Roots

Created: Jan 2018

Contents

List of Figures

The human mind is an iterative processor. It never does anything exactly right the first time. Tom DeMarco - Structured analysis and system specification

1 Abstract

How to calcualte the root of a given number x, where x provides a measure of size in the dimension n?

There exists a colorful spectrum of methods to calculate root, with impressing names, Babylonian method (Hero's method), Isaac Newton method, Bakshali method, Digit by digit calculation, Exponential Identity, Vedic Duplex Method, Goldschmidt's algorithm, Taylor Series, Brahmagupta equation, etc., Wikipedia enumerates and explains them [WIKIROOTS].

Iteration, Recursion and Approximation are the ingredients also for the approach used by the greek philosopher Zeno of Elea to get to the point where the fast and furious Achilles reaches - or not - the slow but similarly famous Tortoise.

There are some similarities in my approach for the root calculation and the Achilles-Tortoise race from Zeno. Unlike the Zeno approach, do my iterations end at some point.

The given paper shall document some own experimentation and considertion on the topic.

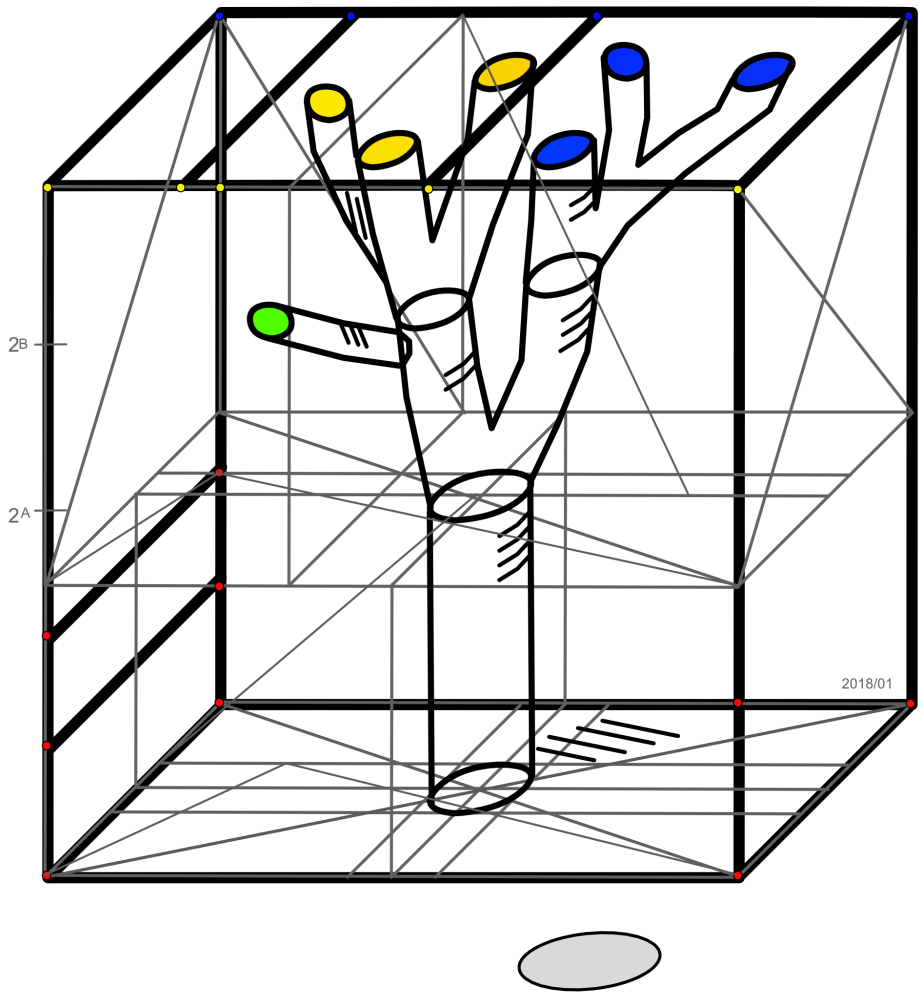

Figure 1: Tree - The roots are hidden

2 Definitions

2.1 Functions Definitions

$$exponentiation \ : \ p \equiv f(b,e) \equiv b^e \tag{2.1.1}$$

$$root \ : \ b \equiv f(p,e) \equiv \sqrt[e]{p} \tag{2.1.2}$$

$$logarithm \ : \ e \equiv f(b,p) \equiv log_b(p) \tag{2.1.3}$$

with base b, power p, and exponent e.

$$product \ : \ p \equiv f(a,b) \equiv a * b \tag{2.1.4}$$

with factors a, b and product p.

$$division \ : \ q \equiv f(a,b) \equiv a/b \tag{2.1.5}$$

with dividend, numerator a, divisor, denominator b and fraction, quotient or ratio q.

2.2 Iterative, Recursive, Referential Root Approximation

In the following formula we a looking for the value y, which is the n'th root for a given value x:

$$y = f(x, n) = \sqrt[n]{x} \tag{2.2.1}$$

or, reverse

$$x = f(y, n) = y^n \tag{2.2.2}$$

Given a reference value r, where we also know the n'th root s, with the same exponent n as above.

$$r = f(s, n) = s^n \tag{2.2.3}$$

$$s = f(r, n) = \sqrt[n]{r} \tag{2.2.4}$$

We know that there is a correlation between the realtions x/r and y/s, the latter are the two bases of the two exponentiations, as follows:

$$y/s = \sqrt[n]{x/r} \equiv \sqrt[n]{x}/\sqrt[n]{r} \tag{2.2.5}$$

With this equation we can derive y as follows:

$$y = s * \sqrt[n]{x/r} \tag{2.2.6}$$

Hmm, we started with the question for $\sqrt{x}$ and end up with the question for $\sqrt{x/r}$, which is not really satisfying, because we still do not know the value for y, and we are left off with the original problem, to calculate a root y for a given number x.

This situation is very much similar to Achilles arriving at the point where the Tortoise started off, only to realize that the Tortoise has advanced, and he his going to enter the next iteration to get to the point where the Tortoise is right now, i.e. to repeat what he just did before.

But with $x > r$ and $r > 1$ we get a second value x/r, for which we want to know the root, which is smaller than the first value x.

Likewise, the distance to the Tortoise bacame smaller, the principal setup remains the same.

We call the second value x', and the root of it y', with

$$x' = x/r \tag{2.2.7}$$

$$y' = \sqrt[n]{x'} \tag{2.2.8}$$

and thus

$$y = s * y' \tag{2.2.9}$$

With this principle in mind we can repeat the calculation now for y' using the same reference r, and the according known s

$$x'' = x'/r \tag{2.2.10}$$

$$y' = s * \sqrt[n]{x''/r} \tag{2.2.11}$$

or

$$y' = s * y'' \tag{2.2.12}$$

$$y = s * s * y'' \tag{2.2.13}$$

If we continue like this, we get to a y value, which is very small, and comes close to the value 1.

$$y \approx s * s * s * ... * s * 1 \tag{2.2.14}$$

To generalize and putting alltogether in a formula we get the following recursion:

$$y_i = s * \sqrt[n]{x_i/r} \mid \begin{array}{ll} i = 0 & x_i = x_0 \\ i > 0 & x_i = x_{i-1}/r \\ x_i \geq r & \end{array} \tag{2.2.15}$$

With $x_i \geq r$ we define a break condition for the iteration or recursion to stop at some point, so we get c number of recursions or iterations. With the given number of iterations c we can make the following statements:

$$x \approx r^c \tag{2.2.16}$$

$$y \approx s^c \tag{2.2.17}$$

We can formulate one single step of iteration:

$$\frac{x^i}{r} = x^{i-1} \tag{2.2.18}$$

2.3 Referential versus Relational

There are essentially two types of constraints on syntactic operations: Those that pertain to the inherent properties of the affected element (referential) and those that are sensitive to the structural configuration in which it occurs (relational). - Google Search finding from linguistics - let's not got there.

For a given exponentation with a base y, exponent n, and the result of the exponentation function, the power x, we utilize a *reference* exponentation with a base s, the same exponent n as before, and the power r. For the given setup we specify the *realtionships* between the objects themselves $x -> y$, $y -> x$, $r -> s$, $s -> r$, $x -> r$, $r -> x$, $y -> s$, $s -> y$, but also identify and specify the relationship between relationships $x/r -> y/s$. The given diagram (figure 2) provides an overview for our setup.

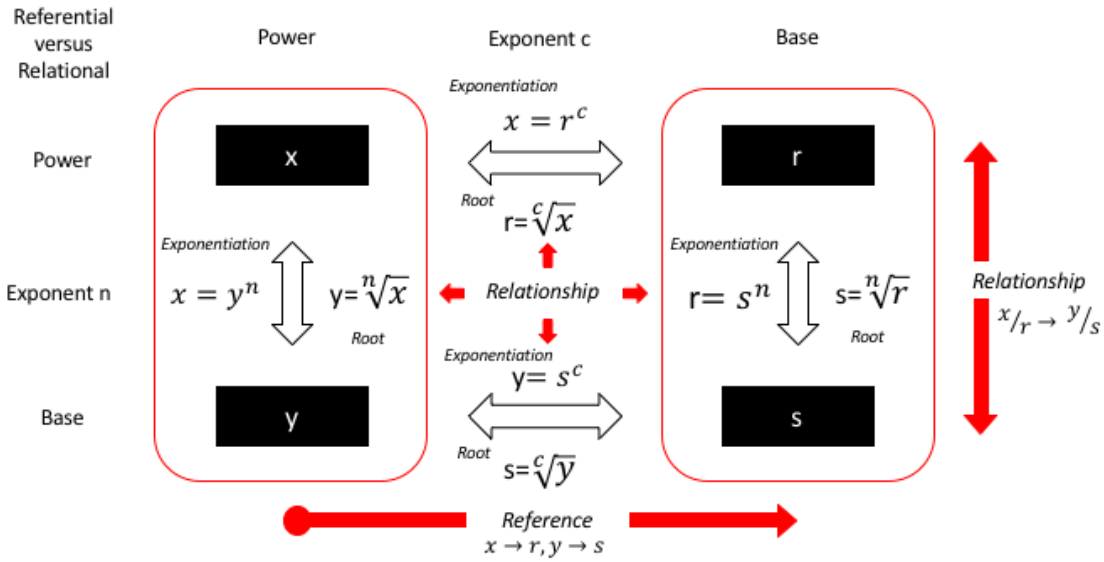

Figure 2: Referential versus Relational

With the understanding of the relationships and with the notion of a known reference exponentation, for which we know the base s, the exponent n, and the resulting power r, also knowing that exponentation calucaltion is easier than root calculation, we can resolve our original exponentation, for which we only know the power x, the exponent n, and for which we want to know the base/root y. We can derive y via the know reference exponentation and the particular realtionships between the stakeholders, as shown in this document.

The diagram shows exponentations on the left and the right side from bottom to top, exponentations on bottom and top, from right to left. The **power** r of the vertical, right side exponentation becomes the **base** r for the horizontal, top exponentation.

The power x is the result of both, vertical and horizontal, exponentations with bases r and y, exponents c and n. We can use this relationships to derive the unknown base y via the reference exponetation to traverse from x to r, from r to s (easy!), and finally from s to y.

The traversion from x to r is done by repeated - **iterative** - divisions of x_i/r until we reach fraction value 1 (or close to 1), where each iteration uses the output of the previous operation as input into the current operation - **recursive** -, by keeping count on the number of iterations, recursions in c. We later use the number of iterations c to go from s to y.

2.4 Error Function

Given that this approach is an approximation, we get an indication for a misfit, deviation or error, when looking at the last x_i in the iterations and the given delta δ to our reference r:

$$\delta = r - x_c \mid x_c = x_{min} \tag{2.4.1}$$

With $\delta = 0$ and $r = x_c$ we would have an ideal reference r with an according s and we could calculate the exact value y for a given x.

At least this is what I thought. Given the break condition of the loop $x_i \geq r$ and the implementation of the loop, the last x $x_c = x_{min}$ is equal 1 or close to value 1 by $x_i = x_{i-1}/r$. Thus δ shall be redefined as such:

$$\delta = x_c - 1 \mid x_c = x_{min} \tag{2.4.2}$$

With this new definition for δ and $\delta = 0$ with $x_c = 1$ we would have an ideal reference r with an according s, and we could calculate an exact y with $y = s^c$

As an additional measure of quality we can use the misfit of x to r^c:

$$\Delta = x - r^c \tag{2.4.3}$$

With our given approach of approximation we can be sure, that x is always greater than, or, for optimum solution, equal to r^c.

The number of iterations c, needed to get down to x_{min}, coming close to r, where c is also the exponent for a base r and s, with $r^c = x$ and $s^c = y$, can also be approximated with the use of the logarithm function:

$$c \approx \log_r(x) \tag{2.4.4}$$

2.5 Calibration

With $y \approx s^c$ we come more or less close to the value of y, the n'th root of x, which we were looking for. The quality of the result depends and varies with how close the latest and smallest x in the iterations of divisions comes to the choosen value r. If x_c is equal to r, then the latest division x_c/r yields value 1 and s^c will be the exact value for y.

In order to assess the quality of the exponent c, we can test and measure the deviation by

$$x_{test} = r^c \tag{2.5.1}$$

$$\Delta = x - r^c \tag{2.5.2}$$

We can now adjust c to get a minimum deviation Δ with the use of a given precision, tolerance τ (greek letter tau):

$$c_k = c, c_k = c_k + \tau \ while \ (r^{c_k} < x) \tag{2.5.3}$$

With the resulting, calibrated c_k we now can compute the proper value for y:

$$y = s^{c_k} \tag{2.5.4}$$

2.6 Reference Value

What shall be the value of s, and by that, the value of r?

It depends. As shown later in the document, using the Euler number e for reference base s shows good results in terms of low number of iterations c and thus low CPU execution time. The value for s must be decreased though, when the value for n becomes high, to avoid very high, or too high numbers for $r = s^n$. There are several example runs shown later in the document for different setups, showing all parameters and benchmarks.

2.7 Signature

To summarize, the following provides a signature, using algebraic, formal specification of a data-type, for the root calculation setup.

The signature for root approximation ROOT:

$$ROOT = (x, y, n, r, s, c, c_k, \tau, pow, div, cal, err) \tag{2.7.1}$$

where

x	is the input value, for which the n'th root we are looking for, with $x = y^n$
y	is the result of the root function $y = \sqrt[n]{x}$
n	is the exponent
r,s	are the reference values, with $r = s^n$
c	denotes the number of iterations, with $c \approx \log_r(x)$
c_k	is the calibrated c, with $x = r^{c_k}$
τ	is the tolerance, desired precision
pow	exponentation function
div	division function
cal	calibration function $c_k = cal(c, \tau, x, r)$
err	error function $err(x, r, c_k) = x - r^{c_k}$

3 Data Flow

The following diagram provides an overview for the data flow in the given calculation. The Diagram is inspired by Tom DeMarco and his Book on *Structured Analysis and System Specification* [DEMARCO]. I am using a slightly modified notation for data and flow, but maintain the principle outlined in the book.

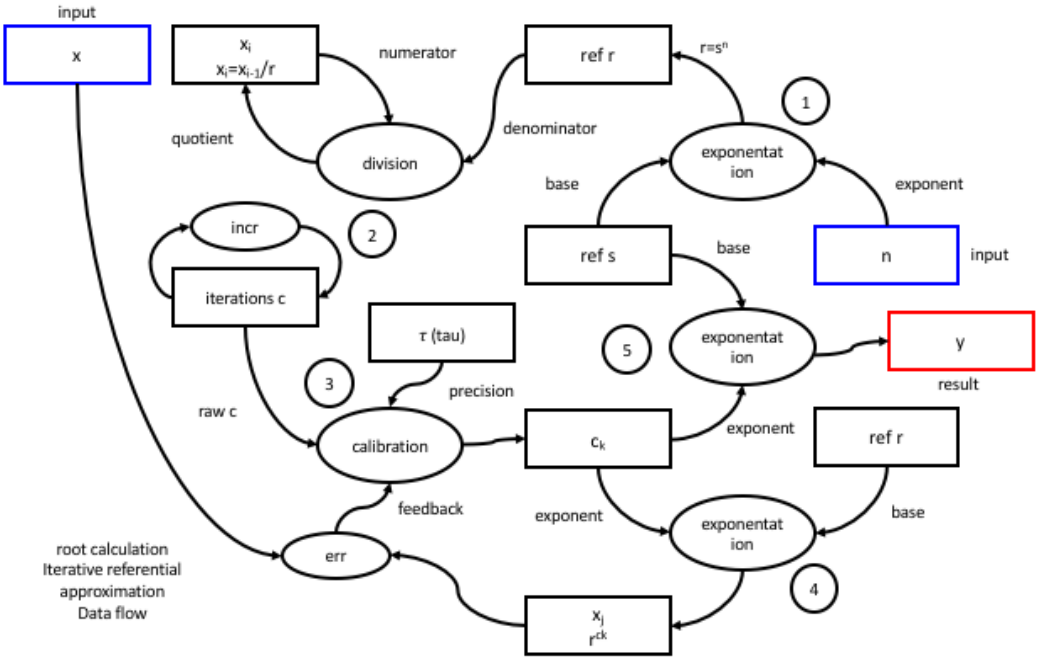

Figure 3: root value iterative appproximation data flow

For a given input value x we want to calculate the n'th root represented as output value y.

With the given input exponent n and a given reference base value s we calculate a reference value r by exponentiation $r = s^n$.

The resulting reference r acts as the divisor in an iterative, recursive process of dividing value x over and over again, where each division output is input dividend into the next division operation, until the result is smaller or equal to value 1. We are keeping count of the number of iterations/recursions and store the value in c.

In the next step we calibrate the value of c, as it has been described in the text above, by successivley adding τ (tau) to c, until we get a good enough result for x_{test}. The resulting calibrated c_k is now fit to be used in the next step to calculate y.

Finally we use the calibrated c and the original reference s in another exponentation to get the value for y, the n'th root of x.

4 The number of independent cycles

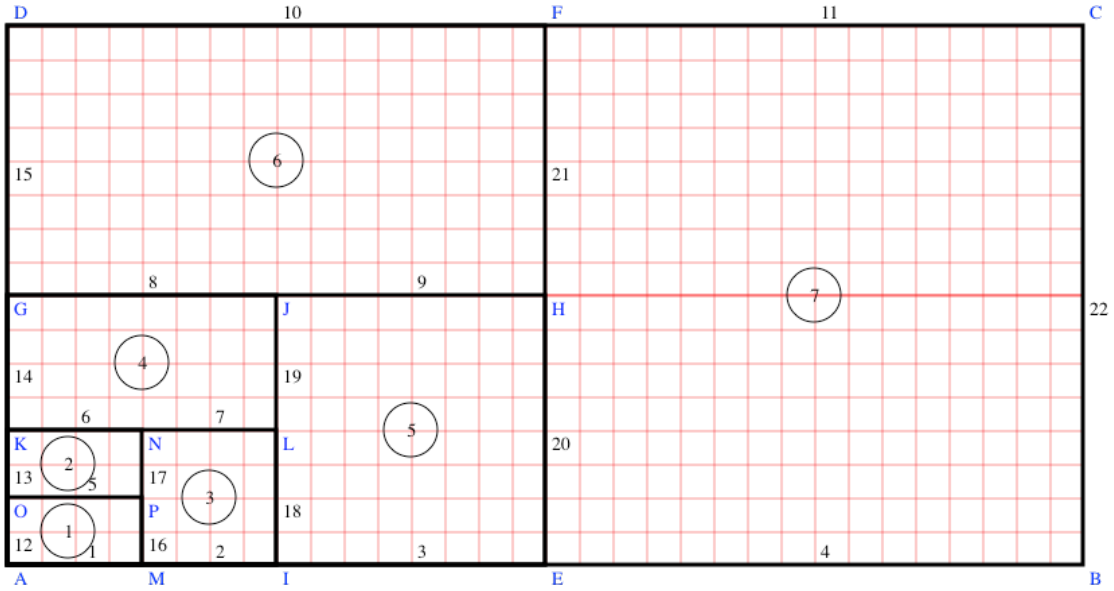

Figure 4: graphical representation of subsequent divisions by base/root 2 - reverse exponentation

AMPO	2^1	AMPO	cycle 1
AMNK	2^2	OPNK	cycle 2
AILK	2^3	MILN	cycle 3
AIJG	2^4	KLJG	cycle 4
AEHG	2^5	IEHJ	cycle 5
AEFD	2^6	GHFD	cycle 6
ABCD	2^7	EBCF	cycle 7

Given diagram (figure 4) shows a graphical representation of an exponentation for a base (= root) 2 and an exponent 7. In the reverese exponentation a given rectangle A-B-C-D is horizontally divided into two, the base. The resulting smaller left square A-E-F-D is vertically divided into two. The resulting lower rectangle A-E-H-G is horizontally divided into two, and so on. We do this seven (n, exponent) times, we start at 2^7 (A-B-C-D) and get down to 2^1 (A-M-P-O), the root.

This procedure of subsequent divisons with the base is also known as logarithm:

wiki: *In mathematics, the logarithm is the inverse operation to exponentiation.*

If we consider the given graphical representation for the logarithm function as a diagram, per the definition of Maxwell in his Treatise on Electricity and Magnetism, we get an interesting and unexpected correlation between the two so unrelated and disconnected topics, as shown in the following.

On first view and reading, completely unrelated to our given topic, James Clerk Maxwell in his *Treatise on Electricity & Magnetism* [MAXWELL], with reference to J. B. Listing, says

Let there be p points in space, and let l lines of any form be drawn joining these points, so that no two lines intersect each other, and no point is left isolated. We shall call a figure composed of lines in this way a Diagram.

Of these lines, $p-1$ are sufficient to join the p points so as to form a connected system. Every new line completes a loop or closed path, or, as we shall call it, a Cycle. The number of independent cycles in the diagram is therefore $k = l - p + 1$.

Let's try it: we have $l = 22$ lines, $p = 16$ points:

$$k = l - p + 1 = 22 - 16 + 1 = 7 \tag{4.0.1}$$

... The existence of cycles is called Cyclosis, and the number of cycles in a diagram is called Cyclomatic number.

Interestingly, the *number of independent cycles* is equivalent to the *exponent* in our graphical representation of an exponentation.

5 Zeno, Achilles, Tortoise

Zeno, in the beginning, never proposed a solution, but an approximation. He proposed a method, to come closer to the point, where the fast Achilles reaches the slow Tortoise. As such, it is the wrong method, to actually get to the point where they meet. Repeating the wrong method over and over again, does not make it right. The setup they call a paradoxon, which is absurd, this is an *absurdon*. Nonetheless, the method and the inclination for its application, to apply the wrong over and over again, with the hope, that repeating the wrong often enough will make it right, seems to persist in all fields dating back since the invention by Zeno until today.

6 Implementation

6.1 Implementation of Iterative Referential Root Approximation

The following paragraph provides the source code for the python implementation for the iterative, recursive, referential approximation for the root value y for a given input x with the exponent n.

6.1.1 Root Class Constructor

```
1
2       def __init__(self,n = 2,tau = 1e-3,s = math.e):
3         """
4         root functions implementation
5         - n ..... dimension, (default: 2)
6         - tau ... tolerance, precision, granularity (default: 0.001)
7         - s ..... reference value s
8         - r ..... reference value r = s^n
9         - x ..... y^n = r^ck
10        - y ..... n'th root y of input value x, y=s^ck
11        - c ..... iteration count, for one x, for one r
12        - ck .... calibrated c, D<tau
13        - D ..... Delta, misfit of r^c to x
14        """
15        self._n  = n
16        self._tau  = tau
17        self._s  = s
18        self._c = 0
19        self._s = s
20        self._r = self._s**self._n
```

6.1.2 Helper - Get Decimal Places

```
1
2       def _get_decimals(self,tau):
3         """
4         get number of decimal places based on tau
5         e.g. tau = 0.001 (1e-3): decimals = 3
6         """
7         return int(math.log10(1/tau))
```

6.1.3 Helper - Calculate C

```
1
2       def _calc_c(self,x,r):
3         """
4         iteration for x for a given reference r
5         repeated division of x with r
6         output of one division becomes input for next division
7         until x becomes equal r or smaller
8         return c, the number of iterations
9         """
10        _x = x
11        _c = 0
12        while (_x >= r):
13          _x = _x/r
14          _c += 1
15        return _c
```

6.1.4 Calibration

```
1
2       def _calibration(self,c,x,r):
3         """
4         calibrate c for x and r
5         return ck ... calibrated c
6         x' = r^ck
7         """
8         _tau=self._tau/1000
9         _decimals = self._get_decimals(_tau)
10        _ck = c
11        while (_decimals>0):
```

```
12          _incr = _tau*(10**(_decimals-1))
13          while (x>(r**_ck)):
14            _ck += _incr
15          _ck -= _incr
16          _decimals -= 1
17        _ck = round(_ck,self._get_decimals(_tau))
18        return _ck
```

6.1.5 Helper - Error Function

```
1
2       def _err(self,x,y,n):
3         """
4         error function, calculate misfit
5         """
6         return (x - y**n)
```

6.1.6 Helper - Calculate Y

```
1
2       def _calc_y(self,s,ck):
3         """
4         y=s^ck
5         """
6         _y = s**ck
7         _y = round(_y,self._get_decimals(self._tau))
8         return _y
```

6.1.7 Root Y

The actual function to calculate the root value y in the given python implementation is wrapped by a wrapper function, which does some time measurements to calculate the execution time of the class method. The wrapper function is y, the original y method is given in *_wrapped_y* as shown below.

```
1
2       def y(self,x):
3           _start_time = time.time()
4           result = self._wrapped_y(x)
5           _end_time = time.time()
6           self._execution_time_y = _end_time - _start_time
7           return result
8
9       def get_execution_time_y(self):
10        return self._execution_time_y
11
12      def _wrapped_y(self,x):
13        """
14        iterative, relational root approximation function
15        return n'th root y for given x
16        """
17        self._c = self._calc_c(x,self._r)
18        self._ck = self._calibration(self._c,x,self._r)
19        _y = self._calc_y(self._s,self._ck)
20        self._D = self._err(x,_y,self._n)
21        return _y
```

7 Application

7.1 Application of the Iterative Referential Root Approximation

The following provides the python code to actually instantiate the root class and invoke the iterative, recursive, referential approximation function

```
1
2       def root_function(self):
3           """
4           application of the root function
5           """
6           root = math_root_t_root.class_t_root(n=2)
7           y = root.y(81)
```

The table below shows some tests for different input values x, n and the result y and the parameters and benchmarks per run, e.g. c_k indicating the number of iterations, time showing the CPU execution time in seconds.

Legend

size	...	bit width of x
n	...	exponent, $x = y^n$, $y = \sqrt[n]{x}$
x	...	power
y	...	base
c_k	...	calibrated c, $r = s^{c_k}$
r	...	reference power
τ (tau)	...	tolerance, precision
err	...	misfit, err $= x - y^n < \tau$
time	...	execution time in seconds

size	n	x	y	c_k	r	s	tau	err	time
8	2	2	1.414214	0.3466	7.3891	2.7183	1e-9	0.0000	0.000056
8	2	4	2.000000	0.6931	7.3891	2.7183	1e-3	0.0000	0.000034
8	2	5	2.236000	0.8047	7.3891	2.7183	1e-3	0.0003	0.000033
8	2	64	8.000000	2.0794	7.3891	2.7183	1e-3	0.0000	0.000030
8	2	81	9.000000	2.1972	7.3891	2.7183	1e-1	0.0000	0.000025
8	2	128	11.313700	2.4260	7.3891	2.7183	1e-4	0.0002	0.000032
8	2	255	15.968700	2.7706	7.3891	2.7183	1e-4	0.0006	0.000036
8	3	3	1.442200	0.3662	20.0855	2.7183	1e-4	0.0003	0.000048
8	3	4	1.587400	0.4621	20.0855	2.7183	1e-4	0.0000	0.000033
8	3	125	5.000000	1.6094	20.0855	2.7183	1e-3	0.0000	0.000034
8	3	128	5.039680	1.6173	20.0855	2.7183	1e-5	0.0003	0.000037
8	3	255	6.341330	1.8471	20.0855	2.7183	1e-5	-0.0005	0.000042
8	256	4	1.005430	0.5442	12.7724	1.0100	1e-6	-0.0001	0.000024
8	256	255	1.021882	2.1754	12.7724	1.0100	1e-8	-0.0003	0.000045
8	1024	2	1.000677	0.0680	26612.5661	1.0100	1e-8	0.0000	0.000030
8	1024	255	1.005426	0.5438	26612.5661	1.0100	1e-8	-0.0005	0.000063
16	2	256	16.000000	2.7726	7.3891	2.7183	1e-6	0.0000	0.000050
16	2	32768	181.019336	5.1986	7.3891	2.7183	1e-6	-0.0000	0.000043
16	2	65535	255.998047	5.5452	7.3891	2.7183	1e-6	-0.0001	0.000044
32	2	65536	256.000000	5.5452	7.3891	2.7183	1e-6	0.0000	0.000044
32	2	250000	500.000000	6.2146	7.3891	2.7183	1e-2	0.0000	0.000026
32	2	2**31-3	46340.949979	10.7438	7.3891	2.7183	1e-10	0.0003	0.000064
32	2	2**31-2	46340.949990	10.7438	7.3891	2.7183	1e-10	0.0004	0.000058
32	2	2**31-1	46340.950001	10.7438	7.3891	2.7183	1e-10	0.0001	0.000049
32	2	2**32-3	65535.999977	11.0904	7.3891	2.7183	1e-10	0.0007	0.000031
32	2	2**32-2	65535.999985	11.0904	7.3891	2.7183	1e-10	-0.0000	0.000030
32	2	2**32-1	65535.999992	11.0904	7.3891	2.7183	1e-10	0.0001	0.000030
64	2	2**32	65536.000000	11.0904	7.3891	2.7183	1e-10	0.0002	0.000031
64	2	2**33	92681.900024	11.4369	7.3891	2.7183	1e-10	-0.0000	0.000031
64	2	2**34	131072.000000	11.7835	7.3891	2.7183	1e-10	0.0024	0.000029
64	2	2**46	8388608.000000	15.9424	7.3891	2.7183	1e-12	0.0781	0.000038
64	2	2**46	8388608.000000	9.4158	29.5562	5.4366	1e-12	0.2812	0.000035
64	2	2**48	16777216.000000	9.8252	29.5562	5.4366	1e-12	1.0000	0.000037
64	2	2**50	33554432.000000	10.2346	29.5562	5.4366	1e-12	3.7500	0.000037
64	2	2**55	189812531.248503	11.2581	29.5562	5.4366	1e-12	200.0000	0.000038
64	2	2**55	189812531.248503	9.0829	66.5015	8.1548	1e-12	96.0000	0.000047
64	2	2**57	379625062.497006	3.4247	102400.0000	320.0000	1e-12	-64.0000	0.000038
64	2	2**60	1073741823.999998	3.6049	102400.0000	320.0000	1e-12	3840.0000	0.000038
64	2	2**61	1518500249.988024	3.6650	102400.0000	320.0000	1e-12	3328.0000	0.000035
64	2	2**62	2147483647.999995	3.7251	102400.0000	320.0000	1e-12	20480.0000	0.000036
64	2	2**63	3037000499.976046	3.7852	102400.0000	320.0000	1e-12	21504.0000	0.000034
64	2	2**64-1	4294967295.999989	3.8453	102400.0000	320.0000	1e-12	98304.0000	0.000039

We get good results until the numbers get very high. With x values above 2^{46} we start getting errors greater 1, i.e. the results are no longer precise enough. Up to this point the precision was kept low by decreasing tau. The given python implementation becomes unresponsive with tau smaller 10^{-12}.

We get good benchmarks with c_k most times kept below 12, i.e. a low umber of iterations to do the calulation, for a chosen $s = e(eulernumber)$. The low number of iterations is reflected by the small CPU execution times in the order of tens of microseconds.

If exponent n increases, the reference s must be kept small, for not getting "*number too big*" exceptions for $r = s^n$.

8 Bibliography

DEMARCO *Structured Analysis and System Specification*, Tom DeMarco, Yourdon Press 1979,1978.

MAXWELL *Treatise on Electricity and Magnetism*, James Clerk Maxwell

WIKIROOTS Methods of computing square roots

Index