2})^n b^0 & \cdots & (\frac{b+1}{2})^n b^n & (\frac{b+1}{2})^n b^{n+1} & \iddots & (\frac{b+1}{2})^2 b^{2n} & 0 & 0 \\ (\frac{b+1}{2})^{n+1} b^0 & \cdots & (\frac{b+1}{2})^{n+1} b^n & (\frac{b+1}{2})^{n+1} b^{n+1} & & & 0 & 0 \\ \vdots & \cdots & \iddots & & & & 0 & 0 \\ (\frac{b+1}{2})^{2n} b^0 & (\frac{b+1}{2})^{2n} b^1 & (\frac{b+1}{2})^{2n} b^2 & & & & 0 & 0 \\ (\frac{b+1}{2})^{2n+1} b^0 & (\frac{b+1}{2})^{2n+1} b^1 & & & & & 0 & 0 \\ (\frac{b+1}{2})^{2n+2} b^0 & & & & & & 0 & 0 \end{matrix}$$

The appearance of this matrix is quite different than the M'_n, but there only are two additional anti diagonal slices added. The main anti diagonal from element $M'_{n+1}[2n+3][1]$ to $M'_{n+1}[1][2n+3]$ and the anti diagonal slice spanning from $M'_{n+1}[2n+2][1]$ to $M'_{n+1}[1][2n+2]$. The final four rows of this matrix can be analyzed in further detail with elements that assumed to be true within M'_n marked.

$$\begin{matrix} (\frac{b+1}{2})^{2n-1} b^0 \in M'_n & (\frac{b+1}{2})^{2n-1} b^1 \in M'_n & (\frac{b+1}{2})^{2n-1} b^2 & (\frac{b+1}{2})^{2n-1} b^3 \\ (\frac{b+1}{2})^{2n} b^0 \in M'_n & (\frac{b+1}{2})^{2n} b^1 & (\frac{b+1}{2})^{2n} b^2 & \\ (\frac{b+1}{2})^{2n+1} b^0 & (\frac{b+1}{2})^{2n+1} b^1 & & \\ (\frac{b+1}{2})^{2n+2} b^0 & & & \end{matrix}$$

Using Formula 2. $SuccessiveMidpoint(A) = [\frac{1}{2^{|A|-1}} \sum_{i=0}^{|A|-1} \binom{|A|-1}{i} a_i]$

$(\frac{b+1}{2})^{2n} b^1$ therefore must be equal to SuccessiveMidpoint($[b^1, ..., b^{2n+1}]$)

$\frac{1}{2^{2n}} \sum_{i=0}^{2n} \binom{2n}{i} b^{i+1}$

$= \frac{1}{2^{2n}} b \sum_{i=0}^{2n} \binom{2n}{i} b^i$

$= \frac{1}{2^{2n}} b(b+1)^{2n}$ (Binomial Theorem)

$= (\frac{b+1}{2})^{2n} b^1$

Using this fact, $\frac{(\frac{b+1}{2})^{2n} b^0 + (\frac{b+1}{2})^{2n} b^1}{2}$

$= \frac{(\frac{b+1}{2})^{2n} (1+b)}{2}$

$= \frac{(b+1)^{2n} (1+b)}{2^{2n+1}}$

$= (\frac{b+1}{2})^{2n+1} b^0$

Also using Formula 2, $(\frac{b+1}{2})^{2n+1} b^1$ must be equal to SuccessiveMidpoint($[b^1, ..., b^{2n+2}]$)

$\frac{1}{2^{2n+1}} \sum_{i=0}^{2n+1} \binom{2n+1}{i} b^{i+1}$

$= \frac{b}{2^{2n+1}} \sum_{i=0}^{2n+1} \binom{2n+1}{i} b^i$

$= \frac{b}{2^{2n+1}} (b+1)^{2n+1}$(Binomial Theorem)

$= (\frac{b+1}{2})^{2n+1} b^1$

Finally $(\frac{b+1}{2})^{2n+2} b^0$ can be shown to be the midpoint of these last two elements.

$\frac{(\frac{b+1}{2})^{2n+1} b^0 + (\frac{b+1}{2})^{2n+1} b^1}{2}$

$= (\frac{b+1}{2})^{2n+1} \frac{1+b}{2}$

$= b^0 (\frac{b+1}{2})^{2n+2}$

The matrix M' clearly is equivalent to the outer product of B' and B leading to a rank of 1. This means that the elements within the M matrix are all elements of a larger cascade matrix. Due to this property, if M has dimension k x k, then simply applying the successive midpoint formula on an initial basis of $[b^0, ..., b^{k-1}]$ to yield a cascade matrix the rest of the matrix can be filled in by simply taking successive products on the elements in the matrix, If this process is done to the final row, then it will be within the form of $[(\frac{b+1}{2})^{k-1}b^0, \ldots, (\frac{b+1}{2})^{k-1}b^{k-1}]$ or $(\frac{b+1}{2})^{k-1}[b^0, \ldots, b^{k-1}]$. Formula 10, $a * MidpointLeadingEdge(\bigcup_{i=0}^{k-1}[b^i]) = \bigcup_{i=0}^{k-1}[a(\frac{b+1}{2})^i]$ where a = $(\frac{b+1}{2})^{k-1}$ dictates that successive midpoints will create a leading edge in the for $(\frac{b+1}{2})^{k-1}a(\frac{b+1}{2})^i$ where i runs from 0 to k - 1.

$(\frac{b+1}{2})^{k-1}(\frac{b+1}{2})^i = (\frac{b+1}{2})^{i+k-1}$ meaning that this will create the remainder of an equivalent leading edge for an input of $[b^0, ..., b^{2k+1}]$ without having to create the N matrix from the above proof $M' = \begin{matrix} M[1:k][1:k] & N \\ P & 0 \end{matrix}$. This cuts out one third of the work required to traditionally calculate the leading edge of basis of a geometric series of 2k+1 when using successive midpoints, but this does not change the complexity.

For now, this fact remains as an interesting explanation as to the nature of the interior elements of the cascade matrix. Due to the successive midpoint property, this fact also applies for the anti midpoint case as well.

7.1.2 The Diagonal of A Cascade Matrix

Another interesting fact that arises from the cascade matrix generate through successive midpoints and anti midpoints of a geometric sequence, relates to the elements upon the main diagonal. From looking at enough cascade matrices resulting from geometric sequence inputs, this fact might have been quite apparent. The diagonal elements do not simply represent random numbers that allow for the successive midpoint property to work, but in fact they are what would be the perfect transition between both sets of geometric sequence. More rigorously, they defined another geometric sequence, that of which the base is b'*b. Using the facts of the previous Section 7.1.1, it has been shown that the elements of the matrix correspond exactly to different combinations of products of b and b'. Rows closer to the first row possess less of an 'influence' of products of b', while columns closer to the first column are more purely elements of geometric sequences of b' with less b products. It can be surmised that the diagonal, where the row and column numbers are identical possess exactly the same 'mixture' of b and b'. This will be demonstrated by an example and then formally shown with a proof.

Example 8. Usual Example

Using the usual example of b = 3 with the anti midpoint weights [-1, 2], the following matrix can be created, it has been expanded to show more terms. (For the purpose of creating this matrix, the elements were not created by actually taken using the anti midpoint weights, but were in fact calculated using the

separable vectors method. This method is a much easier way of creating these matrices.)

1	3	9	27	81	243
2	6	18	54	162	0
4	12	36	108	0	0
8	24	72	0	0	0
16	48	0	0	0	0
32	0	0	0	0	0

Clearly the non zero elements of the diagonal show the start of a geometric sequence with base of 6, or 2 * 3; 1, 6, 36.

Corollary 5. *Nonzero diagonal elements of a successive midpoint property matrix are geometric sequences of the element wise product of the input and output leading edge.*

In Section 7.1.1, it has been shown that $M_{(i,j)} = b^{j-1}(\frac{b+1}{2})^{i-1}$.

Since the diagonal is defined as elements when $i = j$*, this can be reduced to the following.*

$M_{(i,i)} = b^{i-1}(\frac{b+1}{2})^{i-1}$

$= (b(\frac{(b+1)}{2}))^{i-1}$

$= (\frac{b^2+b}{2})^{i-1}$

Therefore the non zero diagonal elements are clearly equal to the geometric sequence with base $b(\frac{(b+1)}{2})$ *or* bb'.

A consequence of this Corollary 5 is that traditional memoization can be entirely skipped under certain conditions. If the required output base is b' then the solutions to the following equations, $\frac{x^2+x}{2} = b'$ or conversely $(2x-1)x = 2x^2 - x = b'$, will result in either the starting point for a singular round (n = 1) of successive midpoints or successive anti midpoints. Limiting the domain of x to integers is the only way to produce exact solutions, however using rational approximations of algebraic numbers which are within the floating point real number representation domain, it may be possible to achieve exact solutions for non integer solutions.

There is no real criteria that would prefer using the midpoint or anti midpoint versions of these formulas. Clearly at least one integer or rational number in the form $\frac{a}{2^n}$,solution would be preferred to use. This means that both formulas should be solved to try and determine if a solution within this criteria fits. Additionally, only real solutions can be used.

Proof. Finding solutions

Simply utilizing the quadratic equation will help find the roots of this polynomial.

Case 1: Solutions requiring the midpoint weights

$\frac{x^2+x}{2} = b'$

$= x^2 + x - 2b' = 0$

$= \frac{-1 \pm \sqrt{1+8b'}}{2}$ (Quadratic formula)

Case 2: Solutions requiring the anti midpoint weights

$2x^2 - x = b'$
$2x^2 - x - b' = 0$
$= \frac{1 \pm \sqrt{1+8b'}}{4}$

These solutions are quite similar. The determinants of both (1+8b') will yield some undefined behavior within the real domain when 1+8b' < 0, b' $< \frac{-1}{8}$. For this reason it will be beneficial to just use |b'| and alternate the final sequence signs, since in this case neither equation will produce real roots.

$f(b) = \frac{-1 \pm \sqrt{1+8|b'|}}{2}, g(b) = \frac{1 \pm \sqrt{1+8|b'|}}{4}$

For the sake of optimization, g can be expressed as a linear transformation of f.

$f(b) = \frac{-1 \pm \sqrt{1+8|b'|}}{2}$
$-f(b) = \frac{1 \mp \sqrt{1+8|b'|}}{2}$
$\frac{-f(b)}{2} = \frac{1 \mp \sqrt{1+8|b'|}}{4}$

This allows for the two solutions of f to just be transformed in order to find the solutions of g. Since g is simply a scaling of $\frac{-1}{2}$ to f, solutions of f can only be rational if and only if solutions of g are rational. This means that simply finding a single solution is sufficient. However, generally finding the solution that requires the fewest memoization steps will be beneficial when actually constructing this initial sequence. The memoizations are valid for b' inputs that are of the form $\frac{a}{2^n}$, the memoizations are as follows with b denoting the base of the starting geometric sequence.

$$GeneralMemoization(a, n, b) = \begin{cases} |b * 2^{\lceil log_2(|a|) \rceil} - a|_2 & n = 0 \\ ||a| - 1|_2 & n \neq 0 \end{cases} \tag{23}$$

Joining this into a single memoization function would be $GeneralMemoization((a, n, b) = \left|a - (b * 2^{\lceil log_2(|a|) \rceil} * 0^n + 1 - 0^n)\right|$ when 0^0 is defined as 1, this just is not as intuitive.

Proof. Generalized memoization bit lengths

Finding the number of bits required for memoizations of each solutions to f and g can be easily done by utilizing these memo cases. Assuming a suitable approximation for the solutions to f and g are already pre computed in the form of $\frac{a}{2^n}$, then this can be divided into a few cases.

Case 1: n = 0, b = 0 -> b' is a negative integer

The memoization just simplified $|b * 2^{\lceil log_2(|a|) \rceil} - a|_2$ to the binary representation of |a| or |b'| therefore the bit length for this will be $\lceil log_2(|a|) \rceil$

Case 2: n = 0. $b = 1$ -> b' is a positive integer

$|b * 2^{\lceil log_2(|a|) \rceil} - a|_2$ becomes limited by $\lceil log_2(|a|) \rceil$ as well.

Case 3: $n \neq 0, b = 1$ -> b' is a positive rational number

$||a| - 1|_2$ which has a bit length of $\lceil log_2(|a| - 1) \rceil$ which can be less than to n in the initial definition due to trailing zeros.

Therefore finding the proper root to start from involves the following formula.

$$MemoBitLength(a, n) = \begin{cases} \lceil log_2(|a|) \rceil & n = 0 \\ n & n \neq 0 \end{cases} \tag{24}$$

The minimum of each root's memo bit length will denote which root to be picked. The memobitlength of the actual b' should also be places within the minimum calculation as well.

This almost allows for a program to be created that will use the diagonal property, but there still is a missing piece. The maximum exponent of the geometric sequence of the initial basis is not equivalent to the exponent of the resulting diagonal sequence. This can be seen within Example 8. The input was $[3^0, 3^1, 3^2, 3^3, 3^4, 3^5]$ while the resultant diagonal only had three non zero elements, $[6^0, 6^1, 6^2]$. Based off of this, it seems like the the number of non zero element on the diagonal appears to be $\lceil \frac{k-1}{2} \rceil$ where k is the highest exponent.

Proof. The number of non zero diagonal elements is $d = \lceil \frac{k+1}{2} \rceil$, sparsity of the diagonal

The elements of the anti diagonal of a square matrix M of dimension n x n are defined as $M_{i,(n-i+1)}$ and the diagonal elements are $M_{i,i}$ where i iterates from 1 to n. Since the diagonal and anti diagonal are discrete lines, they might not have an intersecting element like continuous perpendicular lines. The row coordinates of the diagonal and anti diagonal do match as they both are i so the problem is reduced to equivalency of the column coordinate.

n-i+1=i

n=2i-1

This means that this intersection is only possible when n is represented as the definition of an odd number at index 2i-1.

The dimensions of these matrices are equal to k + 1 x k + 1, therefore there have been k + 1 anti diagonal updates since every new increment of k adds a new anti diagonal. Due to the construction of the geometric sequences by convolution, a new non zero element is added to the diagonal every second increment of k. Then the number of non zero elements suggests a growth rate of $\frac{k+1}{2}$, due to the roughly linear relationship between the dimension of the cascade matrix and the number of diagonal elements. To satisfy the base case d(0) = 1 which represents the singleton matrix $[b^0]$ and d(1) = 1 $\begin{matrix} b^0 & b^1 \\ b'^1 & \end{matrix}$, with only a single diagonal element. Since the diagonal updates on every second increment, $d(k) = \lceil \frac{k+1}{2} \rceil$.

With the knowledge that there are only $d(k) = \lceil \frac{k+1}{2} \rceil$ non zero elements of the geometric sequence within the diagonal of a matrix generated with an input geometric sequence of highest power k, k can be solved for in terms of d.

Proof. Defining k in terms of d

Since $d = \lceil \frac{k+1}{2} \rceil$ encompasses two integers, using the lowest integer of these two will involve one less step.

Case 1: Let k = 2q-1, q is an integer, k odd

$\lceil \frac{2q-1+1}{2} \rceil$
=q
Case 2: Let k = 2q, q is an integer, k even
$\lceil \frac{2q+1}{2} \rceil$
$=\lceil q + \frac{1}{2} \rceil$
=q+1

This means that even k represents the first instance of a new element within the diagonal. Since it would require k' = 2q+1 to achieve d = $\lceil \frac{2q+1+1}{2} \rceil = q+1$, but 2q < 2q+1 so it is beneficial to use an even k.

Since k = 2q, q = $\frac{k}{2}$ then $d = \frac{k}{2} + 1$, solving for k.
$d = \frac{k}{2} + 1$
$2d - 2 = k$

This means that in order to find a diagonal sequence, the input geometric sequence needs to have a highest exponent of the following equation. Although this is not entirely true since having a diagonal of length d does not mean the geometric sequence will have a final exponent of d as well. An exponent of of d + 1 should be used to achieve the correct final exponent. So plugging this into 2d-2, 2(d+1)-2 = 2d, which is the correct degree of the final geometric sequence member must be to generate the sequence using the diagonal method.

$$DiagonalHighestExponent(d) = 2d \tag{25}$$

Even though this method requires doubling the length of the input array, the relation still remains linear while this guarantees that only a single iteration is required n = 1.

Depending on the level of acceptable relative error, for approximating the algebraic solutions for these root problems, some iterations can be saved. This also is able to save iterations when solutions to the root formulas are integers.When ran on integers, 4 to 100000, it was able to only save iterations on .4% of the the numbers with about 444 integers having iteration saves. Which on average saved about 13 iterations or about 93% of the iterations that would have occurred otherwise. When increasing the acceptable error to 10^{-8} it was able to reduce time from about 14% of integers. Increasing the error yet again to 10^{-4}, it was able to save iterations on 99.7% of the integers while still being within this relative error threshold.

The diagonal geometric sequence solver is available at 'Section_7/diagonal_solver.py' or within the command line program at 'Geometric Sequence Solvers/Diagonal Solver'.

The diagonal solver saved iteration measurement function is available at 'Section_7/diagonal_tester.py' or within the command line program at 'Analysis/Diagonal Tester'.

8 General weights and single iteration solvers

The previous sections have been able to provide formal analysis on the generation of geometric sequences of positive integers. The main function responsible for creating these sequences has been the midpoint function as defined across an array as a cumulation of element wise midpoints. The element wise midpoint is of course defined as follows $(\frac{a_0+a_1}{2})$ where a_0 and a_1 are the real number elements. This is equivalent to the following linear combination $\begin{matrix}\frac{1}{2} & \frac{1}{2}\end{matrix}\begin{matrix}a_0 \\ a_1\end{matrix}$ of weights. This section will expand the methodology to include not only a generalized set of binary weights $\begin{matrix}c_0 & c_1\end{matrix}\begin{matrix}a_0 \\ a_1\end{matrix}$, but also generalize this more broadly extending it to include and number of weights as well $\begin{matrix}c_0 & ... & c_m\end{matrix}\begin{matrix}a_0 \\ ... \\ a_m\end{matrix}$.

8.1 General solution for arbitrary kernels and inputs

This section will define formulas for an arbitrary amount of arbitrary weights.

8.1.1 General array function

The function that generalizes the concepts of the midpoint and anti midpoints are quite easily constructed. They are in place algorithms that allow for the correlation of the weights kernel to the elements of the array. C is the weights list while A is the input vector.

$$GeneralArrayCorrelation(A, C) = (A \star C) = \begin{cases} \bigcup_{i=0}^{|A|-|C|} [\sum_{j=0}^{|C|-1} c_i a_{i+j}] & |A| \geq |C| \\ A & |A| < |C| \end{cases} \tag{26}$$

This defines the correlation between A the input and C which acts as a sliding window across the input.

Successive application of the generalized formula yield different matrices than the square cascade matrices generated by midpoints and anti midpoints. The matrix will possess its initial row of A which is the same as before, but there can only be |A| - |C| successive iterations applied. This means that the matrix will be of dimensions |A| x $\lfloor \frac{|A|-1}{|C|-1} \rfloor + 1$ which has a height that terminates after no more applications can be applied. Clearly the only kernels that will make a square matrix are those with only two weights. The number of non zero elements within each row also changes to be the following formula where j is the column number. In the general case, non zero elements could more generally be described as uninitialized or non null values since 0 is a valid value for the interior of the matrix since these no longer explicitly concern geometric sequences. This also assumes that each matrix is initialized with null pointers or null values instead of 0. However the term non zero and non null generally will be regarded to be equivalent.

$$NonNullRowCount(i) = |A| - (i-1)(|C|-1) \tag{27}$$

The number of elements within the entire matrix is then equal to $\sum_{i=0}^{\lfloor \frac{|A|-1}{|C|-1} \rfloor + 1} |A| - (i-1)(|C|-1)$ which simply counts the non null elements within each row.

Proof. Time Complexity of Matrix Generation

$\sum_{i=0}^{\lfloor \frac{|A|-1}{|C|-1} \rfloor + 1} |A| - i|C| + i + |C| - 1$

$\sum_{i=0}^{\lfloor \frac{|A|-1}{|C|-1} \rfloor + 1} |A| + |C| - 1 + i(1-|C|)$

$(\lfloor \frac{|A|-1}{|C|-1} \rfloor + 2)(|A| + |C| - 1) + (1-|C|) \sum_{i=0}^{\lfloor \frac{|A|-1}{|C|-1} \rfloor + 1} i$

$(\lfloor \frac{|A|-1}{|C|-1} \rfloor + 2)(|A| + |C| - 1) + (1-|C|) \frac{(\lfloor \frac{|A|-1}{|C|-1} \rfloor + 1)^2 + \lfloor \frac{|A|-1}{|C|-1} \rfloor + 1}{2}$

Let $\lfloor \frac{|A|-1}{|C|-1} \rfloor + 1 = n$

$(n+1)(|A| + |C| - 1) + (1-|C|) \frac{n^2+n}{2}$

$= n|A| + |A| + n|C| + |C| - n - 1 + \frac{n^2+n}{2} - |C| \frac{n^2+n}{2}$

$= \frac{n^2}{2} + \frac{n}{2} - \frac{|C|n^2}{2} - \frac{|C|n}{2} + n|A| + n|C| - n + |A| + |C| - 1$

$= \frac{n^2}{2} - \frac{|C|n^2}{2} + n|A| - \frac{n}{2} + \frac{|C|n}{2} + |A| + |C| - 1$

$= n^2(\frac{1}{2} - \frac{|C|}{2}) + n^1(|A| - \frac{1}{2} + \frac{|C|}{2}) + n^0(|A| + |C| - 1)$

This clearly is in $O(n^2)$ since each element requires only a single convolution from the previous row to be calculated.

Applying the Discrete Fourier Transform with an input and general kernel would also likely provide an $O(nlgn)$ time complexity since the operation is simply a convolution that would be equivalent to a simple multiplication within the transformed domain.

The general kernel cascade matrix function is available at 'Section_8/general_kernel_cascade_matrix.py' or within the command line program at 'Matrix Functions/General Kernel Cascade Matrix'.

8.1.2 Generalized Leading Edge

A part of solving the matrix that is created when using a generalized kernel involves finding each element's value based on position in terms of the input. Each row of the matrix is simply comprised of the GeneralArrayCorrelation function of the non null elements of the previous row. The arrangement of the matrix rows will now be dependent on the size of the initial input array. An example will be shown to describe the general case with a matrix on a kernel of size 3. The input array values are simply used to represent general variable values. The matrix will be deconstructed since the sums are quite large.

Example 9. Finding the first row 3 element of the matrix for the purpose of a demonstration

$A = M[1] = [a_0 a_1 a_2 a_3 a_4 a_5]$

$[a_0 a_1 a_2 a_3 a_4 a_5] \star [c_0, c_1, c_2]$ = M[2]=[$c_0a_0+c_1a_1+c_2a_2$,$c_0a_1+c_1a_2+c_2a_3$,$c_0a_2+c_1a_3+c_2a_4$,$c_0a_3+c_1a_4+c_2a_5$]

$(M[2] \star C)[1]$ (assuming 1 indexed array is a matrix row) $= [c_0(c_0a_0 + c_1a_1 + c_2a_2) + c_1(c_0a_1 + c_1a_2 + c_2a_3) + c_2(c_0a_2 + c_1a_3 + c_2a_4)]$

$= c_0^2a_0 + c_0c_1a_1 + c_0c_2a_2 + c_0c_1a_1 + c_1^2a_2 + c_1c_2a_3 + c_0c_2a_2 + c_1c_2a_3 + c_2^2a_4$

$= c_0^2a_0 + c_1^2a_2 + c_2^2a_4 + 2c_0c_1a_1 + 2c_0c_2a_2 + 2c_1c_2a_3$

$= c_0^2a_0 + c_1^2a_2 + c_2^2a_4 + 2c_0c_1a_1 + 2c_0c_2a_2 + 2c_1c_2a_3$

$= \sum_{i=0}^{|C|-1} \sum_{j=0}^{|C|-1} c_ic_ja_{i+j}$

This trinomial expansion demonstrates that this will be more of an advanced counting problem than the case in which the length of the kernel is of length 2. The expansion of this element suggests that a general solution to the matrix will involve The Multinomial Theorem. The theorem is as follows.

Theorem 3. *The Multinomial Theorem*

$(\sum_{i=0}^{|A|-1} a_i)^k = \sum_{(\sum_{i=0}^{|A|-1} n_i)=k} \frac{k!}{\prod_{j=0}^{|A|-1}(n_j!)} \prod_{j=0}^{|A|-1}(a_j^{n_j})$

This theorem is rather tedious to use and inefficient to implement since the sheer number of terms involved in each succession is quite large in the general sense. There are quite a few steps and stages within the theorem. As can be seen the actual number of terms is dependent on a number of factors. In practice this will not be used as the performing successive convolutions with the symmetric kernel is equivalent and more intuitive. In his 1996 paper "Multinomial convolution polynomials", Zeng extends the notions of binomial convolution used previously within Section 5.1 of this paper to the multinomial case. These multinomials represent the "multinomial convolution family" in which the "multi-convolution polynomials arise as coefficients" of the multinomial definition. [4]

Lemma 4. *Taking the convolution of multiple convolutions on incrementing index slices is equivalent to taking a single convolution over the entire range.*

A = input array, K = kernel, k = /K/

$[(A[j : j+k-1]*K), (A[j+1 : j+k]*K), ..., (A[j+k-1 : j+2k-2]*K)]*K = A[j : j + 2k - 2] * (K * K)$

First verification of same dimensions.

LHS: Each inner valid convolution will create a singular scalar. There are (j+k-1)-j +1, k elements within the inner array that is a valid convolution again with K to produce a scalar.

*RHS (K*K) will produce a resultant kernel of length 2k-1 using full convolution. The inner array will house* j+2k-2 - j + 1, 2k - 1 elements. The valid convolution between these will produce a scalar.

The following would represent the LHS of a kernel of length n+1, $[(A[j : j + n] * K), (A[j + 1 : j + n + 1] * K), ..., (A[j + n : j + 2n] * K)] * K$.

$= \sum_{m=0}^{n}(\sum_{p=0}^{n} A[j + m + p]k_{n-p})k_{n-m}$ *(Definition of convolution)*

$= \sum_{m=0}^{n}(\sum_{p=0}^{n} A[j + m + p]k_{n-p}k_{n-m})$

Since the full range of elements of A that are being used ranges from A[j:j+2n], these terms can be collected and the summations can be reindexed.

$= \sum_{s=0}^{2n} A[j + s](\sum_{m+p=s} k_{n-p}k_{n-m})$

$= A[j : j + 2n] * (K * K)$

Therefore taking the multiple valid convolutions between a kernel and array slice increments then performing a valid convolution upon those values is equivalent to taking the valid convolution of the entire range of slices with the full convolution of the kernel with itself.

Proof. Determining that successive applications of the kernel will result in the multinomial theorem

Since these kernels C are not cascade, in terms of one dimensional signal processing, taking the convolution is not equivalent to taking the cross correlation $(A * C) \neq (A \star C)$. This can be mitigated by simply reversing C to be C', now $(A \star C) = (A * C')$. Let c = |C| and a=|A|. Also for the purposes of this proof, both A and C will be 1 indexed arrays to be in line with the matrix.

A recurrence relation needs to be defined in order to properly map out the general matrix.

$$M_{i,j} = \begin{cases} \sum_{k=1}^{c} M_{i-1,j+k-1} C'_k & i \neq 1 \\ A_j & i = 1 \end{cases} = \begin{cases} (M_i[j : j + c - 1] * C') & i \neq 1 \\ A_j & i = 1 \end{cases}$$

This recursively takes the convolution between proper elements from the previous row. Each row of the matrix relies on the previous row's elements and the proper element within the reversed kernel.

Since each element is dependent on the convolution of the previous row's matching elements, which are in turn dependent on the previous row's convolution etc, these convolutions can be commuted.

$M_{2,j} = A[j : j + c - 1] * C')$

$M_{3,j} = [(A[j : j + c - 1] * C'), (A[j + 1 : j + c] * C'), \ldots, (A[j + c - 1 : j + 2c - 2] * C')] * C'$

$= [A[j : j + c - 1], A[j + 1 : j + c], \ldots, A[j + c - 1 : j + 2c - 2]] * C' * C'$

$= A[j : j + 2c - 2] * C' * C'$

It seems that the following is true.

$$M_{i,j} = \begin{cases} A[j : j + (i - 1)(c - 1)] * C'^{i-1} & i \neq 1 \\ A[j] & i = 1 \end{cases} \tag{28}$$

Where C'^{n}corresponds to taking the convolution with C' against itself n times.

Proof By Induction

Base case: i = 2

$M_{2,j} = (M_i[j : j + c - 1] * C')$

$= (A[j : j + c - 1] * C')$

$= (A[j : j + (2 - 1)(c - 1)] * C')$

Inductive step: Assume that for i = k, $M_{k,j} = A[j : j + (k-1)(c-1)] * C'^{k-1})$, then when i = k+1 $M_{k+1,j} = A[j : j + (k)(c - 1)] * C'^{k+1})$

Since this defines each element from the previous row, $M_{k,j} = A[j : j + (k - 1)(c - 1)] * C'^{k-1})$, $M_{k+1,j}$ is simply the convolution between the correct elements of $M_k$with C'.

The elements that will be convolved with C' to yield $M_{k+1,j}$ are $[M_{k,j}, M_{k,j+1}, ..., M_{k,j+c-1}]$.

$M_{k+1,j} = [M_{k,j}, M_{k,j+1}, ..., M_{k,j+c-1}] * C'$
$=[A[j : j + (k-1)(c-1)] * C'^{k-1}), ..., A[j + c - 1 : j + c - 1 + (k-1)(c-1)] * C'^{k-1})] * C'$
$= [A[j : j + (k-1)(c-1)] * C'^{k-1}), ..., A[j + c - 1 : j + k(c-1)] * C'^{k-1})] * C'$
$= [A[j : j + (k-1)(c-1)]), ..., A[j + c - 1 : j + k(c-1)])] * C^{k\prime}$ (Lemma 4)
$= A[j : j + (k)(c-1)] * C'^{k+1}$

Since Formula 28, is defined as successive convolutions within the time domain, would lead to the exponentiation of the transformed kernel weights within the frequency domain. This clearly would need to utilize the multinomial theorem since the correspondent C'^{i-1}, will represent the weights as a multinomial in the frequency domain multiplied by the transformed corespondent slice of A.

With the knowledge of Formula 28 in mind, solving the entire general matrix, it seems that performing the explicit multinomial theorem is not necessary and simply taking convolutions upon an input range is more effective.

Additionally, this allows this structure to be classified as a group with the convolution operator acting upon the space of all geometric sequences of length k + 1, and the midpoint/antimidpoint kernels.

- **Identity.** It has been shown that the geometric sequence of base a = 1, b = 0, is equivalent to the Kronecker Delta where the first element is 1 and everything else is 0. Convolving any sequence A with the Kronecker Delta will return A, $A = (A * \delta)$.

- **Inverse.** It has been demonstrated that for some input sequence of base b, an element of $b' = \frac{b+1}{2}$. Applying the anti midpoint kernel to this sequence will return the sequence to its original base. The anti midpoint kernel acts as the inverse by mapping $\frac{b+1}{2} \rightarrow -1 + 2\frac{b+1}{2} = -1 + b + 1 = b$, which returns the initial base.

- **Associativity.** Since the operator defining this group is convolution, the associativity property holds for this structure.

- **Closure.** The midpoint kernel $[\frac{1}{2}, \frac{1}{2}]$ represents the constant geometric sequence a = 1, b = $\frac{1}{2}$, k = 1. The anti midpoint kernel [-1, 2] represents the geometric sequence a = -1, b = -2, k = 1. Therefore both the forwards and inverse operations are geometric sequences. When operating either of these two sequences together with any arbitrary sequence, it has been shown that a sequence is returned.

A simple valid mode cross correlation function (dot product) is available at 'Section_8/cross_correlation.py' or within the command line program at 'Array Functions/Cross Correlation'.

The general kernel using convolution cascade matrix function is available at 'Section_8/kernel_convolution_cascade_matrix.py' or within the command line program at 'Matrix Functions/Kernel Convolution Cascade Matrix'.

8.1.3 Repeated Convolution Example

Example 10. Application of Formula 28.

This application will attempt to find $M_{3,4}$ using only Formula 28. The following represents the fully solved matrix with C = $[\frac{1}{2}, \frac{1}{2}]$ = C' and A = $[3^0, 3^1, 3^2, 3^3, 3^4, 3^5]$. |C| = k = 2

1	3	9	27	81	243
2	6	18	54	162	0
4	12	36	108	0	0
8	24	72	0	0	0
16	48	0	0	0	0
32	0	0	0	0	0

Using Formula 28 states that $M_{3,4} = A[4 : 4 + (3-1)(2-1)] * C'^{3-1}$

$= A[4:6] * C^2$

$[27, 81, 243] * C^2$

Finding the twice full convolution of C is $[\frac{1}{2}, \frac{1}{2}] * [\frac{1}{2}, \frac{1}{2}]$. Any bound not contained within the kernel that the summation might find is assumed to be zero for padding. Assuming C is 0 indexed.

$= \bigcup_{s=0}^{|C|} [\sum_{m+p=s} c_{k-1-p} c_{k-1-m}]$

$= [\sum_{m+p=0} c_{k-1-p} c_{k-1-m}, \sum_{m+p=1} c_{k-1-p} c_{k-1-m}, \sum_{m+p=2} c_{k-1-p} c_{k-1-m}]$

The 0^{th} element

$\sum_{m+p=0} c_{k-1-p} c_{k-1-m}$

$= c_{k-1}^2$

$= \frac{1}{4}$

The 1^{st}element

$\sum_{m+p=1} c_{k-1-p} c_{k-1-m}$

$= c_{k-1}c_{k-2} + c_{k-2}c_{k-1}$

$= 2c_{k-1}c_{k-2}$

$= 2 * \frac{1}{2} * \frac{1}{2}$

$= \frac{1}{2}$

The 2^{nd} element (Even though by the symmetry of kernel, this must equal $\frac{1}{4}$, the calculation will still be shown)

$\sum_{m+p=2} c_{k-1-p} c_{k-1-m}$

$= c_{k-1-0}c_{k-1-2} + c_{k-1-2}c_{k-1-0} + c_{k-1-1}c_{k-1-1}$

$= c_{k-1}c_{k-3} + c_{k-3}c_{k-1} + c_{k-2}c_{k-2}$

$= c_{k-2}^2$

$= \frac{1}{4}$

$C * C = [\frac{1}{4}, \frac{1}{2}, \frac{1}{4}]$

Now $[27, 81, 243] * [\frac{1}{4}, \frac{1}{2}, \frac{1}{4}]$ (Note this does not mean the multiplication between the vectors however the valid convolution is equivalent to the dot product) This is the sum of element wise multiplication

$= \frac{1}{4} * 27 + \frac{1}{2} * 81 + \frac{1}{4} 243$

$= 108$

This shows that the Formula 28, retrieved the identical answer to the previous midpoint/anti midpoint methods.

Another example could use the same input, but switch out the kernel of $[\frac{1}{2}, \frac{1}{2}]$ to be $[\frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$ or even $[\frac{1}{4}, \frac{1}{4}]$. Seeing the effects upon an arbitrary input is not too useful as it already has been solved for every position within the matrix. Determining the generation of leading edges using geometric sequence inputs with this methodology could be helpful. Potentially Formula 28, can be simplified under this input constraint which could generalize Formula 10 and Formula 11.

8.2 General Leading Edge Formula

8.2.1 General Formula

Defining the input as a geometric sequence of $a*b^n$, Formula 28 can be simplified assuming M is a 1 indexed matrix,$M_{i,j} = \begin{cases} (a \bigcup_{s=0}^{j+(i-1)(k-1)-1} [b^{s+j-1}]) * C'^{i-1} & i \neq 1 \\ a * b^{j-1} & i = 1 \end{cases}$

where C' is the rotated kernel, a is a scalar, b is the base of the sequence and |C|=k. Including only the leading edge (the first column) will simplify this even further.

$M_{i,1} = \begin{cases} (a \bigcup_{s=0}^{(i-1)(k-1)} [b^s]) * C'^{i-1} & i \neq 1 \\ 1 & i = 1 \end{cases}$

The valid convolution can be treated as a dot product between the sequence and the kernel.

$M_{i,1} = a \sum_{s=0}^{(i-1)(k-1)} b^s C_s'^{i-1}$ when $C'^0 = [1]$

Since by Lemma 4, it was demonstrated that taking successive full convolutions of a kernel against itself will result in the coefficients of the polynomial $P(x) = (\sum_{s=0}^{k-1} c_s x^s)^{i-1}$ where c_s represents the value of $C_s'^{i-1}$, simply taking x = b, will result in an equivalency to this $M_{i,1}$

$= aP(b)^{i-1}$

Therefore the general leading edge generation formula is of the following form.

$a * MidpointLeadingEdge(\bigcup_{i=0}^{n-1} [b^i]) = \bigcup_{i=0}^{n-1} [a(\frac{b+1}{2})^i]$

$$a * GeneralLeadingEdge(\bigcup_{i=0}^{n-1} [b^i]) = \bigcup_{i=0}^{n-1} aP(b)^{i-1} \tag{29}$$

Where $P(x) = (\sum_{s=0}^{k-1} c_s x^s)^{i-1}$ or the full convolution of the kernel with itself i times then with the geometric sequence.

This shows that any geometric sequence input with any arbitrary kernel will achieve a leading edge geometric sequence.

By using this formula, it is sufficient to simply calculate $P(b)$ or the kernel convolved against the first k element in the input to find out the base of the sequence. Additionally, WLOG, a = 1.

$$ResultingBase(b, C) = P(b) = b' = \sum_{n=0}^{|C|-1} b^n C_n \tag{30}$$

The proposed examples $C = [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}], [\frac{1}{4}, \frac{1}{4}]$ will now be simply demonstrated.

Example 11. $C = [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$

This will first be demonstrated on a generic base b.

$b' = b^0\frac{1}{2} + b^1\frac{1}{2} + b^2\frac{1}{2}$

$= \frac{b^0+b^1+b^2}{2}$

This kernel represents a triple midpoint of sorts, but this kernel does not represent a weighted mean and would have to be normalized to be considered such.

Following the original Example 10, b = 3. So this kernel would result in $b' = \frac{3^0+3^1+3^2}{2} = 6.5$

For kernels of equal weights, the following identity can be used.

$$EqualWeightedKernel(b, C) = b' = c_0 \sum_{n=0}^{|C|-1} b^n = c_0 \frac{b^{|C|} - 1}{b - 1} \tag{31}$$

So $b' = \frac{1}{2}\frac{3^3-1}{3-1} = 6.5$

Example 12. $C = [\frac{1}{4}, \frac{1}{4}]$, b = 3

Using Formula 31.

$b' = \frac{1}{4}\frac{3^2-1}{3-1} = 1$

This is interesting as it would suggest that using these weights would provide a direct step from 3 to 1, the kernel that would be corespondent to inverse of this operation could be found which will be discussed in Section 8.2.2.

Proof. Finding if the the cascade matrix is equivalent to the outer product of the input and output vectors.

By Formula 30, $b' = \sum_{n=0}^{|C|-1} b^n C_n$ and by Formula 28, $M_{i,j} = \begin{cases} A[j : j + (i-1)(c-1)] * C'^{i-1} & i \neq 1 \\ A[j] & i = 1 \end{cases}$

Therefore $M_{i,j} = b'^{i-1}b^{j-1} = (\sum_{n=0}^{|C|-1} b^n C_n)^{i-1} b^{j-1}$.

When i,j=0, this has already been shown to be the case since these represent the inputs or the leading edge which are purely geometric sequences of b or b'.

This means that the problem is simplified to $(\sum_{n=0}^{|C|-1} b^n C_n)^{i-1} b^{j-1} = \bigcup_{s=0}^{(i-1)(c-1)} [b^{s+j}] * C'^{i-1}$

P(b) $= \sum_{n=0}^{|C|-1} b^n C_n$

$(P(b))^{i-1} = \sum_{s=0}^{(i-1)(|C|-1)} C'^{i-1}_s b^s$ (By The Multinomial Theorem)

So $(\sum_{n=0}^{|C|-1} b^n C_n)^{i-1} b^{j-1} = \sum_{s=0}^{(i-1)(|C|-1)} C'^{i-1}_s b^s$

This is equivalent to the convolution within the RHS.

8.2.2 General Kernel Solutions

Since this method always produces a geometric sequence upon the leading edge, an easy shortcut can be taken.

An inverse kernel K must be found to satisfy the following condition.

$\sum_{i=0}^{|K|-1}(\sum_{n=0}^{|C|-1} b^n C_n)^i K_n] = b$

By having K not necessarily being the same length as C, too many free variables are introduced, but by imposing an additional constraint |K|=|C|, some simplification can occur.

$\sum_{i=0}^{|C|-1}(\sum_{n=0}^{|C|-1} b^n C_n)^i K_n] = b$

$=\sum_{i=0}^{|C|-1}(b')^i K_i]$

This is identical to solving the linear system, just using a general b and C kernel, $[b'] = [b^0, b^1, ..., b^{|C|}] \cdot [c_0, c_1, ..., c_{|C|-1}]^t$

There a |C|-1 free variables within this system which only allows for a clearly defined kernel when |C|=1.

There are some constraints that can be imposed upon the kernel to allow for a potentially easier time finding a working kernel.

When a kernel can only possess a single repeated weight this system becomes simplified.

Using Formula 31, $c_0 \frac{b^{|C|}-1}{b-1} = b'$

Solving for the weight can be done easily.

$c_0 = \frac{b'(b-1)}{b^{|C|}-1}$, which gives the weight.

Other than this property, the weights can be arbitrarily defined as long as they follow the dot product condition properly equating to b'. Kernels might be chosen for a variety of reasons, ease of calculation, symmetry or to use a single repeated weight.

k=2

Assuming a defined b and b', then

$c_0 b^0 + c_1 b^1 = b'$

$c_0 + c_1 b = b'$

Now the inverse of this

$k_0 b'^0 + k_1 b'^1 = b$

$k_0 + k_1(c_0 + c_1 b) = b$ (Expressing b' in terms of b)

$k_0 + k_1 c_0 + k_1 c_1 b = b$

This can now be split up into two different equations separating the linear and constant terms using the method of undetermined coefficients.

(1) $k_0 + k_1 c_0 = 0$ and (2) $k_1 c_1 b = b, k_1 c_1 = 1, k_1 = \frac{1}{c_1}$ (No variables can be 0)

Plugging (2) into (1), $k_0 + \frac{c_0}{c_1} = 0, k_0 = \frac{-c_0}{c_1}$

Now $K = \frac{1}{c_1}[-c_0, 1]$, fully represented in terms of C.

This means that for the input kernel to have the arithmetic mean property that $c_0 + c_1 = 1$, $K = \frac{1}{1-c_0}[-c_0, 1]$ and $\frac{-c_0}{1-c_0} + \frac{1}{1-c_0} = 1$ is true. Meaning that an arithmetic mean kernel will result in an arithmetic mean inverse kernel as well.

This describes the relationship between the kernel C and the kernel K, that when applied successively will act as the inverse when both are of length 2.

When verifying this for the midpoint and anti midpoint weights, this correctly determines the inverse weights.

For anti midpoint $c_0 = -1, c_1 = 2$ -> $k_0 = \frac{1}{2}, k_1 = \frac{1}{2}$, but when using these k values as c it doesn't work.

for $c_0 = \frac{1}{2} = c_1 \to k_0 = -1, k_1 = 2$.

k=3

(1) $c_0b^0 + c_1b^1 + c_2b^2 = b'$ and (2) $k_0b'^0 + k_1b'^1 + k_2b'^2 = b$

Plugging (1) into (2)

$k_0 + k_1(c_0b^0 + c_1b^1 + c_2b^2) + k_2(c_0b^0 + c_1b^1 + c_2b^2)^2 = b$

$k_0 + k_1c_0 + k_1c_1b + k_1c_2b^2 + k_2(c_0 + c_1b + c_2b^2)^2 = b$

This already involves a quadratic trinomial that needs to be expanded which would need to use The Multinomial Theorem.

However the largest power of this will include only a single $(c_2b^2)^2, k_2c_2^2b^4$, but there clearly is no b^4 terms on the RHS so this means that $k_2c_2^2b^4 = 0$ so $k_2 = 0$ leading for an easy simplification of the rest.

$k_1c_2b^2 = 0$ $k_1 = 0$

$k_1c_1b = 1$, $0 = 1$

This contradiction determines that this type of inverse kernel is not possible.

Proof. Inverse kernels of length greater than 2 are not possible.

Assume k $>$ 2 produces a valid inverse kernel.

(1) $\sum_{i=0}^{k-1} b^ic_i = b'$ and (2) $\sum_{j=0}^{k-1} b'^jk_j = b$

Plugging in (1) into (2)

$\sum_{j=0}^{k-1}(\sum_{i=0}^{k-1} b^ic_i)^jk_j = b$

This inner sum will create some polynomial,P with weights p_i and degrees of b ranging from 0 to (k-1)j

$\sum_{j=0}^{k-1} P(b) = b$

$\sum_{j=0}^{k-1}\sum_{i=0}^{j(k-1)} p_ib^i = b$

These two sum will create yet another polynomial W with coefficients $w_i = \sum_{j=\lceil \frac{i}{k-1} \rceil}^{k-1} K_jC_i'^j$ with degrees ranging from 0 to $(k-1)^2$

$\sum_{j=0}^{(k-1)^2} W(b)$

$\sum_{j=0}^{(k-1)^2} w_jb^j = b$

Clearly using the method of undetermined coefficients, the RHS only contains a singular linear term of b. This would result in all of the weights w_j where $j > 1$ to be $w_jb^j = 0, w_j = 0$. When j = 1, $w_1b^1 = 1$ which cannot be true since by the definition of $w_1 = \sum_{j=\lceil \frac{1}{k-1} \rceil}^{k-1} K_1C_1'^j = \sum_{j=1}^{k-1} K_jC_1'^j$, w_1 depends on all of the other weights from 1 to k-1 which were determined to be 0 for j $>$ 1. so $w_1 = K_1C_1' = 1$. Although since it was determined that $w_{j>1} = 0 = \sum_{j=1}^{k-1} K_jC_i'^j$, all K_j must be zero in order to fulfill this since each term is linearly independent. This results in $w_1 = 0$ which is false since it was already determined to be 1.

Due to this property, only kernels of size 2 would be ideal for performing back and forth calculations. Kernels of a larger size might not be guaranteed to have an inverse kernel that would undo the operation.

8.3 Specific Kernels

8.3.1 Binary Kernel

Since the of these operations are geometric sequences and binary numbers are simply linear combinations of geometric sequences of base 2, it would be possible to build a kernel that takes advantage of this.

Lets assume a base b = 2 and a kernel C, which is the binary representation for some number b' (For example b' = 6 -> C = [0,1,1]) where the least significant bit is the left most bit. Applying this kernel upon the base $[2^0, 2^1, 2^2...]$ should produce an interesting result. Since this is a kernel C of arbitrary length and it is being operated upon a geometric series, the resulting leading edge must be another geometric sequence. This assumes that b' is a positive integer.

Proof. The leading edge provided by successive operating on a binary kernel representing b' with the least significant bit first, upon a base 2 input results in the geometric sequence of the base of b'.

Using Formula 30, The base of the leading edge is equal to the following $\sum_{n=0}^{|C|-1} b^n C_n$.

Since b = 2, then $\sum_{n=0}^{|C|-1} 2^n C_n$.

C_n represents the binary version of b'. So this sum would match the least significant bit of b' with 2^0 and the rest of the bits with their corresponding power of 2 to fully recover b'.

Therefore using the binary representation of b' as the base yields a leading edge geometric sequence base of b'.

Example 13. Binary base with b' = 10

This result should not be very surprising since the cascade matrix has been shown to be is equivalent to the outer product of the input and leading edge..

$$\begin{array}{lccccccc} & 2^0 & & 2^1 & 2^2 & 2^3 & 2^4 & 2^5 & 2^6 \\ \text{Using b'} = 10, \text{k=[0101]} & 2^0 + 2^3 & & 2^1 + 2^4 & 2^2 + 2^5 & 2^3 + 2^6 & & & \\ & (2^1 + 2^4) + (2^3 + 2^6) & & & & & & & \end{array}$$

This matrix is equivalent to

$$\begin{array}{ccccccc} 2^0 & & 2^1 & 2^2 & 2^3 & 2^4 & 2^5 & 2^6 \\ 2^0(10) & & 2^1(10) & 2^2(10) & 2^3(10) & & & \\ 2^2(10) + 2^3(10) = 10(10) & & & & & & & \end{array}$$

A solver could be created to perform this task as this kernel guarantees that b' will be reached in a single iteration. If a geometric sequence of highest exponent k would be found, the height would need to be equal to k + 1.

Proof. Length of input when using a binary kernel to generate a geometric sequence

$k + 1 = \lfloor \frac{|A|-1}{|C|-1} \rfloor + 1$

$k = \lfloor \frac{|A|-1}{|C|-1} \rfloor$

Since C is simply the binary representation of b', then |C| = bit length of b'.

$k = \lfloor \frac{|A|-1}{log_2(b')-1} \rfloor$
$k \leq \frac{|A|-1}{log_2(b')-1} \leq k+1$
Case 1:
$k \leq \frac{|A|-1}{log_2(b')-1}$
$klog_2(b') \leq |A|$
Case 2:
$\frac{|A|-1}{log_2(b')-1} \leq k+1$
$|A|-1 \leq (log_2(b')-1)(k+1)$
$|A|-1 \leq klog_2(b') + log_2(b') - k - 1$
$|A| \leq klog_2(b') + log_2(b') - k$

|A| being within this margin will guarantee that there is enough input values to convolve with. Although this might not be an integer which is necessary as an array length so simply using

$|A| = \lceil klog_2(b') + log_2(b') - k \rceil$ will suffice.

The following program uses these principals to accurately compute the geometric sequence of an integer in a single iteration. This leads to a time complexity in $O(k^2)$, since the singular iteration is able to completely remove the lnb' memoization procedure.

The binary base solver is available at 'Section_8/binary_base.py' or within the command line program at 'Geometric Sequence Solvers/Binary Base'.

8.4 General base b

The midpoint methods were defined to use either an initial b = 1 for positive integers and rational numbers, or b = 0 for negative integers. This meant that the initial base after the initial transformation was b = 2 for positive integers, b = $\frac{1}{2}$ for rational numbers and $b = -1, -2$ for negative integers. This was shown to be a valid way to calculate an floating point valued real number as a geometric sequence. Calculating for example the geometric sequence of b' would require $\lceil log_2 b' \rceil$ full iterations of matrix solves or element wise multiplication by the base 2 geometric sequence. Could starting from some other base potentially result in a memoization of bit length $\lceil log_b b' \rceil$? Would this also mean that the memoization would be a base b number? This section will attempt to generalize the midpoint and anti midpoint base building techniques and memoizations.

8.4.1 Defining the general kernel

It has been shown that any arbitrary kernel can be used, but in order for the memoization to be easily transferable into a base b, string certain conditions must be met. The initial base should start from b = 1, but the only thing that is different would be the weights of the kernel. Defining a kernel of length 2 is also preferred since an easily achievable inverse kernel can be found as well.

Let C = $[c_0, c_1]$, then the resulting base would be $b' = c_0 + bc_1$ or in the inverse kernel,$K = \frac{1}{c_1}[-c_0, 1]$, b = $\frac{-c_0}{c_1} + \frac{b'}{c_1}$, since b' is equal to the weights

applied to the first elements of the sequence. This inverse can be verified b = $\frac{-c_0}{c_1} + \frac{c_0+bc_1}{c_1}$.

Now a kernel must be constructed to take b = 1 to the arbitrary b' and b' to b.

$b' = c_0 + c_1$

This still leaves several options for the kernel and does not simplify the search much at all. Another condition must be applied as well. When previously creating the constant memoization, the following technique was used. $f(s,n) = b = s2^{n+1} - \sum_{i=0}^{n} 2^{n-i} m_i .where \lceil log_2(b) \rceil = n+1$. This helped define the memoization as successive applications of the anti midpoint function upon the input $[m_i, s]$ where s is the starting point and m_i is the i^{th} memoization bit. This meant that when the memoization was set to be equal to 0, the c_0 term was not present. This zero memoization is what encoded the jump from 1 to 2 as simply a multiplication by 2. The same attribute can be applied to the general case, wherein the 0 memoization must be able to take b to b' only with c_1.

Applying this to the general case. $b' = c_0 + c_1 = c_0 m_i + c_1 b$

Setting this memoization bit to 0 $c_1 b = b'$. Since the b has been set to 1 in the initial clause for this section, $c_1 = b'$. This allows for the kernel and inverse kernel to be totally solved. $b' = c_0 + b', c_0 = 0$, but this solution is not helpful, the memoization must be able to influence the output base. Simply setting $c_0 = c_1 = b'$ is a suitable solution.

This will allow the forwards kernel to be as simple as $C = [b', b']$ and the inverse kernel to be $K = [-1, \frac{1}{b'}]$.

8.4.2 Memoization

The memoization can be described as taking the C weights to b until b' is reached or taking the K weights to b' until b is reached. The meaning of b' is now the final base of the geometric sequence after multiple iterations. b=s = 1.

$b' = (c_0 m_n ... c_1 (c_0 m_1 + c_1 (c_0 m_0 + c_1 s)))$

n=0

$c_0 m_0 + c_1 s$

n=1

$c_0 m_1 + c_1 (c_0 m_0 + c_1 s)$

$c_0 m_1 + c_1 c_0 m_0 + c_1^2 s$

n=2

$c_0 m_2 + c_1 (c_0 m_1 + c_1 c_0 m_0 + c_1^2 s)$

$c_0 m_2 + c_1 c_0 m_1 + c_1^2 c_0 m_0 + c_1^3 s$

It seems like the following pattern is true. $b' = s c_1^{n+1} + c_0 \sum_{i=0}^{n} m_i c_1^i$

Isolating the memoization terms.

$\frac{b' - s c_1^{n+1}}{c_0} = \sum_{i=0}^{n} m_i c_1^i$ where $\lceil log_2(b) \rceil$ = n+1

$$GeneralMemoization(b, b', s, c_0, c_1) = \frac{b' - s c_1^{\lceil log_b(b') \rceil}}{c_0} \tag{32}$$

b = the base of the memoization encoding

b' = the final desired base of the geometric sequence

s = the initial starting point

c_0 = first weight in the kernel

c_1=second weight in the kernel

This general memoization describes the memoization for any two element kernel in the contractive mode when b < 1.

Since C $= [c_0, c_1] = [b, b]$, this can be simplified to be $b' - sb^{\lceil log_b(b') \rceil}$.

The inverse kernel K $= [-1, \frac{1}{b}]$ in the expansion mode when b < 1, can be simplified from $\frac{b' - s\frac{1}{b}^{\lceil log_b(b') \rceil}}{-1}$ to $s(b)^{-\lceil log_b b' \rceil} - b'$.

The final piece that allows a program to be created with this information is for rational numbers to be covered.

They will be treated identically to the b = 2, case. The memoization will be simply $(a-1)_b$, as was before. Additionally, the approximation must be slightly altered from the b = 2 case. The previous approximations were trying to approximate the rational number between 0 and 1 to be of the form $\frac{a}{2^n}$, however this clearly is in the incorrect base, therefore the approximation is found in the form $\frac{a}{b^n}$. The proof of the error bounds remains almost entirely the same from Section 6.3.1, but the 2^n scale factor is of course changed to b^n, which does not effect the bounds other than that simple substitution. Finally the scaling that must occur for rational numbers over 1 must be changed to be of factors of b^n.

The general base memoization is available at 'Section_8/general_memoization.py' or within the command line program at 'Memoization Functions/-General Memoization'.

The general base approximation function is available at 'Section_8/base_b_rational_approxi or within the command line program at 'Rational Approximation Functions/Base b Rational Approximator'.

The general base geometric sequence solver is available at 'Section_8/general_base.py' or within the command line program at 'Geometric Sequence Solvers/General Base'.

8.5 Dynamically Assigned General Base

The base parameter within the function, adds additional control that is quite important for creating more efficient integer memoizations and precise rational bases that do not need to be approximated. Previously with b = 2, finding the geometric sequence for something as simple as $\frac{1}{3}$ would require 54 iterations with full 10^{-16} precision approximated as $\frac{6004799503160661}{2^{54}}$, which is correct, but in b = 3, this can be found with a single iteration $\frac{1}{3^1}$. This method does not alter the time complexity since the memoization is of the length $\lceil log_b b' \rceil$, which from the Section 8.1.1 proof, executes a total of

$\lceil log_b b' \rceil (n^2(\frac{1}{2} - \frac{|C|}{2}) + n^1(|A| - \frac{1}{2} + \frac{|C|}{2}) + n^0(|A| + |C| - 1))$,

$\lceil \frac{lnb'}{lnb} \rceil (\frac{n^2}{2} + n^1(k + 1 + \frac{1}{2}) + k)$ operations. (|C| = 2, |A| = k+1)

This still tends to be within $O(n^2 lnb')$.

This does not alter then time complexity, but it can alter the number of operations quite considerably.

8.5.1 Finding the optimal base

Clearly it is easy to manually adjust the base that should be used. The base should be the largest possible that does not have more numerical representations than the b'. This directly points to using b = b' for integers which will solve the entire sequence in a single iteration with memoization of 0. For rational numbers, the denominator should be used as b, which will exactly yield b' with no need for approximation. Using these large arbitrary bases is equivalent to just performing, $\prod_{i=0}^{n} b^i$. This uses costly multiplications by large numbers. However applying a constraint upon b where it must be equal to a power of 2, allows for successive bit shifting an subtraction to produce the geometric sequence. This is more in line with the core ideas behind the genesis of this paper. Leveraging the speed of the bit shifts to calculate geometric sequences.

8.5.2 Bases in the form 2^t

This will transform b from an external parameter, to be dynamically calculated and dependent on b'. The most optimal solution will be defined as the base that yields the shortest memoization with the minimum base. This can be done in a brute force method using Formula 32, to simply find the first base that follows this criteria.

Example 14. b = $\frac{1}{2}^t$, t is an increasing natural number, b' = 10. s = 1 for integer.

The general memoization function can be altered to the following
$2^{t\lceil log_{2^t} b' \rceil} - b'$
t=1
$2^{\lceil log_2 10 \rceil} - 10 = 6, [0, 1, 1, 0]$
t=2
$2^{2\lceil log_4 10 \rceil} - 10 = 6, [1, 2]$
t=3
$2^{3\lceil log_8 10 \rceil} - 10 = 54, [6, 6]$
t=4
$2^{4\lceil log_{16} 10 \rceil} - 10 = 6, [6]$
t=5
$2^{4\lceil log_{16} 10 \rceil} - 10 = 22, [22]$
Clearly at this point since $2^4 > 10$, the memoization will just be $2^t - 10$

On the topic of this Example 14, it is not very clear which solution actually provides the greatest efficiency gains. It is somewhat more apparent which solutions fail to do this. Any t > 4 clearly is not helpful.

t = 1 has the lowest 2 norm
t = 2 has the fewest elements with the most minimum values
t = 3 has higher values than t = 2, but has a larger base

t = 4 possesses the minimum number of elements with the minimum base

The following are points of minimization that might effect the overall complexity.

1. The 2 norm is an important metric when ranking the memoizations as it suggests the distance of the memo elements are from 0 which suggests perfect memoization.

2. The length of the memoization is also an important metric when ranking the memoization as it reduces the number of iterations required which are quite costly.

3. Having a larger base could be beneficial since it means the memoization could be shorter since the starting point is higher, however this is not a steadfast rule

Since the entire scheme requires $\lceil \frac{lnb'}{lnb} \rceil (\frac{n^2}{2} + n^1(k+1+\frac{1}{2}) + k)$ n = $\lfloor \frac{|A|-1}{|C|-1} \rfloor + 1 = k+1$ operations, this function should be minimized.

$\lceil \frac{lnb'}{lnb} \rceil (\frac{(k+1)^2}{2} + (k+1)(k+1+\frac{1}{2}) + k)$

$= \lceil \frac{lnb'}{lnb} \rceil (1.5k^2 + 4.5k + 2)$

Clearly the largest factors to the number of operations is the exponent k, which is not controllable. $\lceil \frac{lnb'}{lnb} \rceil$,however is controllable. Since ln(b) is a monotonic increasing function, taking a larger value of b will result in less overall operations completed, but will not alter the running time complexity. This item works by altering the maximum length of the memoization which is helpful in reducing additional iterations. This raises t = 5, to potentially being a viable solution as well since it minimizes the overall number of memoization steps.

When comparing t = 4 and t = 5, there are two different concerns that might influence the efficiency of these operations. Using a lower base will require less bit shifts to occur since the kernel $[-1, 16]$ requires four bit shifts per convolution and m_i subtraction by 1 operations. The kernel $[-1, 32]$ requires five bit shifts and m_i subtractions per convolution. This means that t = 4, has $(1.5k^2+4.5k+2)$ less bit shift operations which might increase the efficiency. The number of subtraction operations is indeterminate in the general case as it depends on b', but in Example 14, t = 4 $6 - 16b$, t=5 $22 - 32b$, the difference between the two constants is 16, which generally has no performance difference. Although if this difference was on higher orders of magnitudes, reminiscent of a larger b', then minimum of these two would be preferential. This is is not the most important feature although it might be useful to note when exceeding native word sizes within an embedded system with a smaller frame or with very large exponents. For this sake, the memoization with the minimum number of items and minimum 2 norm.

For large b' it is important that manually iterating through every single memoization would add $\lceil log_2 b' \rceil + 1$, additional operations. Memoizations should only be checked within a neighborhood around b'. Clearly, no memoization

should be observed after t = $\lceil log_2 b' \rceil + 1$, potentially t = $\lceil log_2 b' \rceil$, might be the optimal solution.

Proof. $t = \lceil log_2 b' \rceil$

The memoization will be $2^{\lceil log_2 b' \rceil \lceil log_{2^{\lceil log_2 b' \rceil}} b' \rceil} - b'$

$2^{t\lceil log_{2^t} b' \rceil} - b'$

$= 2^{t\lceil \frac{log_2 b'}{log_2 2^t} \rceil} - b'$

$= 2^{L\lceil \frac{L}{Llog_2 2} \rceil} - b'$

$= 2^{t\lceil \frac{t}{t} \rceil} - b'$

$= 2^t - b'$

$2^{\lceil log_2 b' \rceil} - b'$

This means that the L memoization is equivalent to the t = 1 memoization.

$(2^t - b')_{2^t}$,

The corresponding $t = \lceil log_2 b' \rceil + 1$, would possess another bit in the binary representation than when $t = \lceil log_2 b' \rceil$. This extra bit will make the 2 norm of $t = \lceil log_2 b' \rceil$ to be smaller. Since the memo items within the memoization are the groupings of different bits, this additional bit could potentially add another grouping of t bits. This allows for $t = \lceil log_2 b' \rceil$ to possess the minimum memo items and minimum 2 norm. b' will always be less than 2^t, so due to this, the memoization will always be of length 1.

Adding a pre stage to call the general_base() function, will guarantee that the solver will execute in a singular iteration. This does alter the running time complexity of the algorithm to be in $O(k^2)$ as this is only a single stage since the memoization will always be of length 1.

The dynamic base geometric sequence solver is available at 'Section_8/dynamic_general_base.py' or within the command line program at 'Geometric Sequence Solvers/Dynamic General Base'.

9 Results Comparing The Execution Time of Each Method

This concludes the mathematical portion of the work to transition into measuring the actual performance of the proposed algorithms.These solvers will be compared against each other and by other geometric sequence generation algorithms based on their shared domain. The python programs are not hyper optimized and they represent almost exactly how the formulas have been shown to work. No python specific tricks or faster libraries have been used. The only outside libraries used are the scipy signal convolutions and the numpy Fast Fourier Transforms. The function that will be tested is of the kind $f(b, k) = t$, where b is the desired output base of the sequence, k is the highest exponent of the sequence, and t which is the output time in seconds. The sections regarding the practical running time of these algorithms will be broken down by base domains: $\mathbb{N}$,$\mathbb{Z}$,$\mathbb{Q}$ and exponent domains:$\mathbb{N}$,$\mathbb{Z}$.

The sections will include charts and tables detailing the execution over several inputs. The specific time values are not the main focus of these trials, it only serves as a general way to measure the relative efficacy of function against each other. Each trial outputs two charts and one statistical table. The first of the charts is four quartile divisions of the sampled domain in which the groupings have their timing averaged for each exponent. Each of the solver methods are overlaid onto the charts as a form of comparison. The next chart is the timing surfaces for the sampled inputs. This chart shows the full timing surface with no averaging, showing the effect of the base cross exponent input. The final chart of the series is a heat map that compares the difference between the numpy_geometric_sequence solver and each of the other bespoke solvers. This gives what is essentially a top down view of the surface charts. This heat map matrix allows for trends to be shown in the timing of different bases and exponents when compared against the baseline numpy solver.

Initial runs of the solver testers were conducted on a general purpose M1 MacBook Pro with 8GB of ram. The limited processing speed as to which this was running under was only enough to produce single iteration tests with some variation amongst tests. Transitioning the tests onto a more substantial RTX 4060 Ti with 32 GB of ram, allows for multiple iterations to be ran smoothing variations. Each curve, surface, and table is the result of 30 averaged trial. This gives more of a clear picture of the true nature to the shapes of these curves and surfaces. Unsurprisingly, these tests provided results with lower runtimes on the more advanced hardware which is able to more effectively demonstrate the uses for some of these methods.

This section will end off with some practical applications of geometric sequences, evaluating power series through function approximation, ordinary differential equation solving, and inelastic bouncing simulation. These all transform geometric sequences in some capacity which are generated using the rational_sequence solver.

The geometric sequence solver comparison function is available at

'Section_9/solver_comparison.py' or within the command line program at 'Analysis/Solver Comparison'.

9.1 Positive Integer Domain

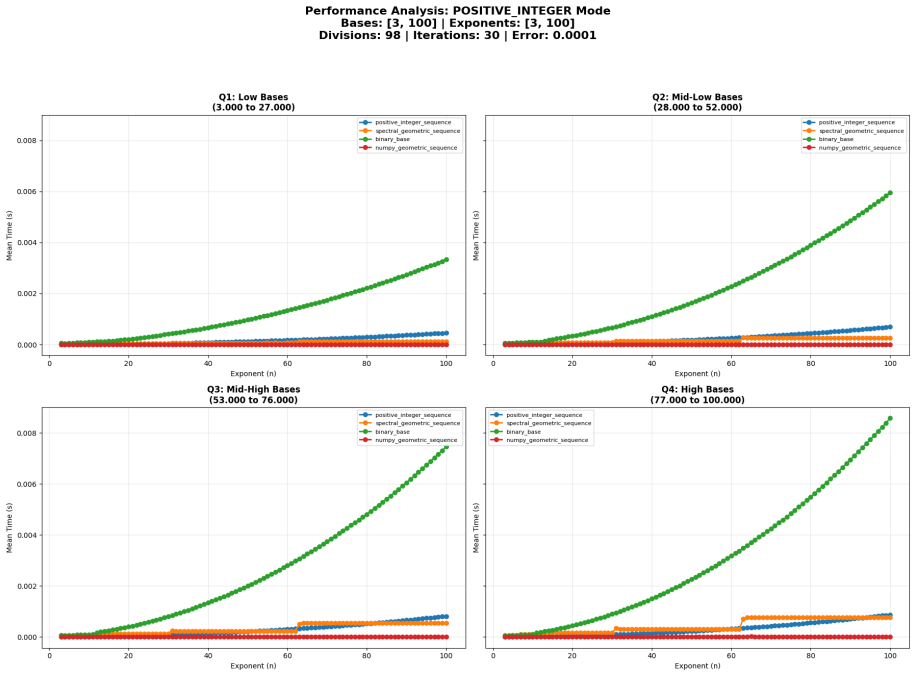
9.1.1 Positive Integer Base, Positive Integer Exponent: Function Inputs

The first class of functions are the positive integer domain solvers. These solvers require both a positive integer base and exponent. Both bases and exponents range from 3 to 100 which encompasses evaluating numbers from 3^3 to 100^{100}. This limit was arbitrarily set and is equivalent to a little more than 2^{664}, which is far larger than the maximum 64 bit integer and is below 128^{128}, which has crashed the Fast Fourier Transform. The functions that will be tested are Section 6 positive_integer_sequence, Section 6 spectral_geometric_sequence, and Section 8 binary_base which will be tested against the Section 9 numpy_geometric_sequence.

The numpy geometric sequence solver is available at 'Section_9/numpy_geometric_sequence.
or within the command line program at 'Geometric Sequence Solvers/Numpy Geometric Sequence'.

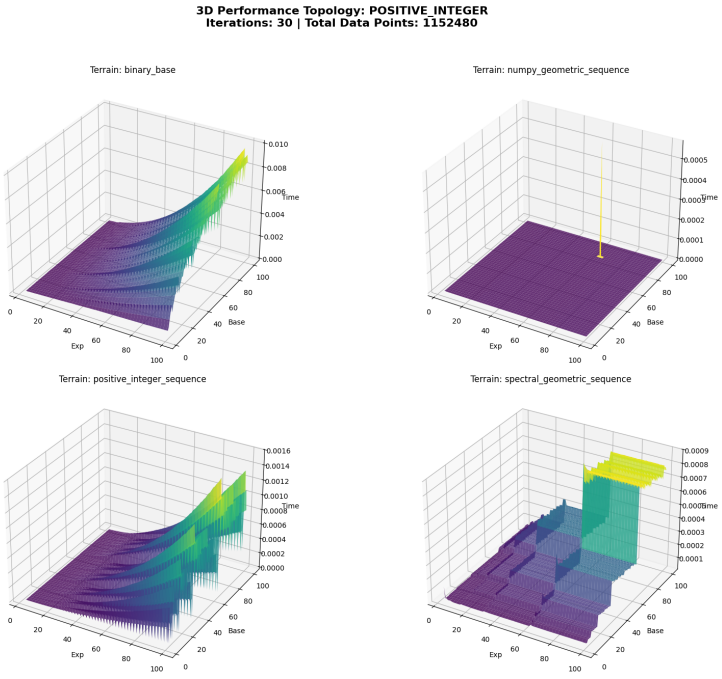
9.1.2 Positive Integer Base, Positive Integer Exponent: Quartile Average Charts

Figure 6: Positive Integer Solver Quartile Averages



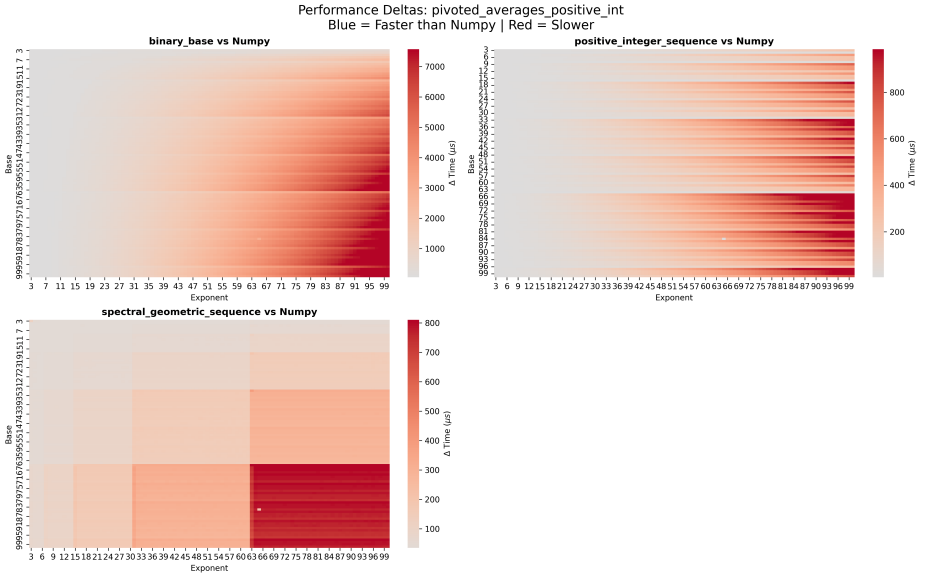
9.1.3 Positive Integer Base, Positive Integer Exponent: Surface Charts

Figure 7: Positive Integer Solver Surfaces



9.1.4 Positive Integer Base, Positive Integer Exponent: Heat Maps

Figure 8: Positive Integer Solver Heat Map



9.1.5 Positive Integer Base, Positive Integer Exponent: Function Statistics

Table 1: Positive Integer Solver Statistics

Function Name	Mean Time (μs)	Standard Deviation (μs)	Max Time (μs)	Median Time (μs)
numpy_geometric_sequence	1.34	32.12	17241.48	1.19
spectral_geometric_sequence	247.75	235.77	1132.49	159.26
positive_integer_sequence	249.44	261.34	1666.78	159.98
binary_base	2260.09	2107.59	10091.78	1593.11

9.2 Integer Base Positive Integer Exponent

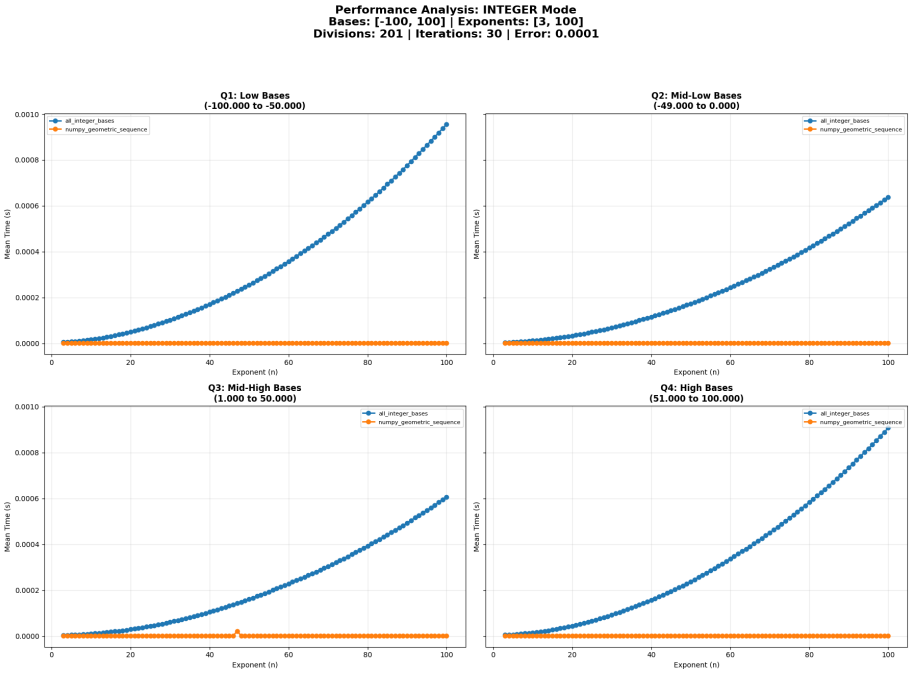
9.2.1 Integer Base, Positive Integer Exponent: Function Inputs

The class of functions that can only process an integer base and positive integer exponent only possesses a single function, Section 6 all_integer_bases. This method is essentially an extension of Section 6 positive_integer_sequence, but with negative integer aware memoization. Integer bases -100 to 100 were tested with exponent integers ranging from 3 to 100. It is important to note that the

memoization involved does not simply alternate the base's sign, but it is a unique combination with a different starting point from the positive counterpart.

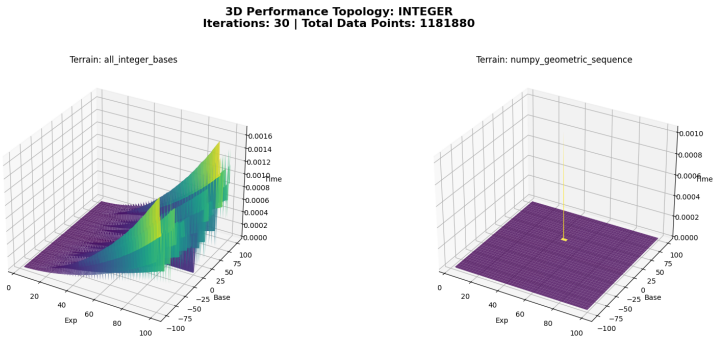
9.2.2 Integer Base, Positive Integer Exponent: Quartile Average Charts

Figure 9: Integer Solver Quartile Averages



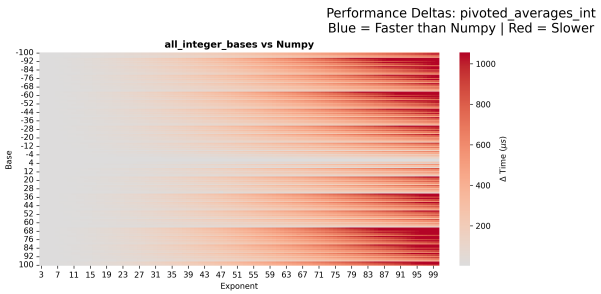
9.2.3 Integer Base, Positive Integer Exponent: Surface Chart

Figure 10: Integer Solver Surface Charts



9.2.4 Integer Base, Positive Integer Exponent: Heat Maps

Figure 11: Integer Solver Surface Heat Map



9.2.5 Integer Base, Positive Integer Exponent: Function Statistics

Table 2: Integer Solver Statistics

Function Name	Mean Time (μs)	Standard Deviation (μs)	Max Time (μs)	Median Time (μs)
numpy_geometric_sequence	1.38	40.35	31016.35	1.19
all_integer_bases	276.7	286.06	2102.14	178.58

9.3 Rational Domain, $0 \leq b' \leq 1$

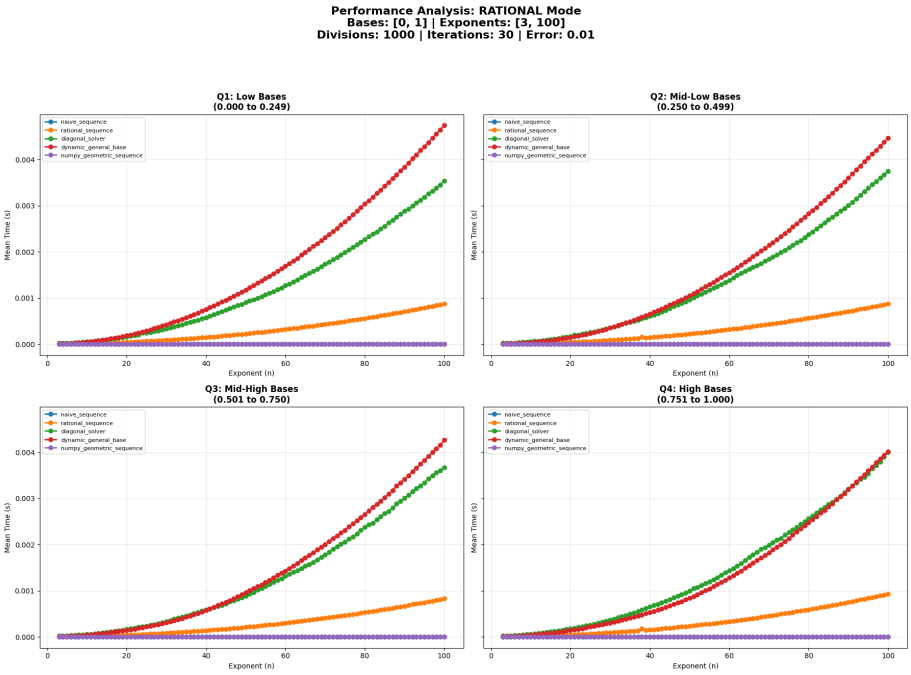
9.3.1 Rational Base, Positive Integer Exponent, Error 10^{-2} and 10^{-8}: Function Inputs

The rational bases has been spit into a singular set. This set contains only bases that fall between 0 and 1. The domain subsections are done in this way to be able to efficiently visualize the effects of each of the subsets. The domain is sampled into 1000 intervals of length .001 intervals from from 0 to 1. Having the .001 precision. The remaining rational solvers are able to handle any rational base input and any integer exponent. Only positive integers are demonstrated since the logic of negative integers within this rational domain number 1 is to the same effect as numbers within the next rational domain 2. Negative rational bases are also not demonstrated since the logic that allows that to work simply takes the positive version and alternates the signs which would not add any real time to the execution. Finally, rational bases that are larger than 1 are not used as well since in practice, these are simply scaled to be within $0 < b' < 1$.

The rational solvers rely on the rational approximation algorithms which require an error bound. Tests were conducted with a bound of .01 and 10^{-8}, to demonstrate a potential difference in execution time. This calculates about three million sequences per function and takes several hours to achieve these results.

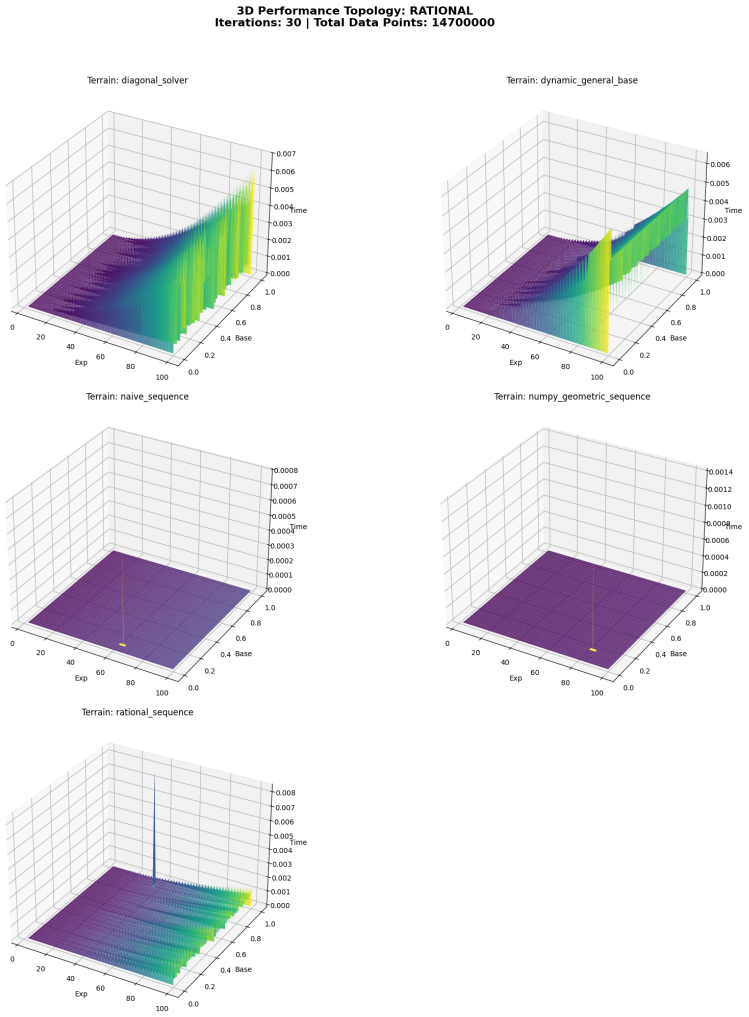
9.3.2 Rational Base, Positive Integer Exponent, Error 10^{-2}: Quartile Average Charts

Figure 12: Rational Solver Quartile Averages Error 10^{-2}



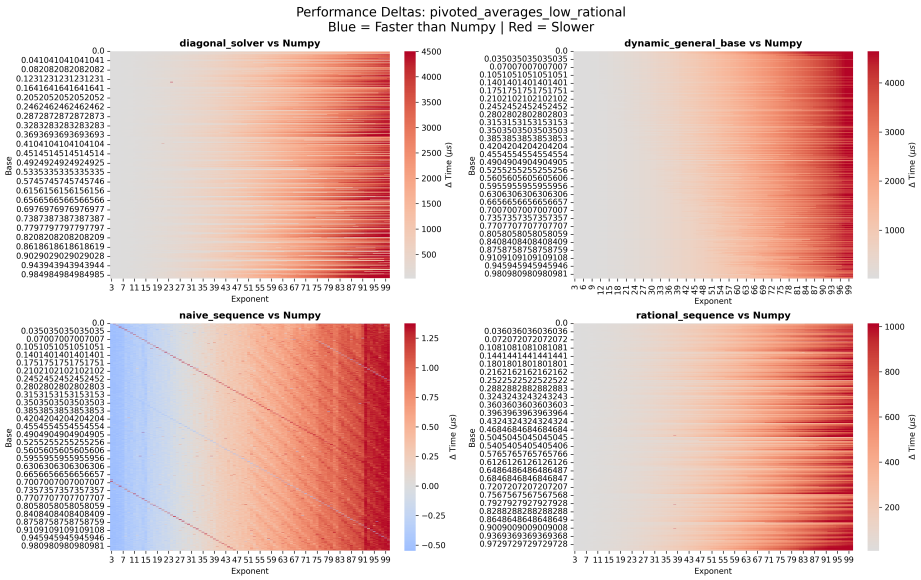
9.3.3 Rational Base, Positive Integer Exponent, Error 10^{-2} Surface Charts

Figure 13: Rational Solver Surfaces Error 10^{-2}



9.3.4 Rational Base, Positive Integer Exponent, Error 10^{-2} Heat Maps

Figure 14: Rational Solver Heat Maps Error 10^{-2}



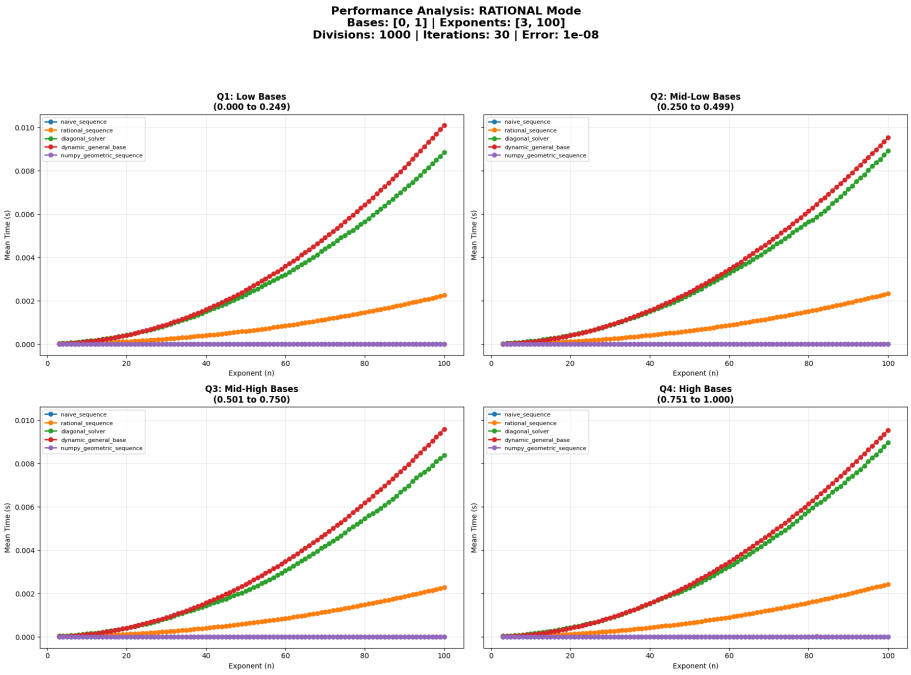
9.3.5 Rational Base, Positive Integer Exponent, Error 10^{-2} Function Statistics

Table 3: Rational Solver Statistics Error 10^{-2}

Function Name	Mean Time (μs)	Standard Deviation (μs)	Max Time (μs)	Median Time (μs)
numpy_geometric_sequence	1.43	24.41	41838.65	1.43
naive_sequence	1.83	13.83	23651.12	1.91
rational_sequence	308.85	336.75	245622.63	224.59
diagonal_solver	1300.05	1282.61	138550.04	835.9
dynamic_general_base	1465.37	1369.2	11747.6	1030.45

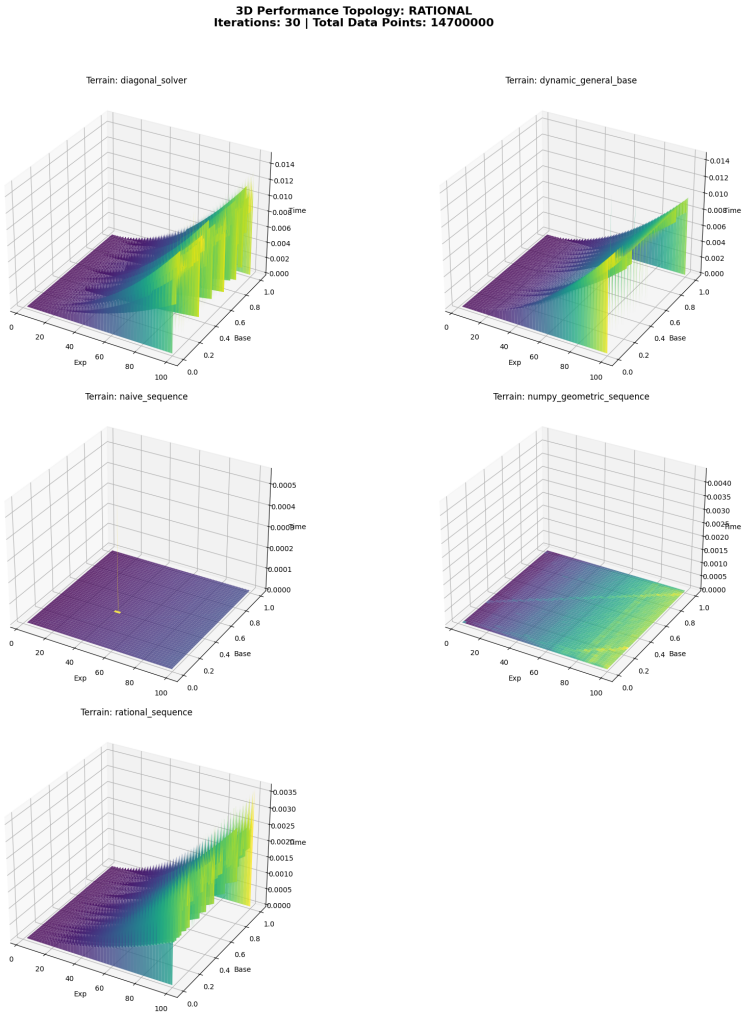
9.3.6 Rational Base, Positive Integer Exponent, Error 10^{-8} Quartile Average Charts

Figure 15: Rational Solver Quartile Averages Error 10^{-8}



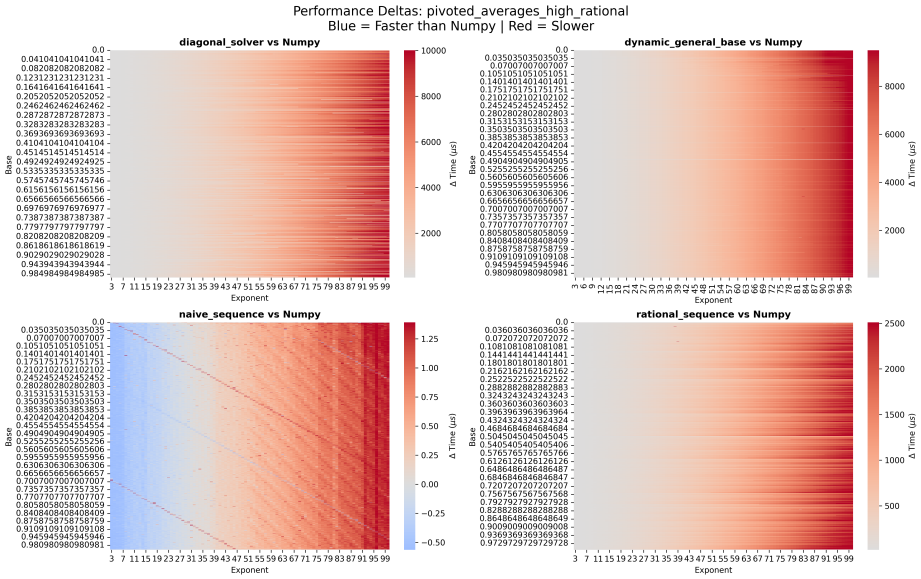
9.3.7 Rational Base, Positive Integer Exponent, Error 10^{-8} Surface Charts

Figure 16: Rational Solver Surfaces Error 10^{-8}



9.3.8 Rational Base, Positive Integer Exponent, Error 10^{-8} Heat Maps

Figure 17: Rational Solver Heat Map Error 10^{-8}



9.3.9 Rational Base, Positive Integer Exponent, Error 10^{-8} Function Statistics

Table 4: Rational Solver Statistics Error: 10^{-8}

Function Name	Mean Time (μs)	Standard Deviation (μs)	Max Time (μs)	Median Time (μs)
numpy_geometric_sequence	1.48	80.22	131955.62	1.43
naive_sequence	1.81	9.81	16735.79	1.67
rational_sequence	835.64	725.92	93438.39	638.96
diagonal_solver	3077.79	2898.58	16970.63	2064.23
dynamic_general_base	3365.81	2945.5	242035.15	2562.05

9.4 Applications of Geometric Sequences

All of the following applications make use of an x^n term with an increasing n. Any sort of formulation that uses this form can make use of geometric sequences where the maximum value of n is calculated which allows for a sequence to be pre calculated. These applications are interesting demonstrations of the full range of uses of geometric functions. They also demonstrate that there is a direct

path from simplistic midpoint and anti midpoint operations to more complex or useful solutions.

9.4.1 Taylor Series Function Evaluation

The Taylor Series is defined as the following, where a represents a point in which the sequence is centered around, f is a one dimensional, infinity differentiable function at 'a' with an input of x.

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \tag{33}$$

Clearly the $(x-a)^n$ represents the elements of a geometric sequence. An error bound E can be introduced to limit the number of terms calculated within the sum. A simple bound can be instated where $|t_n = \frac{f^{(n)}(a)}{n!}| < E$ denotes that the bound has been reached. This could be pre calculated, however this would be unnecessary since this $t_n$value is used to scale the geometric sequence. In practice, a prefixed large number of elements (200 or 500) are generated to compensate for this. When trying to evaluate for x, it is ideal to use a value of a that is very close to x so the approximation can be as accurate as possible, while obliging by the domain of f.

By engineering certain functions, desired approximation results can be achieved.

- $f(x) = e^x, f(1) = e$. Using an error bound E of 10^{-16} allows for an approximation of e to be 2.718281828459045 which has all of its digits correct for all of its digits and exactly matches numpy.e.
- $f(x) = 4arctan, f(1) = \pi$. Using the same error bound $E = 10^{-16}$, this yields an approximation of π at 3.1415926535897936 (17 digits) which is accurate until the last digit which should be a 2. This approximation error occurs even when increasing the bounds and is likely due to the limits of the rational approximation method in 64 bits. It is however exactly precise to numpy.pi when the last digit is removed.
- $f(x) = \sqrt{x}, f(2) = \sqrt{2}$. Using $E = 10^{-16}$,$\sqrt{2} \approx 1.4142135623730951$ (17 digits), which is again accurate to the 64 bit limit where the final digit should be 0. This however is exactly equivalent to numpy.sqrt(2).
- This can also be used in a manner that is quite overkill. $f(x) = 2^x, f(20) = 2^{20}$. This is completely valid as a series although the implementation is unnecessary since a geometric sequence can be used to solve this exactly. It results in 1048575.9999999999 which does round to 1048576 which is correct.

The Taylor Series solver is available at 'Section_9/power_series_approximation.py' or within the command line program at 'Applications/Power Series Approximation'.

9.4.2 First Order Ordinary Differentiation Equation Solver

The Taylor Series can also be used to solve Ordinary Differential Equations initial value problems. The formula is as follows where $y^{(k)}$ is a differential equation of t and y This iterative method solves for y at different intervals i where h is the step size, n controls the precision, by using the initial value or previous calculated value.

$$y_{i+1} = y_i + \sum_{k=1}^{n} \frac{h^k}{k!} y^{(k)}(t_i, y_i) \tag{34}$$

This is equivalent to The Newton Method for initial value problems when n = 1, which will only produce the first order linear term. The practical application is only able to solve these equations in the first order, but it is possible to augment it to allow for systems of first order equations that are equivalent to higher orders.

Example 15. Solve the initial value problem $y' = y, y(0) = 1$

Using the Laplace Transform.

$\mathscr{L}(y') = \mathscr{L}(y)$

$s\mathscr{L}(y) - f(0) = \mathscr{L}(y)$

$s\mathscr{L}(y) + \mathscr{L}(y) = 1$

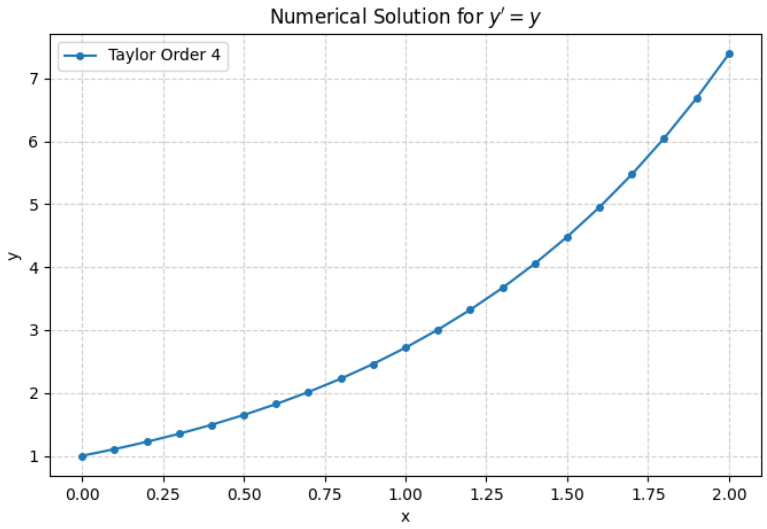
$\mathscr{L}(y)(s+1) = 1$

$\mathscr{L}^{-1}(\mathscr{L}(y)) = \mathscr{L}^{-1}(\frac{1}{s+1})$

$y = e^t$

The solver outputs the following function, which approximates the analytical solution $y(t) = e^t$.

Figure 18: Initial Value Problem Solver $y' = y, y(0) = 1, y(t) = e^t$



Another main initial value problem solver method is known as Runge Kutta. This iterative method uses the weighted averages of points at various slopes to find the resultant curve. The following table shows the comparison of the RMSE that is calculated between various solver methods at different error resolutions. The methods that are tested are the Taylor solver using the geometric sequences, the Runge Kutta method, and the actual analytical solution to the initial value problem which the iterative methods are tested against. These methods are all tested at the same fourth order where applicable upon the same 0 to 2 interval with a step size of $\frac{1}{10}$.

Table 5: Initial Value Problem Root Mean Squared Errors

ODE	Method	Control Method	Approximation Error	RMSE
$y' = y$	Taylor	Runge Kutta	10^{-4}	$1.60 * 10^{-4}$
$y' = y$	Taylor	$y(t) = e^t$	10^{-4}	$9.86 * 10^{-5}$
$y' = y$	Runge Kutta	$y(t) = e^t$	10^{-4}	$6.47 * 10^{-5}$
$y' = y$	Taylor	$y(t) = e^t$	10^{-16}	$4.72 * 10^{-6}$
$y' = y$	Taylor	Runge Kutta	10^{-16}	$4.72 * 10^{-6}$
$y' = y$	Runge Kutta	$y(t) = e^t$	10^{-16}	$3.55 * 10^{-14}$

This next example will solve another initial value problem $y' = y+t, y(0) = 1$, upon the same interval. There will also be the same comparison upon the root mean squared error resultant from the iterative methods.

Example 16. Solve the IVP $y' = y + t, y(0) = 1$

Using the Laplace Transform.

$\mathscr{L}(y') = \mathscr{L}(y + t)$

$s\mathscr{L}(y) - f(0) = \mathscr{L}(y) + \mathscr{L}(t)$

$\mathscr{L}(y)(s-1) = \mathscr{L}(t) + 1$

$\mathscr{L}(y) = \frac{1}{(s-1)}(\frac{1}{s^2} + 1)$

$\mathscr{L}(y) = \frac{1}{(s-1)s^2} + \frac{1}{(s-1)}$

$\frac{1}{(s-1)s^2} = \frac{A}{s-1} + \frac{B}{s} + \frac{C}{s^2} = \frac{As^2+Bs(s-1)+C(s-1)}{(s-1)s^2}$

$1 = As^2 + Bs(s-1) + C(s-1)$

$1 = A$ (s=1)

$-1 = C$ (s=0)

$0 = s^2 - s + Bs(s-1)$

$0 = s(s-1)(1+B)$

$-1 = B$

$\mathscr{L}(y) = \frac{1}{s-1} - \frac{1}{s} - \frac{1}{s^2} + \frac{1}{(s-1)}$

$\mathscr{L}(y) = \frac{2}{s-1} - \frac{1}{s} - \frac{1}{s^2}$

$y = 2e^t - 1 - t$

The output figure is the following, approximating $y(t) = 2e^t - t - 1$

Figure 19: Initial Value Problem Solver $y = y + t, y(0) = 1, y(t) = 2e^t - t - 1$

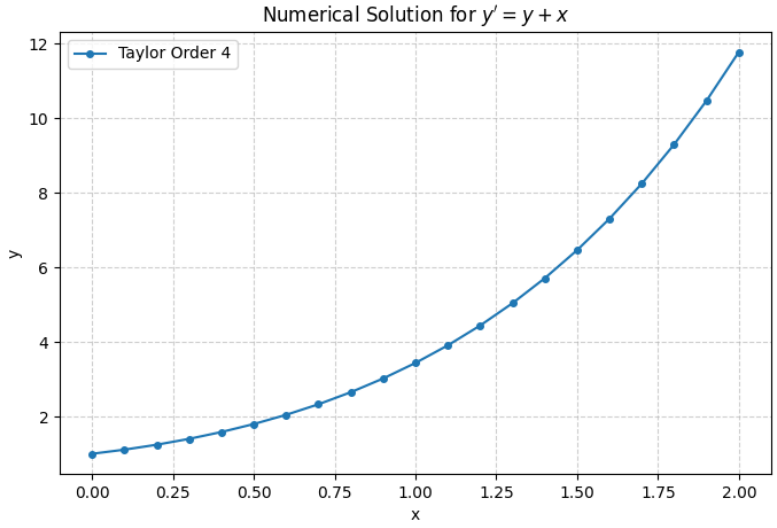


Table 6: Initial Value Problem Root Mean Squared Errors

ODE	Method	Control Method	Approximation Error	RMSE
$y = y + t$	Taylor	$y(t) = 2e^t - t - 1$	10^{-4}	$1.55 * 10^{-4}$
$y = y + t$	Taylor	Runge Kutta	10^{-4}	$2.49 * 10^{-4}$
$y = y + t$	Runge Kutta	$y(t) = 2e^t - t - 1$	10^{-4}	$9.92 * 10^{-5}$
$y = y + t$	Taylor	$y(t) = 2e^t - t - 1$	10^{-16}	$9.43 * 10^{-6}$
$y = y + t$	Taylor	Runge Kutta	10^{-16}	$9.43 * 10^{-6}$
$y = y + t$	Runge Kutta	$y(t) = 2e^t - t - 1$	10^{-16}	$4.62 * 10^{-14}$

Both of these initial value problems were solved using The Taylor method in the fourth order using geometric sequences that were completely constructed through applications of the successive midpoint. The geometric sequence that was used was able to generate the h^k within the sum of Formula 34. This sequence represents the step size exponentiated by the current order of the IVP approximation. For these simple initial value examples, it is more practical to simply perform the initial work to solve the equation so an exact solution can be found.

When comparing these solvers against each other, clearly at the same error bound the Runge Kutta method is able to provide a more accurate solution to the initial value problem than the geometric sequence Taylor method. Decreasing the acceptable approximation error does help the geometric sequence be more accurate which leads to a lower RMSE. The jump from an error of 10^{-4} to 10^{-16} only provides two orders of magnitude of improvement upon the RMSE for the Taylor method. For the same reduction in error, the Runge Kutta method receives an RMSE decrease of 9 orders of magnitude.

As a simple test, the rational_sequence solver, powering the ode_ivp_solver was changed to use the naive_sequence solver which produces the exact geometric sequence. The RMSE returned from the swapping of the geometric sequence solver within the Taylor method was completely unchanged. This suggests that the limit of this Taylor solver does not lie within the geometric sequence approximation.

The initial value problem solver is available at 'Section_9/ode_ivp_solver.py' or within the command line program at 'Applications/ODE IVP Solver'.

The initial value problem comparison function is available at 'Section_9/runge_kutta_comparison.py' or within the command line program at 'Analysis/Runge Kutta Comparison'.

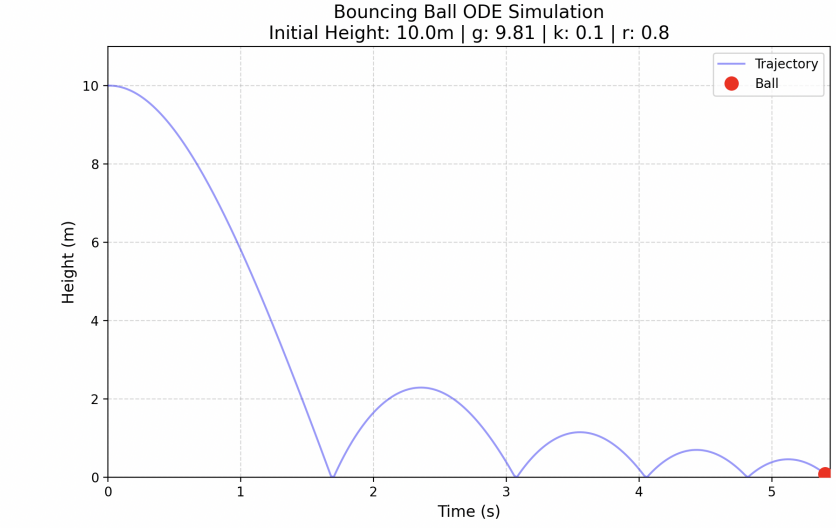
9.4.3 Physical Simulation of Energy Lost Within An Inelastic Collision

There are plenty of physical processes that are described by the decay and growth of energy. One such process is the decay of energy from a bouncing as it collides with the floor.

It is possible to create a physical simulation of a ball bouncing purely with algebraic formulas purely using geometric sequence since. This can be done as the velocity of the ball is reduced by a factor of the restitution coefficient when impacting the floor while the max height is reduced by a factor of the restitution squared. Each element in the sequence would represent this decay of energy leaving the system upon each bounce. A geometric sequence where the base is the restitution could be created to be able to effectually model the height of the ball over the count of bounces.

Using these algebraic methods will result in the ball's motion being accurately modeled, but a more advanced solution can be created using the previously created ordinary differential equation solver. The ball's height over time is able to be modeled as a second order differential equation $y'' = -g$ where g represents the acceleration applied onto the falling object, in this case gravitational. This second order differential equation $y'' = -g$ can be turned into $v' = -g$, $y' = v$, representing the velocity and change in velocity. An additional term can also be added to the acceleration to act as Newtonian air resistance imparted as acceleration in the inverse direction of the velocity. This quadratic drag relationship can be modeled as $v' = -g - k * v * |v|$, $y' = v$ where k is the air resistance within the system. When there is no air resistance present within the system, k = 0, which reduces the system of equations to $y'' = -g, y' = v$. The solutions from this system is able to be used to simulate the height of a ball as it bounces.

Figure 20: Ball Bouncing Simulation



This demonstrates that what is a complex physical process, can be represented as what is essentially a series of bit shifts and subtraction by one operations.

The bouncing ball simulation is available at 'Section_9/simulate_bouncing_ball.py' or within the command line program at 'Applications/Bouncing Ball Simulation'.

10 Conclusion

10.1 Discussion

10.1.1 General Discussion of Methods Used

The main thesis of this work has been proven with some minor caveats. It was proven that it is possible to exactly represent geometric sequences of any rational base of the form $\{\frac{a}{2^n} : a, n \in \mathbb{Z}\}$, purely by taking midpoints and anti midpoints upon an initial zeros or ones array. This does not extend to include any rational or any real number base, but can utilize techniques to achieve an acceptable level of approximation. It is however possible to represent any rational number $\{\frac{a}{b^n} : a, b, n \in \mathbb{Z}\}$without approximation by using a more general form with altered midpoint $[\frac{1}{b}, \frac{1}{b}]$, and anti midpoint $[-1, b]$ weights. Additionally, these generated geometric sequences were able to be used within various applications including function approximation and in the solving of initial value problems.

10.1.2 Discussion of Timing Results

Even though it has been shown that it is in fact possible to generate a geometric sequence through this construction, the implementations created are unable to perform as well as usual techniques. These methods have been deemed to be inferior in both theoretical time complexity and in execution time when compared against a brute force geometric sequence method. The algorithms outlined generally operate on time scales within a few orders of magnitude greater than the standard methods. The following lists the algorithms' execution times ascending order of mean execution time. The precision of the actual timings is up for debate as well since it could be affected by background processes and unoptimized code. The function performance is shown to the microsecond scale which might not be entirely accurate so this analysis is meant to look at the general performance. The 30 rounds of iterations do help to visualize trends in the data.

Table 7: Solver Statistics

Function Name	Base Domain	Error Bound	Mean Time (μs)	Standard Deviation (μs)	Median Time(μs)
numpy_geometric_sequence	$\mathbb{N}$	0	1.34	32.12	1.19
numpy_geometric_sequence	$\mathbb{Z}$	0	1.38	40.35	1.19
numpy_geometric_sequence	$\mathbb{Q}$	10^{-2}	1.43	24.41	1.43
numpy_geometric_sequence	$\mathbb{Q}$	10^{-8}	1.48	80.22	1.43
naive_sequence	$\mathbb{Q}$	10^{-8}	1.81	9.81	1.67
naive_sequence	$\mathbb{Q}$	10^{-2}	1.83	13.83	1.91
spectral_geometric_sequence	$\mathbb{N}$	0	247.75	235.77	159.26
positive_integer_sequence	$\mathbb{N}$	0	249.44	261.34	159.98
all_integer_bases	$\mathbb{Z}$	0	276.70	286.06	178.58
rational_sequence	$\mathbb{Q}$	10^{-2}	308.85	336.75	224.59
rational_sequence	$\mathbb{Q}$	10^{-8}	835.64	725.92	638.96
diagonal_solver	$\mathbb{Q}$	10^{-2}	1300.05	1282.61	835.90
dynamic_general_base	$\mathbb{Q}$	10^{-2}	1465.37	1369.20	1030.45
binary_base	$\mathbb{N}$	0	2260.09	2107.59	1593.11
diagonal_solver	$\mathbb{Q}$	10^{-8}	3077.79	2898.58	2064.23
dynamic_general_base	$\mathbb{Q}$	10^{-8}	3365.91	2945.50	2562.05

From this data, it appears that there are three orders of function. The following enumeration will begin to describe general trends within the data.

1. The highest performing methods, within the range of $1\mu s$ on average are solvers that are built upon successively multiplying upon previous elements. These methods (numpy_geometric_sequence and naive_sequence) performed the best for all input types, with the numpy based solver performed the best. The difference between the error used also mostly distinguished in what run they were a part of, since these functions bypass any type of error approximation methods, with the average times between the two error variants to being negligible. It can also be noted that these solvers do perform better on more simple inputs and take longer on the rational inputs although the difference is within tenths of micro seconds (1.34 - 1.83 μs).

2. The median performing class of functions operate on average within two orders of magnitude larger than the previous class of functions within hundreds of micro seconds. These solvers also have an internal ordering amongst themselves where natural inputs are the fastest ($\approx 250\mu s$), followed by the integer input ($\approx 280\mu s$),then ending with the rational_sequence solvers at ($\approx 310 - 840\mu s$) depending on the error level which is almost a three times increase in time.

3. Finally functions that are on the order of single milliseconds or thousands of microseconds. These function are mostly rational solvers with binary_base as the only natural solver. These solvers were intended to be optimizations upon the original midpoint base 2 solvers. Although the optimizations for all of these solvers were done in an effort to reduce the memoization length while using larger bases other than 2. Clearly these failed to run in a more effective manner when compared to the rational_sequence solver. The solutions are also clearly affected by the level of error used in the rational approximations. The difference on average of the calculation time is more than double when jumping from and error of 10^{-2} to 10^{-8}, which is true of the previous rational_sequence solver's response to error as well.x

Within Figure 10.1.2 Positive Integer Base, Positive Integer Exponent: Surface Charts, it seems that the spectral_geometric_sequence has a timing function similar to some sort of step function. It appears that these steps take place, not on random intervals, but in fact powers of two over both the base and exponent inputs. This is interesting as it suggests that there is quite a noticeable difference between the times within these power of two steps.The nearly flat surface topology upon these ridges supposes that there is almost no difference in the underlying frequency domain implementation of this algorithm when within a $\lceil log_2 b' \rceil x \lceil log_2 k \rceil$region. This is likely due to how the exponentiation of the kernel is handled on the power of 2 input.

When observing the difference in times between the rational_sequence solver and all of the general base class 3 solvers, clearly the base 2 approach seems superior in execution time. For the general approaches, too much time must have been used on either the memoization calculation stages or the actual applications of the kernels. This clearly shows that the length of the memoization might not be the only determining factor when optimizing a solver which opposes the time complexity argument proposed in Section 8.5.1. Finding a solution in terms powers of two might actually be more effective than simply producing a short memoization in a larger base. The underlying hardware does work in a language of base two which was determined to have some statistically significant effect over the execution time of the solvers.

Purely by observing Figure 10.2.2 Integer Base, Positive Integer Exponent: Quartile Average Charts, it is clear that the outer quartiles and inner quartiles perform in a similar capacity. The outer quartiles (-100 to -50 and 50 to 100) both appear to follow the same curve when overlaid. They both end at around .001 seconds while the inner quartiles (-49 to 49) both end at around 0.0006 seconds when averaging time execution time of all of these bases. This is likely due to the fact that the memoizations of a positive and negative base are roughly the same length.

It appears that for small integer exponents centered around 0, the performance time is similar to that of the product methods irregardless of base as shown on the heat maps. The difference heat maps generally show less of a difference when compared against the control numpy_geometric_sequence solver

with lower exponents than higher exponents. Within the novel algorithms, the larger exponents do add significant expense to the execution as each exponent adds an entire new column to be executed upon, updating the anti diagonal of the cascade matrix. This of course needs to be done for each memoization step which for the sampled bases 3 - 100 is only at most 7 steps ($\lceil log_2(100) \rceil$). This highlights a fundamental limitation of the midpoint method when scaling up to larger exponents. With each additional element in the sequence added, much more work must be done when compared against increasing the base.

The error approximation does make a different in the execution time. For the shift between 10^{-2} and 10^{-8}, the difference in timings remains within the same timescale. Due to the large amount of time it takes to generate these charts, the final error bound of $10^{-16}$was not tested, but it would assumedly take a much greater time to execute since the memoizations required to approximate on that scale are much larger.

Within Figure 10.3.3 Rational Base, Positive Integer Exponent, Error 10^{-2} Surface Charts, in the naive_sequence surface there appears to be an exponent and base combination that consistently takes far longer than its neighbors. This spike corresponds to the geometric sequence of $\frac{26}{333}$with a maximum exponent of 63. This generates the 64 elements in $23651.12\mu s$ a single time when the other iterations average about 3.2 μs. This random spike must have skewed the mean of those $\frac{26}{333}$, 63 runs quite considerably to be visible. Within the same figure, rational_sequence experiences a spike at $\frac{100}{111}$ at an exponent of 38 for a time of $245622.63\mu s$. This is also the same for Figure 10.3.6 Rational Base, Positive Integer Exponent, Error 10^{-8} Surface Charts in the same naive_sequence for base $\frac{377}{999}$ at exponent 43 taking 16735.79 μs. In that same image the numpy_geometric_sequence solver experiences a spike at $\frac{274}{333}$with an exponent of 82 for $131955.62\mu s$. These spikes are not statistically significant, falling way above multiple standard deviations from the mean, however it is necessary to explain these maximums representing extreme outliers.

Overall, the rational_sequence solver is the most versatile of the rational solvers. Due to the implantation of this algorithm, rational_sequence reduces into the all_integer_bases solver when dealing with integer bases and the positive_integer_sequence when a positive base is input. It is likely that if these inputs were actually tested within rational_sequence, that the execution times would perform in a similar manner to these functions. Potentially a further domain expansion can be proposed onto this rational solver to make it even more robust.

10.2 Future work

There is much potential for remaining work that can be done on this topic. The goals of these suggestions are either ways to make the current solutions more robust or simply more efficient. Additional work could even be done to reduce the impact that the highest exponent holds on the complexity.

10.2.1 Shifted Exponents

One of the unsolved cases of this geometric sequence generation problem involves exponents not starting at 0. For example $[2^{-1}, 2^0, 2^1]$is a valid geometric sequence however within the current implementation it is not entirely proven that this will yield $[3^{-1}, 3^0, 3^1]$ with the [-1,2] kernel as expected. By running some simple tests it does appear that this does not work as expected so potentially another kernel would have to be used to account for this shift. It is also possible to approach this problem as two subsets by first calculating the positive exponent sequence then combine it together with the calculated negative sequence. This is also equivalent to simply scaling the entire sequence to match the desired exponents.

Another potential example that is not currently possible within the methods is the case of generating a geometric sequence similar to $[b^{10}, b^{11}, b^{12}]$. Clearly this is equivalent to a scaled sequence $b^{10}[b^0, b^1, b^2]$, but this is not yet possible within any of the methods. Potentially there might be some kernel to transform the input $[2^{10}, 2^{11}, 2^{12}]$ without the need for scaling.

General scaling by a factor of a is also not yet possible within the methods, although scaling the input has been shown to be equivalent to scaling the output by the same scalar. Adding this within the methods would be able to be done simply without any real additional time cost.

As mentioned before, the only supported exponents for the geometric sequences are monotonic integer sequences. However a geometric sequence may be defined with for any incremental exponents, not just this specific integer subset. Potentially, there might be some kernel that is able to transform an input of integer powers into that of rational powers. This might likely be exactly possible in the infinite case, but would likely require approximation within the discrete case. Additionally, it is possible to treat these rational exponents again as a scalar shift over the entire sequence.

Finally, sequence of decreasing exponents starting from the positive side and increasing exponents starting from the negative side also need to be implemented. This can be done by employing the previously discussed shifting operations or by reversing the order of a calculated sequence. Potentially there might even be a kernel that can perform these transformations as well.

All of these shifts can easily be implemented within the existing methods to handle the additional scale factors. It could be possible to use the Taylor Series functional approximate to generate the scale factors to shift these sequences. However it is more interesting to find a kernel that can properly carry out these transformations that simply using an algebraic transformation. There might even be a unique kernel that is able to create a memoization of negative rational bases without the need to treat them as being positive.

10.2.2 Expanded Domain For Spectral Solver

The best performing custom made geometric sequence solver is the spectral_geometric_sequence which transforms the input into the frequency domain to efficiently multiple the

kernel. This produced the time complexity of the lowest order and the lowest execution time on average ($247.75\mu s$). This however only operated upon the base domain of positive integers which is a severe restriction when compared to the other rational solvers. Potentially a future iteration of this solver can be made to accepting rational base and exponent inputs, making use of the efficiencies afforded by the Fast Fourier Transform. It could even be expanded to utilize the dynamic base algorithm to be able to more effectively work with rational numbers by using a more precise approximation even if the spatial dynamic_general_base solver performed the worse. This would require the Fourier Transform to be proven in the general case as currently it was only shown to work for base 2.

10.2.3 Optimizations

In a running system like the solver_comparison, it might be possible to cache the results of sequences to be able to be used later. The cached results could be shifted upon to make transformed sequences that are fit to purpose. This would help in operations that reuse the same slight variations of sequences constantly. This would require memory management that would have to outlive the solver function which introduces other challenges, but could significantly reduce times.

There are python specific optimizations that can be made to the solvers that would either take advantage of features of the runtime or are just simply more quick operations. For example, using numpy arrays instead of python lists is generally advisable as they are of fixed length and use C as their implementation. For the base 2 solvers, explicitly using bit shift operations might help in reducing the execution time as well. There are additional methods like numpy dot product and scipy convolution that allow for the optimized implementations of these common operations. Simply moving the solvers to use a compiled or generally faster language could also greatly help the performance.

An aggregate solver could also be created, combining the strengths of each of the solvers. This would use the data retrieved from the experimentation to use the best possible solver when dealing with specific combinations of sequence base and exponents. The solver could follow the heat map data to ensure that the most optimal solver for the input conditions is used so the lowest execution time can be met.

10.3 Final Words

This body of work answers the question in regards to the midpoint methods of 'is it possible?', rather than 'is it practical?'. It has clearly been shown that in the current implementation that simply successively multiplying numbers together is more efficient. Applying successive midpoints and anti midpoints to a sequence has been shown to be able to exactly yield any rational base with a power of two denominator. In the general case, any rational number base sequence can be expressed through successive convolution with some kernel. Potentially with further work and optimizations, this method could decrease in

execution time becoming more viable. The decomposition provided throughout this work, is able to be a guiding method for someone interested in calculating geometric sequences by hand or only with simplistic tools. Thank you for reading.

References

[1] Donald E. Knuth. *The art of computer programming volumes 1-4A-1.* Addison-Wesley, 1997.

[2] Alan V Oppenheim, Ronald W Schafer, and John R Buck. *Discrete-Time Signal Processing*, volume 2. Prentice-Hall, Upper Saddle River, NJ, 2 edition, 1999.

[3] Ronald L Graham, Donald E Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley Professional, 02 1994.

[4] Jiang Zeng. Multinomial convolution polynomials. *Discrete Mathematics*, 160:219–228, 1996.