Representing Geometric Series As Efficient Linear Combinations of Base 2

Stefan Gerber

December 1, 2025

Motivations

To be able to generate an exponent b^n by only performing simple additions and bit shifts.

This involves either taking successive midpoints (‘$(a + b)/2 = c$‘) or successive anti-midpoints (‘$2c - b = a$‘) upon a predetermined exponential basis, ‘$[b^0, b^1, \ldots, b^{n-1}]$‘.

To be able to find which series of bases are able to successfully generate the desired exponent, an inverse dyadic mapping can be applied onto the desired base to form an address that allows for its reconstruction.

This is able to simply find exponents of any integer b.

Proof 1: Successive Midpoints - Proof by Induction

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

Base Case

Let there be a base B of length 2: $B = [b_0, b_1]$. The successive midpoint of this basis 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

Now there is a basis B of length $k+1$. Since the basis 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 $m_0 = \frac{1}{2^{k-1}} \sum_{i=0}^{k-1} \binom{k-1}{i} b_i$. The successive midpoint of B_1 is $m_1 = \frac{1}{2^{k-1}} \sum_{i=0}^{k-1} \binom{k-1}{i} b_{i+1}$.

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 $\frac{1}{2^k} \sum_{i=0}^{k} b_i \binom{k}{i}$.

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

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

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

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

Therefore

$$\begin{aligned}
&= \frac{1}{2^k} \left(b_0 + \sum_{i=1}^{k-1} b_i \binom{k}{i} + b_k \right) \\
&= \frac{1}{2^k} \sum_{i=0}^{k} b_i \binom{k}{i} \quad \text{QED.}
\end{aligned}$$

Successive Midpoints: Examples and Visualizations

Now that the successive mapping of midpoints has been formulated, analysis and conclusions can be drawn from the findings.

If there is some finite basis B, it is possible to compute successive midpoints upon the basis and resultant bases to yield the ultimate midpoint. This however is quite inefficient for performing calculation.

For a basis B of n elements, each round of performing a successive midpoint can be summarized by the following pseudocode.

Algorithm 1 Successive Midpoint

```
1: function SUCCESSIVE_MIDPOINT($B$ :: basis)
2:     $n \leftarrow |B|$
3:     if $n < 2$ then return $B$
4:     end if
5:     for $i \leftarrow 0 \dots n-2$ do
6:         $B[i] \leftarrow (B[i] + B[i+1])/2$        ▷ Using integer division ‘» 1‘ for
   efficiency if values are integers
7:     end for
8:     $B.\text{pop\_back}()$ return $B$
9: end function
```

Algorithm 2 Recursive Ultimate Midpoint

```
1: function RECURSIVE_ULTIMATE_MIDPOINT($B$ :: basis)
2:     $n \leftarrow |B|$
3:     if $n < 2$ then return $B$
4:     end if
5:     while $n > 1$ do
6:         $B \leftarrow$ SUCCESSIVE_MIDPOINT($B$)
7:         $n \leftarrow n - 1$
8:     end whilereturn $B$
9: end function
```

Using the example basis $B = [0, 8, 24]$ the following steps are constructed when calling ‘recursive_ultimate_midpoint(B)‘:

$$[0, 8, 24]$$
$$[(0+8)/2 = 4, (8+24)/2 = 16]$$
$$[(4+16)/2 = 10]$$

The sequence of bases generated is:

$$[0, 8, 24]$$

$$[4, 16]$$
$$[10]$$

This does create a triangular appearance that when put into a matrix forms an upper triangular matrix:

$$\begin{pmatrix} 0 & 8 & 24 \\ 0 & 4 & 16 \\ 0 & 0 & 10 \end{pmatrix}$$

The main features of this matrix are the following:

- The first row is equivalent to the initial B basis.
- The diagonal or sometimes referred to as the leading edge forms a new basis B'.

Clearly this ‘successive_midpoint‘ function performs in $\mathcal{O}(n)$ time complexity while attempting to be in-place with memory.

Since each round of ‘successive_midpoint‘ functions returns a basis with one less element, for a basis B of size n, it would take $n-1$ iterations for this to output the ultimate midpoint of one element. Therefore generating the ultimate midpoint in this successive midpoint fashion is in $\mathcal{O}(n^2)$.

The following proposed improved ‘ultimate_midpoint‘ algorithm seeks to reduce this time complexity.

Algorithm 3 Optimized Ultimate Midpoint Calculation

```
1: function ULTIMATE_MIDPOINT($B$ :: basis)
2:     if $B == []$ then return $B$
3:     end if
4:     $n \leftarrow |B|$
5:     running_sum ← 0
6:     for $i \leftarrow 0 \dots n-1$ do
7:         running_sum ← running_sum + $\binom{n-1}{i} \cdot B[i]$
8:     end for
9:     ultimate_midpoint_value ← running_sum/$2^{n-1}$    ▷ Using ‘» (n-1)‘ for
   efficiency if values are integers return [ultimate_midpoint_value]
10: end function
```

Assuming that the binomial coefficient calculations are derived from some lookup table this method allows for an $\mathcal{O}(n)$ time to recover the ultimate midpoint.

This is quite the increase in efficiency.

If a leading edge is required to be calculated, being the basis calculated from the ultimate midpoints of the following bases, $B[0:1]$, $B[0:2]$, $\dots$, B, the time complexity is the following.

Algorithm 4 Leading Edge Calculation

```
function LEADING_EDGE($B$ :: basis)
    if $B == []$ then return $B$
    end if
    $n \leftarrow |B|$
    $B\_prime = []$
    for $i \leftarrow 0 \dots n - -1$ do
        $B\_prime.\text{append}(\text{ULTIMATE\_MIDPOINT}(B[0:i]))$
    end forreturn $B\_prime$
end function
```

The generation of this is clearly still $\mathcal{O}(n^2)$, but this does not generate any of the elements of each successive basis like in 'successive_midpoint', besides from the first element which makes up the leading edge.

Applying 'leading_edge' to the same example $B = [0, 8, 24]$ would have the following flow.

- 'ultimate_midpoint([0])'
  - $B\_prime = [1 \cdot 0/2^0 = 0]$
- 'ultimate_midpoint([0, 8])'
  - $B\_prime = [(1 \cdot 0 + 1 \cdot 8)/2^1 = 4]$
- 'ultimate_midpoint([0, 8, 24])'
  - $B\_prime = [(1 \cdot 0 + 2 \cdot 8 + 1 \cdot 24)/2^2 = (0 + 16 + 24)/4 = 40/4 = 10]$

Thus, $B\_prime = [0, 4, 10]$.

Anti Midpoint

Previously, ultimate midpoints have been calculated upon a basis B to yield B' or an element of B'. This was either conducted using successive midpoint mappings or using the binomial coefficient weighted average.

Now the other way must be considered.

Could a leading edge basis B' be used to generate the initial basis B? Using the previous example,

$$\begin{pmatrix} 0 & 8 & 24 \\ 0 & 4 & 16 \\ 0 & 0 & 10 \end{pmatrix}$$

where 'leading_edge(B) -> B'', 'leading_edge([0, 8, 24]) -> [0, 4, 10]', how could the inverse be defined? 'inverse_leading_edge([0, 4, 10]) -> [0, 8, 24]'. This would generate the basis needed to generate the series of ultimate midpoints resulting in the leading edge. Additionally, it is apparent to see that $b_0 = b'_0$ since the midpoint of a single element is that element, $(b_0 + b_0)/2 = b_0 = b'_0$.

The midpoint of two numbers a and b is $(a+b)/2 = c$ where c is the midpoint. Solving for a results in the following equation, $2c - b = a$. This means that by inputting a midpoint and an adjacent point, the other adjacent point can be found.

As an example let $B' = [0, 4, 10]$ where B' is the leading edge of a successive midpoint mapping on B. Since B' is the left edge then $b_0 = 0 = b'_0$. From the definition of successive midpoints: $b'_1 = (b_0 + b_1)/2 \implies 4 = (0 + b_1)/2$ $b'_2 = ((b_0 + b_1)/2 + (b_1 + b_2)/2)/2 \implies 10 = (b_0 + 2b_1 + b_2)/4$

Or using proof 1 (the coefficients are different in the formula from proof 1 for a direct calculation of b'_n): $4 = (0 + b_1)/2$ $10 = (1 \cdot 0 + 2 \cdot b_1 + 1 \cdot b_2)/4$

Clearly these linear equations can be solved: From the first equation: $8 = b_1$. From the second equation: $40 = 2b_1 + b_2 \implies 40 = 2 \cdot 8 + b_2 \implies 40 = 16 + b_2 \implies b_2 = 24$.

Thus, $B = [0, 8, 24]$.

This construction is based upon the Anti Midpoint. Taking successive mappings of B' using $2x - y$ where the index of x is greater than the index of y will result in the basis B.

Using this on the example $B' = [0, 4, 10]$:

$$\begin{aligned} &[0, 4, 10] \\ &[2 \cdot 4 - 0 = 8, 2 \cdot 10 - 4 = 16] \\ &[2 \cdot 16 - 8 = 24] \end{aligned}$$

Which has a leftmost edge of $B = [0, 8, 24]$.

Expanding out each of these rows in a general form is:

$$\begin{aligned} b_0 &= b'_0 \\ b_1 &= 2b'_1 - b'_0 \\ b_2 &= 2(2b'_2 - b'_1) - (2b'_1 - b'_0) = 4b'_2 - 4b'_1 + b'_0 \end{aligned}$$

It is not immediately clear what pattern these coefficients might follow. Although some sort of binomial coefficients are probable due to the nature of the combinations.

The coefficients of the recovered B elements, when expressed in terms of B' elements, follow this pattern. For b_0: $[1]$ For b_1: $[-1, 2]$ For b_2: $[1, -4, 4]$

Using the Online Encyclopedia of Integer Sequences, this sequence appears to potentially explain this sequence. The triangle $T(n, k)$ of coefficients relating to Bézier curve continuity is defined as $T(n, k) = (-1)^k \cdot 2^{n-k} \cdot \binom{n}{k}$.

If this were to be the correct formulation of coefficients, then the formula to achieve the successive Anti Midpoint (i.e., to recover b_n from $b'_0, \ldots, b'_n$) would be as follows:

$$b_n = \sum_{i=0}^{n} (-1)^i \cdot 2^{(n-i)} \binom{n}{i} b'_{n-i}$$

By using the symmetry of binomial coefficients $\binom{n}{i} = \binom{n}{n-i}$ and re-indexing the sum, this can also be written as:

$$b_n = \sum_{j=0}^{n} (-1)^{(n-j)} \cdot 2^j \binom{n}{j} b'_j$$

Where $n = |B'| - 1$ and b'_j are the elements of B'.

Proof 2: Successive Anti Midpoints - Proof by Induction

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

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

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

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:

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

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

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

$$\begin{aligned}
b_k &= 2\left(\sum_{i=0}^{k-1} (-1)^{(k-1-i)} \cdot 2^{(i)} \binom{k-1}{k-1-i} b'_{(i+1)}\right) \\
&\quad - \left(\sum_{i=0}^{k-1} (-1)^{(k-1-i)} \cdot 2^{(i)} \binom{k-1}{k-1-i} b'_i\right) \\
&= \sum_{i=1}^{k} (-1)^{(k-1-(i-1))} \cdot 2^{(i)} \binom{k-1}{k-1-(i-1)} b'_i \\
&\quad - \sum_{i=0}^{k-1} (-1)^{(k-1-i)} \cdot 2^{(i)} \binom{k-1}{k-1-i} b'_i \\
&= \sum_{i=1}^{k} (-1)^{(k-i)} \cdot 2^i \binom{k-1}{k-i} b'_i + \sum_{i=0}^{k-1} (-1)^{(k-i)} \cdot 2^{(i)} \binom{k-1}{k-1-i} b'_i \\
&= (-1)^{(k-k)} \cdot 2^k \binom{k-1}{k-k} b'_k \\
&\quad + \sum_{i=1}^{k-1} (-1)^{(k-i)} \cdot 2^i \binom{k-1}{k-i} b'_i \\
&\quad + \sum_{i=1}^{k-1} (-1)^{(k-i)} \cdot 2^{(i)} \binom{k-1}{k-1-i} b'_i \\
&\quad + (-1)^{(k-0)} \cdot 2^{(0)} \binom{k-1}{k-1-0} b'_0 \\
&= \left((-1)^{(k-k)} \cdot 2^k \binom{k-1}{k-k} b'_k\right) \\
&\quad + \left(\sum_{i=1}^{k-1} \left((-1)^{(k-i)} \cdot 2^i b'_i\right) \left(\binom{k-1}{k-i} + \binom{k-1}{k-1-i}\right)\right) \\
&\quad + \left((-1)^{(k-0)} \cdot 2^{(0)} \binom{k-1}{k-1-0} b'_0\right)
\end{aligned}$$

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

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

$$= \sum_{i=0}^{k} \left((-1)^{(k-i)} \cdot 2^i \binom{k}{k-i} b'_i \right) \quad \text{QED.}$$

Successive Anti Midpoints: Examples and Visualizations

As before, taking the leading edge $B' = [0, 4, 10]$, the resultant basis B that would be used to generate this basis could be found by successive applications of $2c - b = a$, where the index of c is larger than the index of b.

Although the previously proven formulation states that the basis can be found using a similar method to the ‘`leading_edge`‘ function.

The functions are as follows.

Algorithm 5 Successive Anti Midpoint

```
1: function SUCCESSIVE_DETERMINISTIC_INVERSE_MIDPOINT($B'$ :: basis)
2:     $n \leftarrow |B'|$
3:     if $n < 2$ then return $B'$
4:     end if
5:     for $i \leftarrow 0 \ldots n-2$ do
6:         $B'[i] \leftarrow 2 \cdot B'[i+1] - B'[i]$        ▷ Original formulation $2c - b = a$
7:     end for
8:     $B'$. pop_back() return $B'$
9: end function
```

Algorithm 6 Recursive Ultimate Initial Basis Point

```
1: function RECURSIVE_ULTIMATE_INITIAL_BASIS_POINT($B'$ :: basis)
2:     $n \leftarrow |B'|$
3:     if $n < 2$ then return $B'$
4:     end if
5:     while $n >= 1$ do
6:         $B' \leftarrow$ SUCCESSIVE_DETERMINISTIC_INVERSE_MIDPOINT($B'$)
7:         $n \leftarrow n - 1$
8:     end whilereturn $B'$
9: end function
```

Algorithm 7 Ultimate Initial Basis Point

```
function ULTIMATE__INITIAL__BASIS__POINT($B'$ :: basis)
    if $B' == []$ then return $B'$
    end if
    $n \leftarrow |B'|$
    running_sum $\leftarrow 0$
    for $i \leftarrow 0 \dots n-1$ do
        running_sum $\leftarrow$ running_sum $+ ((-1)^{(n\,-\,1\,-\,i)}) \cdot (2^i) \cdot \binom{n-1}{n-1-i} \cdot B'[i]$
    end for
    return [running_sum]
end function
```

Algorithm 8 Initial Basis Calculation

```
function INITIAL__BASIS($B'$ :: basis)
    if $B' == []$ then return $B'$
    end if
    $n \leftarrow |B'|$
    $B = []$
    for $i \leftarrow 0 \dots n-1$ do
        $B$.append(ULTIMATE__INITIAL__BASIS__POINT($B'[0:i]$))
    end forreturn $B$
end function
```

Applications of Successive Midpoints and Anti Midpoints

As of this point, the basis B has been composed of any integer values in any order. Although taking specific bases can lend to interesting or desired results.

One such basis is $B = [b^x \mid x \in [0, \dots, n-1]]$, where b is some real number and n is the desired length of B.

Since $B = [b^0, b^1, \dots, b^{n-1}]$, the first element of B' would be b^0. The second element would be $(b^0 + b^1)/2$. The third element would be $(b^0 + 2b^1 + b^2)/4$ and so on.

Corollary 1

Based upon Proof 1, the formula found to be true for the ultimate midpoint is $m = \frac{1}{2^{n-1}} \sum_{i=0}^{n-1} \binom{n-1}{i} b_i$ (where n is the length of the input basis, so $n-1$ is the power of 2 and the upper index of summation). Let's use the current length of B as N, so $N = |B|$. The formula is $\frac{1}{2^{N-1}} \sum_{i=0}^{N-1} \binom{N-1}{i} b^i$.

By using the binomial theorem, $(a+b)^{N-1} = \sum_{i=0}^{N-1} \binom{N-1}{i} a^{N-1-i} b^i$. If we let $a = 1$, then $(1+b)^{N-1} = \sum_{i=0}^{N-1} \binom{N-1}{i} 1^{N-1-i} b^i = \sum_{i=0}^{N-1} \binom{N-1}{i} b^i$.

Therefore, substituting this into the ultimate midpoint formula:

$$\begin{aligned} m &= \frac{1}{2^{N-1}} \sum_{i=0}^{N-1} \binom{N-1}{i} b^i \\ &= \frac{1}{2^{N-1}} (1+b)^{N-1} \\ &= \left(\frac{1+b}{2}\right)^{N-1} \end{aligned}$$

Therefore, taking successive midpoints of an exponential basis $B = [b^x \mid x \in [0, \dots, N-1]]$, the resultant leading edge element b'_{N-1} is $\left(\frac{1+b}{2}\right)^{N-1}$.

This means that the *ultimate midpoint* of such a basis is also an exponential term. To find a specific exponent b', if we set the new base to $b' = (1+b)/2$, then the ultimate midpoint of B will be $(b')^{N-1}$.

Example $B = [1, 3, 9]$ (here $N = 3$, so b^0, b^1, b^2). Base $b = 3$. From the corollary, the new base $b' = (1+3)/2 = 2$. The ultimate midpoint should be $2^{3-1} = 2^2 = 4$. Let's demonstrate with successive midpoints:

$$\begin{aligned} &[1, 3, 9] \\ &[(1+3)/2 = 2, (3+9)/2 = 6] \\ &[(2+6)/2 = 4] \end{aligned}$$

The sequence of bases generated is:

$$\begin{aligned} &[1, 3, 9] \\ &[2, 6] \\ &[4] \end{aligned}$$

The ultimate midpoint is 4. The leading edge B' (ultimate midpoints of $B[0:1]$, $B[0:2]$, $B[0:3]$) would be $[1, 2, 4]$. This is indeed an exponential basis with base $b' = 2$.

This process will only yield exponential bases with base b' that have a smaller magnitude than that of the input base b if $b > -1$ and a larger magnitude if $b < -1$.

In order to find exponential bases in the other direction (a larger base b' if $b < -1$ and a smaller base b' if $b > -1$), the Anti Midpoint should be employed.

Corollary 2

Using the formula from Proof 2, for recovering b_n from $B' = [b'_0, \dots, b'_n]$: $b_n = \sum_{i=0}^{n} (-1)^{(n-i)} 2^i \binom{n}{n-i} b'_i$. Let $B' = [b'^x \mid x \in [0, \dots, n]]$. So $b'_i = (b')^i$. We want to show this yields $(2b'-1)^n$.

Let's work through the sum:

$$b_n = \sum_{i=0}^{n} (-1)^{(n-i)} 2^i \binom{n}{n-i} (b')^i$$

$$
\begin{aligned}
&= \sum_{i=0}^{n} (-1)^n (-1)^{-i} 2^i \binom{n}{n-i} (b')^i \\
&= \sum_{i=0}^{n} (-1)^n \frac{1}{(-1)^i} 2^i \binom{n}{n-i} (b')^i \\
&= (-1)^n \sum_{i=0}^{n} \left(\frac{2b'}{-1}\right)^i \binom{n}{n-i} \\
&= (-1)^n \sum_{i=0}^{n} (-2b')^i \binom{n}{n-i}
\end{aligned}
$$

Using the symmetry of binomial coefficients $\binom{n}{n-i} = \binom{n}{i}$:

$$= (-1)^n \sum_{i=0}^{n} \binom{n}{i} (-2b')^i$$

Using the binomial theorem $(a+x)^n = \sum_{i=0}^{n} \binom{n}{i} a^{n-i} x^i$. If we set $a = 1$ and $x = (-2b')$, then $\sum_{i=0}^{n} \binom{n}{i} (1)^{n-i} (-2b')^i = (1-2b')^n$. So:

$$
\begin{aligned}
b_n &= (-1)^n (1-2b')^n \\
&= (-1)^n (-(2b'-1))^n \\
&= (-1)^n (-1)^n (2b'-1)^n \\
&= (1)(2b'-1)^n \\
&= (2b'-1)^n
\end{aligned}
$$

Using an example $B' = [1, 4, 16]$ (here $n = 2$, so b'^0, b'^1, b'^2). Base $b' = 4$. Corollary 2 states that the resultant exponential basis base should be $b = 2 \cdot 4 - 1 = 7$. Then the recovered basis B should be $[1, 7, 49]$. Using the successive Anti Midpoint process:

$$
\begin{aligned}
&[1, 4, 16] \\
&[2 \cdot 4 - 1 = 7,\ 2 \cdot 16 - 4 = 28] \\
&[2 \cdot 28 - 7 = 49]
\end{aligned}
$$

The sequence of bases generated is:

$$
\begin{aligned}
&[1, 4, 16] \\
&[7, 28] \\
&[49]
\end{aligned}
$$

The ultimate recovered element is 49. The initial basis B (recovering b_0, b_1, b_2) would be $[1, 7, 49]$. This is indeed an exponential basis with base $b = 7$.

Now it has been shown that any real number exponential basis can be derived from some another real number exponential base by either successive midpoints or Anti Midpoints.

You can have recursive calls of these functions where the leading edge generated by one base becomes the new basis for another leading edge to be calculated.

So as an example, starting with a base of $b = -1.5$ and using the successive midpoint method of Corollary 1 twice would result in the following.

First pass: $b_1 = (-1.5 + 1)/2 = -0.25$. Second pass: $b_2 = (-0.25 + 1)/2 = 0.375$.

Setting $N = 3$.

Pass 1: $B = [1, -1.5, 2.25]$ (since $(-1.5)^0 = 1$, $(-1.5)^1 = -1.5$, $(-1.5)^2 = 2.25$). Applying successive midpoints:

$$\begin{aligned}
&[1, -1.5, 2.25] \\
&[(1 + -1.5)/2 = -0.25,\ (-1.5 + 2.25)/2 = 0.375] \\
&[(-0.25 + 0.375)/2 = 0.0625]
\end{aligned}$$

The sequence of bases generated is:

$$\begin{aligned}
&[1, -1.5, 2.25] \\
&[-0.25, 0.375] \\
&[0.0625]
\end{aligned}$$

The leading edge B' from this pass is $[1, -0.25, 0.0625]$. This is an exponential basis with base -0.25. Note that the 0.375 on the second row is an intermediate calculation, not the base of the final exponential basis.

Pass 2: Setting the B' of Pass 1 as the B for this pass. $B = [1, -0.25, 0.0625]$ (exponential basis with base -0.25). Applying successive midpoints:

$$\begin{aligned}
&[1, -0.25, 0.0625] \\
&[(1 + -0.25)/2 = 0.375,\ (-0.25 + 0.0625)/2 = -0.09375] \\
&[(0.375 + -0.09375)/2 = 0.140625]
\end{aligned}$$

The sequence of bases generated is:

$$\begin{aligned}
&[1, -0.25, 0.0625] \\
&[0.375, -0.09375] \\
&[0.140625]
\end{aligned}$$

The leading edge B' from this pass is $[1, 0.375, 0.140625]$. This is an exponential basis with base 0.375.

Therefore, $0.375^2 = 0.140625$. Additionally, it can be presumed that $375^2 = 140625$ if scaled.

Although this formulation is interesting in generating some exponential based upon some known base, it would be more useful to be able to generate any exponential based off of a known base.

Powers of Two and Basis Transformations

Midpoints are comprised of two operations: addition and right bit shifts (division by 2). Anti Midpoints are comprised of the inverse of these operations: subtraction and left bit shifts (multiplication by 2).

These are generally considered to be operations that the computer can calculate in very few ALU cycles, if not one.

By starting with a basis B that is the exponential basis of a power of 2, any integer number base can be achieved only through a series of midpoints, Anti Midpoints, and bit shifts.

Since b' is an integer, its even/odd parity dictates which operation to use.

If b' is odd then performing the following equation $(b'+1)/2$ or '$\lceil b'/2 \rceil$' will result in a new smaller integer b'. If b' is even then performing the right bit shift $b'/2$ or '$\lceil b'/2 \rceil$' as well will result in a new smaller integer b'.

Therefore repeatedly performing '$\lceil b'/2 \rceil$' will generate the required mapping to reconstruct the original base.

Let $f(x) = \lceil x/2 \rceil$. Even though this seems to be wildly incorrect at first glance, it appears to be true that f composed with f n times is equivalent to $\lceil x/2^n \rceil$ by seemingly distributing the division by 2.

Proof 3

$f(x) = \lceil x/2 \rceil$, $f^n(x) = \lceil x/2^n \rceil$ where f^n is successive compositions of the function and n is the amount of compositions $+1$.

The identity

$$\operatorname{ceil}\left(\frac{x+m}{n}\right) = \operatorname{ceil}\left(\frac{\operatorname{ceil}(x)+m}{n}\right)$$

Theorem 3.11 Graham, Knuth, & Patashnik, p. 72 `https://theswissbay.ch/pdf/Gentoomen%20Library/Maths/Comp%20Sci%20Math/Concrete%20Mathematics%20A%20Foundation%20for%20Computer%20Science~tqw~_darksiderg.pdf`.

By setting $m = 0$ and $n = 2^k$:

$$\operatorname{ceil}\left(\frac{\operatorname{ceil}(x)}{2^k}\right) = \operatorname{ceil}\left(\frac{x}{2^k}\right)$$

Setting x to $x/2$:

$$\operatorname{ceil}\left(\frac{\operatorname{ceil}\left(\frac{x}{2}\right)}{2^k}\right) = \operatorname{ceil}\left(\frac{x}{2^{(k+1)}}\right)$$

This allows for successive ceiling divisions to be collected.

References

[1] Graham, R. L., Knuth, D. E., & Patashnik, O. (1994). *Concrete Mathematics: A Foundation for Computer Science* (2nd ed.). Addison-Wesley.

Proof 4

Clearly successively applying f onto some b' will eventually yield a result of 0 if $b' \leq 0$ or a result of 1 if $b' > 0$.

There are 4 different conceptual steps that a b' can take to be transformed into the eventual base 2 basis: ‘$b' < 0 \to 0 \to 0.5 \to 1 \to 2$‘ ‘ $\uparrow$‘

‘ $0 < b'$‘

For all integer b', there exists some path that can be taken to transform it into 2 by only performing midpoint by 1 operations or right bit shifts (‘$\lceil b'/2 \rceil$‘) and one left bit shift. Wherever an integer b' starts, it can be put on the path and follow the remaining steps to yield the desired result of 2.

Step 1: $b' < 0$ (Turning b' into 0)

Clearly when b' is less than 0, ‘$\lim_{n\to\infty} \lceil b'/2^n \rceil = 0$‘. Since for all natural numbers n, $b'/2^n$ will always be slightly less than 0 (for $b' < 0$), which rounds up to 0.

To find the minimum n required to round up to 0 is as follows: We want ‘$\lceil x/2^n \rceil = 0$‘, which implies ‘$-1 < x/2^n \leq 0$‘. Since $x < 0$, we have ‘$x/2^n \leq 0$‘ naturally. We need to satisfy ‘$-1 < x/2^n$‘. ‘

$$\begin{aligned} \frac{x}{2^n} &> -1 \\ x &> -2^n \\ -x &< 2^n \end{aligned}$$

‘ Since $x = b' < 0$, we substitute $-x = |b'|$. ‘

$$\begin{aligned} |b'| &< 2^n \\ \log_2(|b'|) &< n \end{aligned}$$

‘ Therefore, for an n greater than ‘$\log_2(|b'|)$‘, successive ceiling divisions ‘$\lceil b'/2^n \rceil$‘ will result in 0. The number of times the successive ‘$\lceil b'/2 \rceil$‘ must be taken to yield 0 is ‘$\lfloor \log_2(|b'|) \rfloor + 1$‘ (for $b' \neq 0$).

Step 2: Turning $b' = 0$ into 0.5

When $b' = 0$, taking the midpoint of this value with 1, ‘$(0+1)/2$‘, yields 0.5.

Step 3: $0 < b'$ (Turning b' into 2)

‘$\lim_{n\to\infty} \lceil b'/2^n \rceil = 1$‘. Since for all natural numbers n, $b'/2^n$ will always be slightly larger than 0 (for $b' > 0$), which rounds up to 1. This 1 can then be left bit shifted once to achieve 2.

Now, let's consider the number of times ‘$\lceil b'/2 \rceil$‘ must be applied to reach 1 or 2.

Case 1: If $0 < b' \leq 2$ A single pass of ‘$\lceil b'/2 \rceil$‘ is necessary to result in 1.

- If $b' = 1$, ‘$\lceil 1/2 \rceil = 1$‘.
- If $b' = 2$, ‘$\lceil 2/2 \rceil = 1$‘.

A single left bit shift ‘$(1 \ll 1)$‘ will then result in a base of 2.

Case 2: If $2 < b'$ We want to find n such that ‘$\lceil b'/2^n \rceil = 2$‘. This implies ‘$1 < b'/2^n \leq 2$‘. Focus on ‘$b'/2^n \leq 2$‘: ‘

$$\begin{aligned} b' &\leq 2 \cdot 2^n \\ b' &\leq 2^{n+1} \\ \log_2(b') &\leq n+1 \\ \log_2(b') - 1 &\leq n \end{aligned}$$

‘ Therefore, the number of times that successive ‘$\lceil b'/2 \rceil$‘ must be taken upon b' to return 2 is ‘$\lfloor \log_2(b') - 1 \rfloor + 1 = \lfloor \log_2(b') \rfloor$‘. This represents the number of times ‘$\lceil b'/2 \rceil$‘ must be applied to yield 2.

Therefore for all integers b', a base of 2 can be reached by only performing operations corresponding to successive midpoints (when b' is odd) or right bit shifts (when b' is even), combined with the special case of $(0+1)/2$ and left bit shifts.

Applications

Example $b' = 10$ Since $b' > 0$ and $b' > 2$, we apply ‘$\lceil b'/2 \rceil$‘ repeatedly until we reach 2. ‘$\lfloor \log_2(10) \rfloor = \lfloor 3.32 \rfloor = 3$‘. So 3 applications are expected.

- ‘$\lceil 10/2 \rceil = 5$‘
- ‘$\lceil 5/2 \rceil = \lceil 2.5 \rceil = 3$‘
- ‘$\lceil 3/2 \rceil = \lceil 1.5 \rceil = 2$‘ (Reached 2 in 3 steps, as expected)

Example $b' = -23$

- **Step 1: Turning** $b' = -23$ **into** 0 We need ‘$\lfloor \log_2(|-23|) \rfloor + 1 = \lfloor \log_2(23) \rfloor + 1 = \lfloor 4.52 \rfloor + 1 = 4 + 1 = 5$‘ applications.
  - ‘$-23 \to \lceil -23/2 \rceil = \lceil -11.5 \rceil = -11$‘ (Odd)
  - ‘$-11 \to \lceil -11/2 \rceil = \lceil -5.5 \rceil = -5$‘ (Odd)
  - ‘$-5 \to \lceil -5/2 \rceil = \lceil -2.5 \rceil = -2$‘ (Odd)
  - ‘$-2 \to \lceil -2/2 \rceil = -1$‘ (Even)
  - ‘$-1 \to \lceil -1/2 \rceil = 0$‘ (Odd) (Desired result in 5 steps)
- **Step 2: Turning** 0 **into** 0.5 Taking the midpoint of 0 with 1: ‘$(0+1)/2 = 0.5$‘.

- **Step 3: Turning 0.5 into 2** Apply '$\lceil x/2 \rceil$' until 1 is reached, then left bit shift.
    - '$\lceil 0.5/2 \rceil = \lceil 0.25 \rceil = 1$'
    - '$(1 \ll 1) = 2$'

Using this example, it can be shown that performing the inverse of these steps can result in the original base -23 from 2.

Inverse Construction for $b' = -23$

We recorded the "parity" (or operation type) during the reduction: '[Odd, Odd, Odd, Even, Odd]' for the '$\lceil b'/2 \rceil$' steps to reach 0. The inverse operations are:

- Inverse of odd rule ('$b \to (b+1)/2$') is '$b' \to 2b' - 1$'
- Inverse of even rule ('$b \to b/2$') is '$b' \to 2b'$'

The path was '2 -> 1 -> 0.5 -> 0 -> -1 -> -2 -> -5 -> -11 -> -23'. Let's reverse it.

Let's trace the inverse path from base 2:

1. Start from base 2.
2. Inverse of '$(1 \ll 1)$' (which resulted in 2) is '$(2 \gg 1) = 1$'. Current base is 1.
3. Inverse of '$\lceil 0.5/2 \rceil = 1$'. The operation was '$\lceil y/2 \rceil = 1$'. The specific y that mapped to 1 via this operation was $y = 0.5$. Current base is 0.5.
4. Inverse of '$(0+1)/2 = 0.5$'. This is a midpoint. If $C = (A+B)/2$, then $A = 2C - B$. Here $C = 0.5, B = 1$. So $A = 2(0.5) - 1 = 0$. Current base is 0.
5. Now, we use the recorded '[Odd, Odd, Odd, Even, Odd]' sequence in reverse to reconstruct from 0:
    - Current base $b = 0$. Last reduction was 'Odd' ('$-1 \to 0$'). Inverse: '$(2 \cdot 0) - 1 = -1$'. Current base is -1.
    - Current base $b = -1$. Reduction was 'Even' ('$-2 \to -1$'). Inverse: '$(2 \cdot -1) = -2$'. Current base is -2.
    - Current base $b = -2$. Reduction was 'Odd' ('$-5 \to -2$'). Inverse: '$(2 \cdot -2) - 1 = -5$'. Current base is -5.
    - Current base $b = -5$. Reduction was 'Odd' ('$-11 \to -5$'). Inverse: '$(2 \cdot -5) - 1 = -11$'. Current base is -11.
    - Current base $b = -11$. Reduction was 'Odd' ('$-23 \to -11$'). Inverse: '$(2 \cdot -11) - 1 = -23$'. Current base is -23.

The base of -23 has been successfully recovered.

The following algorithm captures the methodologies proven within Proof 4.

```
import math

def inverse_dyadic_mapping(base):
    inverse_base_address = []

    if (base < 0):
        inverse_base_address += step_1(base)
        base = 0

    if (base == 0):
        inverse_base_address += step_2(base)
        base = .5

    if (0 < base and base <= 2):
        inverse_base_address += step_3_case_1(base)
        base = 2

    if (2 < base):
        inverse_base_address += step_3_case_2(base)
        base = 2

    return inverse_base_address

def step_1(base):
    n = math.floor(math.log2(abs(base))) + 1
    inverse_base_address = []
    for i in range(n):
        inverse_base_address.append(base % 2)
        base = math.ceil(base / 2)
    return inverse_base_address

def step_2(base):
    base = (base + 1) / 2
    return [-1]

def step_3_case_1(base):
    base = math.ceil(base / 2)
    base = base << 1
    return ['.']

def step_3_case_2(base):
    n = math.floor(math.log2(abs(base)))
    inverse_base_address = []
    for i in range(n):
        inverse_base_address.append(base % 2)
        base = math.ceil(base / 2)
    return inverse_base_address
```

What is returned from the main ‘inverse_dyadic_mapping‘ function is the instructions on how to return to the original basis. The base gets encoded into an address.

The decoding of the address to the base is simply as follows.

```
def forwards_dyadic_map(base_address):
    base = 2
    for i in base_address:
        if (i == 0):
            base = base << 1

        if (i == 1):
            base = (base << 1) - 1

        if (i == "."):
            base = .5

        if (i == -1):
            base = 0
    return base
```

If the trivial bases $b' = 2$, $b' = 1$, $b' = 0$ are removed, then the encoding can be much more simple.

```
import math

def simple_inverse_dyadic_mapping(desired_base):
    # Assuming that the inputs are never base = 0, 1, 2

    n = math.floor(math.log2(abs(desired_base)))
    inverse_base_address = ['.'] * (n)

    if (desired_base < 0):
        n += 1
        inverse_base_address += ['.','.']

    for i in range(n):
        inverse_base_address[i] = (desired_base % 2)
        desired_base = math.ceil(desired_base / 2)

    return inverse_base_address
```

```
def simple_forwards_dyadic_map(base_address):
    desired_base = 2
    for i in base_address:
        if (i == 0):
            desired_base = desired_base << 1

        if (i == 1):
            desired_base = (desired_base << 1) - 1

        if (i == "."):
            desired_base = 0

    return desired_base
```

Since the logic in the previous steps 1 and steps 3.2 were essentially the same besides from a count of one, they were combined. Additionally, since a desired base of 0, 1, or 2 are no longer inputs, steps 3.1 and steps 2 can be combined to be the single ‘.‘ operator.

The ‘.‘ operator is ‘$.(b) = \frac{b}{2} - 1$‘ (bit shifting b right once followed by subtraction by 1) where the inverse is ‘$2(b' + 1)$‘ (an addition by one followed by a left bit shift of 1). In this context, the ‘.‘ is only used when transforming the base from ‘$0 \leftrightarrow 2$‘, which is possible due to the domain restrictions imposed upon the base.

Clearly the running time complexity of these algorithms is $O(\log n)$ since the amount of iterations is simply the logarithm of the input base.

This means that to encode a base x it takes y bits, ‘$\lfloor \log_2(|x|) \rfloor = y$‘. This implies ‘$\log_2(|x|) \approx y$‘, so ‘ $|x| \approx 2^y$‘. Thus, the largest base to be encoded in this fashion is ‘$\approx 2^y$‘.

This means that a standard 32-bit integer will be able to encode a positive base up to 2^{32} or about 4 billion, and a 64-bit integer can encode a positive base up to 2^{64} or about 18 quintillion.

Finally all of the pieces as described within this writeup can be combined to yield the final algorithm

Generating Integer Exponentials Using Only Bit Shifting and Addition on Powers of 2

Example

Assume that a user wishes to know 29^2. Let $b' = 29$, $n = 2 + 1 = 3$. (The exponent is 2, so we need terms up to b^2, which means $n = 3$ elements in the basis, $x \in [0, 1, 2]$).

First, the ‘simple_inverse_dyadic_mapping(29)‘ should be conducted to encode the base into the address space. This is done as follows:

- $29 \to \lceil 29/2 \rceil = 15$ (Odd)
- $15 \to \lceil 15/2 \rceil = 8$ (Odd)
- $8 \to \lceil 8/2 \rceil = 4$ (Even)
- $4 \to \lceil 4/2 \rceil = 2$ (Even)
- $2 \to$ End (The mapping stops here, or rather, the simplified map uses this as an implicit end point)

Therefore, the address (read from bottom-up for inverse construction, or top-down if we consider the parity list in reverse) of 29 is '[1,1,0,0]'.

There will be 4 steps in reconstructing this base: $2 \to 4 \to 8 \to 15 \to 29$, where 2 is the starting base for the reconstruction.

Now let $B = [b^x \mid x \in [0 \dots n-1]]$, starting off where $b = 2$, and for $n = 3$ (exponent 2), $B = [2^0, 2^1, 2^2] = [1, 2, 4]$.

Observing the inverse of the address, '[0,0,1,1]' (which is the original sequence '[1,1,0,0]' reversed) construction upon the base can begin.

Step 1

Current base $b = 2$, current address instruction $i = 0$ (meaning the reduction step was 'Even').

The even step means that each element of the basis B needs to be bit shifted j times, where j is the index of the element. This is because this should yield the next exponential basis B' of base $2b = b'$. If b^j is the current element, then $b^j \to (2b)^j = 2^j \cdot b^j$, which is equivalent to left bit shifting b^j by j positions. Original Basis: $[1, 2, 4]$ Operation: '$[1 \ll 0, 2 \ll 1, 4 \ll 2] = [1, 4, 16]$' New Basis: $[1, 4, 16]$. Current base $b = 4$.

Step 2

Current base $b = 4$, current address instruction $i = 0$ (meaning the reduction step was 'Even').

Operation: '$[1 \ll 0, 4 \ll 1, 16 \ll 2] = [1, 8, 64]$' New Basis: $[1, 8, 64]$. Current base $b = 8$.

Step 3

Current base $b = 8$, current address instruction $i = 1$ (meaning the reduction step was 'Odd').

The odd step implies an inverse midpoint operation: $A = 2C - B$. We apply this operation iteratively across the basis elements. Original Basis: $[1, 8, 64]$ For $j = 0$: '$(2 \cdot \text{basis}[1] - \text{basis}[0]) = (2 \cdot 8 - 1) = 15$' For $j = 1$: '$(2 \cdot \text{basis}[2] - \text{basis}[1]) = (2 \cdot 64 - 8) = 120$' This creates a new leading edge sequence which needs further reduction to find the single element for the next power. This part of the construction needs careful handling to generate the correct sequence. The provided method uses ‘get_leading_edge‘ which performs this reduction. Let’s trace the values from the example: From ‘[1, 8, 64]‘ ‘leading_edge = [1]‘ ‘basis = $[1, 8, 64]$‘ First iteration in ‘get_leading_edge‘ loop: ‘succesive_midpoint($[1, 8, 64]$)‘: ‘basis[0] = $(2 \cdot 8 - 1) = 15$‘ ‘basis[1] = $(2 \cdot 64 - 8) = 120$‘ ‘basis.$pop() \to [15, 120]$‘ ‘leading_edge.$append$(basis[0]) $\to [1, 15]$‘ ‘basis = $[15, 120]$‘ Second iteration in ‘get_leading_edge‘ loop: ‘succesive_midpoint($[15, 120]$)‘: ‘basis[0] = $(2 \cdot 120 - 15) = 225$‘ ‘basis.$pop() \to [225]$‘ ‘leading_edge.$append$(basis[0]) $\to [1, 15, 225]$‘ New Basis: $[1, 15, 225]$. Current base $b = 15$.

Step 4

Current base $b = 15$, current address instruction $i = 1$ (meaning the reduction step was ‘Odd‘).

Apply the ‘get_leading_edge‘ process again: Original Basis: ‘[1, 15, 225]‘ From ‘[1, 15, 225]‘ ‘leading_edge = [1]‘ ‘basis = $[1, 15, 225]$‘ First iteration in ‘get_leading_edge‘ loop: ‘succesive_midpoint($[1, 15, 225]$)‘: ‘basis[0] = $(2 \cdot 15 - 1) = 29$‘ ‘basis[1] = $(2 \cdot 225 - 15) = 435$‘ ‘basis.$pop() \to [29, 435]$‘ ‘leading_edge.$append$(basis[0]) $\to [1, 29]$‘ ‘basis = $[29, 435]$‘ Second iteration in ‘get_leading_edge‘ loop: ‘succesive_midpoint($[29, 435]$)‘: ‘basis[0] = $(2 \cdot 435 - 29) = 841$‘ ‘basis.$pop() \to [841]$‘ ‘leading_edge.$append$(basis[0]) $\to [1, 29, 841]$‘ New Basis: $[1, 29, 841]$. Current base $b = 29$.

The final element of the resulting basis is the recovered exponent: $29^2 = 841$.

The combined algorithms result in the following:

```
def simple_inverse_dyadic_mapping(desired_base):
    # Assuming that the inputs are never base = 0, 1, 2

    n = math.floor(math.log2(abs(desired_base)))
    inverse_base_address = ['.'] * (n)

    if (desired_base < 0):
        n += 1
        inverse_base_address += ['.','.']

    for i in range(n):
        inverse_base_address[i] = (desired_base % 2)
        desired_base = math.ceil(desired_base / 2)

    return inverse_base_address

def succesive_midpoint(basis):
    n = len(basis)
    if (n < 2):
        return basis
    for i in range(n - 1):
        basis[i] = (2 * basis[i + 1] - basis[i])
    basis.pop()
    return basis

def get_leading_edge(basis):
    leading_edge = [basis[0]]
    for i in range(len(basis) - 1):
        basis = succesive_midpoint(basis)
        leading_edge.append(basis[0])
    return leading_edge

def power(base, exponent):
    base_address = simple_inverse_dyadic_mapping(base)[::-1]

    working_basis = [2**n for n in range(exponent + 1)]
    for i in base_address:
        if (i == 0): # Even step (division by 2 in reduction)
            working_basis = [working_basis[j] << j for j in range(len(working_basis))]

        if (i == 1): # Odd step (b = (b'+1)/2 in reduction)
            working_basis = get_leading_edge(working_basis)

        if (i == "."): # Special handling for 0, 1, 2
            working_basis = [0 for j in range(exponent + 1)]

    return working_basis[-1]
```

Accidental Finding

For the function

```
def succesive_midpoint(basis):
    n = len(basis)
    if (n < 2):
        return basis
    for i in range(n - 1):
        basis[i] = (basis[i] + basis[i + 1]) / 1
    basis.pop()

    return basis
```

I accidentally set the denominator of the sum to 1 when changing the bit shifting to division to account for floating points.

The leading edge that resulted from the following function when inputting the powers of 2 base, returns

```
def get_leading_edge(basis):
    leading_edge = [basis[0]]
    for i in range(len(basis) - 1):
        basis = succesive_midpoint(basis)
        leading_edge.append(basis[0])
    return leading_edge
```

For an input ‘basis = [1, 2, 4, 8, ...]‘ (powers of 2), the ‘leading_edge‘ was ‘[1, 3.0, 9.0, 27.0, 81.0, 243.0, 729.0, 2187.0, 6561.0, 19683.0]‘. Which quite clearly is the powers of three. This was shocking since I was expecting powers of 1.5 instead, which is the result of the standard midpoint when dividing by 2.

It seems that this denominator can be varied to be different values to yield different exponentials rather than the pattern returned by the previously defined dyadic mapping, (‘$2x$‘ or ‘$2x-1$‘).

Setting this denominator to 1 seems to result in sequential exponents where b generates $b' = b + 1$. Setting it to 3 using powers of $b = 11$ yields $b' = 4$.

This mapping needs to be further explored.

Update

It seems that by plugging in any coefficients a_0, a_1 will generate exponentials where a_0, a_1 are rational numbers: ‘basis$[i]$ = $a_0 \cdot$ basis$[i]$ + $a_1 \cdot$ basis$[i+1]$‘ (Corrected from ‘a1 * basis[i] + a2 * basis[i + 1]‘ to a_0, a_1 for consistency with general notation).

This means that this scheme can be generalized even further for any mappings ‘$a_0 b_i + a_1 b_{i+1}$‘.

This is likely due to the binomial theorem, but it now allows for optimization on the power solver.

Example

Say that I want to find 1590002^2.

Before, with the dyadic mapping, this would result in a lengthy basis construction address ‘[0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0]‘ with 20 steps. Although now it appears that it can be found only in two steps by using the construction ‘$183 \cdot b_i + 847 \cdot b_{i+1}$‘ twice, starting with $b = 2$.

Step 1 Initial Basis: $[1, 2, 4]$ ‘succesive_midpoint_generalized($[1, 2, 4]$)‘ with coefficients $a_0 = 183, a_1 = 847$:

- ‘$(183 \cdot 1 + 847 \cdot 2) = 1877$‘
- ‘$(183 \cdot 2 + 847 \cdot 4) = 3754$‘

Resulting intermediate list: ‘$[1877, 3754]$‘ Applying ‘get_leading_edge‘ (which performs successive reductions):

- ‘$(183 \cdot 1877 + 847 \cdot 3754) = 3523129$‘

New Basis: $[1, 1877, 3523129]$. (The first element 1 is preserved as $2^0 = 1$).

Step 2 Initial Basis: $[1, 1877, 3523129]$ ‘succesive_midpoint_generalized($[1, 1877, 3523129]$)‘ with coefficients $a_0 = 183, a_1 = 847$:

- ‘$(183 \cdot 1 + 847 \cdot 1877) = 1590002$‘
- ‘$(183 \cdot 1877 + 847 \cdot 3523129) = 2984433754$‘

Resulting intermediate list: ‘$[1590002, 2984433754]$‘ Applying ‘get_leading_edge‘ (successive reductions):

- ‘$(183 \cdot 1590002 + 847 \cdot 2984433754) = 2528106360004$‘

New Basis: $[1, 1590002, 2528106360004]$.

$1590002^2 = 2528106360004$. Doing the actual arithmetic on this is quite odd, but it satisfyingly yields the exponents even if it seems like it is a linear combination of seemingly random numbers.

This no longer provides the added benefits of easy bit shifting that midpoints provide, but it does allow for much fewer steps to be taken. What needs to be done now is to find out how to efficiently find these a_0, a_1 coefficients and potentially investigate using $a_2, a_3, \ldots$ as well.

Further Topics

- Expanding the scheme to be able to handle any rational number input.
- **Optimizations**
  - Applying the symmetry of the binomials.
  - Using exponent by squaring.
  - Potentially expand the final exponent to be in terms of powers of 2, or maybe as binomial coefficients.
  - Finding a way to find a_0, a_1 coefficients to reduce the steps to find the exponent.
  - Finding a way to generate the base address without first preprocessing it.
  - For negative bases, using the positive base and determining the sign might be more efficent

Current Algorithm Time Complexity $O(n^2 \lg(b))$ where $n =$ exponent, $b =$ base. For large bases and small exponents this could be performant, but using the above optimizations will help decrease complexity.