



Generation of n Digits of the Golden Ratio (Phi)



Kanhar Munshi

3 Mar 2010 [CPOL](#)

A heavily optimized program written in Scheme that generates accurately the digits of the golden Ratio

Project Objective

The objective of this project is to implement and evaluate an algorithm employing the concept of infinite streams in Scheme in order to evaluate precisely, the n^{th} digit of an infinite series of numbers.

The series we specifically plan to evaluate is the Golden Ratio or the limit of the ratio of the Fibonacci numbers F_{n+1} over F_n as n tends to infinity.

It is important that our solution does not approximate this value.

We also plan to compare the correctness of our solution by using a more precise mathematical approach to determine the value of the Golden Ratio, exploiting an intrinsic property of the series whereby, the Limit of the Golden Ratio as the number of terms tends to infinity is represented by the formula:

$$\text{Golden Ratio Value} = \frac{(1 + \sqrt{5})}{2}$$

Introduction

The [Golden Ratio](#), number that arises naturally in mathematics, is one of the most fascinating numbers in the discipline, and because of its occurrence in the natural world, where it is observed in the angular geometry of seeds in sunflowers, cone flowers and pine cones, and numerous other places. It is also used heavily in architecture, and there is now some proof that starting from the Greeks, the ratio was used in some form or the other in most architectural works after.

There of course exist many ways to calculate this ratio (which incidentally is an irrational number) and so is represented by the convergence of either an infinite series or by formulae involving irrational numbers. The ratio is however real, and does not include an imaginary component.

Below are a few ways to evaluate this ratio.

```

0.14285
-----
7| 10
   7
  --
  30
  28
  --
  20
  14
  --
  60
  56
  --
  40
  35

```

Implementation

We zeroed in on two methods to approach the problem of generating an infinite Stream that represents the Golden Ratio.

The first method involves computing the ratio of the $(N+1)^{\text{th}}$ Fibonacci number to the N^{th} Fibonacci number.

The second method involves computing the ratio $(1 + \sqrt{5})/2$. The focus in this method is the computation of the Square Root of Root 5 which we represent by an infinite Stream. Scheme allows for delayed evaluation of expressions through the use of keywords `delay`, and then for the evaluation of the expression, through the use of keyword `force`.

a) Cons-Stream

Streams, essentially comprise of one already evaluated value as the first value in the stream, and the promise of another value that would be returned if the stream would be forced to return another value.

Two things happen above:

1. It adds `cons-stream` as a new syntax rule for use again later in the environment (as a syntax)
2. It sets up the definition of `cons-stream` to imply a pair of objects, where the first object represents a tangible value that can be retrieved immediately while the second object is the promise to return another value if the stream is asked to of course do so. (This is essentially the implication of the keyword `delay`).

a) Fibonacci Method of Approximation

```

;Infinite Fibonacci Stream
(define fibs
  (cons-stream 0 (cons-stream 1 (add (tail fibs) fibs))))

(define goldenRatioStream
  (lambda (n)
    (/ (getn fibs (+ n 1)) (getn fibs n))))

```

b) Square Root Method of Approximation

The algorithm we used to compute the Square Root of 5 was the Divide and Average algorithm known to the [Babylonians](#) in around 1700 BC.

The following is an extract of this algorithm from www.mathpath.org. The initial guess is arbitrary but must be nonZero, and while the example given below sets A = 2, I set to A = 1 for simplicity.

Points of Interest

I implemented a custom division algorithm that generates an infinite division loop to divide two non divisible numbers that can be reused across many different applications. The code is astonishingly simple for a program that can accurately print a mathematical number to an infinite length to any precision.

```
;Custom Scheme Fraction to Decimal Conversion
(define gr_fib_exact
  (lambda (n precision)
    (divide (getn fibs (+ n 1)) (getn fibs n) precision )))
```

Conclusion

We conclude by looking at our data that depicts increasing terms of the Golden Ratio as evaluated, that a greater number of digits lock into certain places as the series increases, and do not change their value as the number of the terms in the series increases.

On further inspection, we notice that if a digit **d** (say) at index **i** (**i** places from the decimal reading left to right) occurs more than once consecutively in two terms of the Golden Ratio (with increasing **n**) then it locks into that place, and increasing terms of the series does not change the value **d** at index **i**.

By referring to Appendix C, we have proved theoretically that a given **d** where **d** refers to a digit **i** places from the decimal, is completely accurate to the correct value of the golden ratio at that index (**i** places from the decimal) if the associated **N** for that value of **d** as given by the formula is greater than the value of **n** used to evaluate the golden ratio.

$$n = 5/2(d + 1)$$

Code Listing

Call.SCM

```
;This calls the EXACT fibonacci function n Times from (1 to n) with a Precision of 300
(loop gr_fib_exact 10 300)

;This calls the EXACT square root algorithm function n Times from (1 to n)
with a Precision of 10
;This function is much slower and a n value > 15 took more than 1 minute
on my 2 GB RAM machine
(loop gr_square_inexact4 10 10)
```

Golden.SCM

```

;Evaluate Golden Ratio using Different Methods
(require (lib "trace.ss"))
;Loads Relevant Stream Functions
(require "stream.scm")

;Square Root Functions
(define (average x y) (/ (+ x y) 2))

(define (sqrt-improve guess x)
  (average guess (/ x guess)))

;Note the 1.0 (this is inexact since Decimals are forced in the recursion)
(define (sqrt1 x)
  (define guesses (cons-stream 1.0 (applyproc (lambda (guess)
    (sqrt-improve guess x)) guesses))
  )
  guesses
)
;Note that this is Exact ( since only Fractions are used in Recursion)
(define (sqrt2 x)
  (define guesses (cons-stream 1 (applyproc (lambda (guess)
    (sqrt-improve guess x)) guesses))
  )
  guesses
)

;Four Ways of Using Recursion varying by Exact |
  Inexact Recursion and Exact | Inexact Calling towards the End
(define gr_square_inexact1
  (lambda (n)
    ( exact->inexact (/ (+ 1 (getn (sqrt1 5) n)) 2))))

(define gr_square_inexact2
  (lambda (n)
    ( / (+ 1 (getn (sqrt2 5) n)) 2)))

(define gr_square_inexact3
  (lambda (n)
    ( exact->inexact (/ (+ 1 (getn (sqrt2 5) n)) 2))))

(define gr_square_inexact4
  (lambda (n precision)
    (
      letrec((x (/ (+ 1 (getn (sqrt2 5) n)) 2)))

      (divide (numerator x) (denominator x) precision))))

;Infinite Fibonacci Stream
(define fibs
  (cons-stream 0 (cons-stream 1 (add (tail fibs) fibs))))

(define goldenRatioStream
  (lambda (n)
    (/ (getn fibs (+ n 1)) (getn fibs n))))

;Native Scheme Fraction to Decimal Conversion
(define gr_fib_inexact
  (lambda (n)
    ( exact->inexact (/ (getn fibs (+ n 1)) (getn fibs n)) )))

;Custom Scheme Fraction to Decimal Conversion
(define gr_fib_exact
  (lambda (n precision)

```

```

    (divide (getn fibs (+ n 1)) (getn fibs n) precision )))

(define ge2
  (lambda (n)
    ( exact->inexact (/ (+ 1 (getn (sqrt 5) n)) 2) )))

(define ge2
  (lambda (n)
    (let ((x (/ (+ 1 (getn (sqrt 5) n)) 2)))
      (divide (numerator x) (denominator x) n)
      )))

(define ratio
  (lambda(term precision)
    (display 'Golden_Ratio_using_Fibonacci_at_N= )
    (display term)
    (display " ")
    (display 'Precision=)
    (display precision)
    (display " ")
    (display 'is )
    (display " ")
    (newline)
    (display (ge1 term precision))
    (newline)
    (display 'Golden_Ratio_using_Square_Root_Method )
    (display " ")
    (display 'Precision=)
    (display precision)
    (display " ")
    (display 'is )
    (display " ")
    (newline)
    (display (list (ge2 term ) ) )
    )))

(define integers
  (lambda(n)
    (cons-stream n (integers (+ n 1)))))

;Evaluates a Function from 1 to N, spaced by a NewLine
(define (loop proc n precision)
  (call-with-current-continuation
    (lambda (stop)
      (define (inner-loop proc i)
        (cond ((> i n) (stop 'done))
              (else
               (display (proc i precision))
               (newline)
               (inner-loop proc (+ i 1))
               )))
      (inner-loop proc 1))))

```

List.SCM ;A Tutorial In Lazy Evaluation in Scheme

```

(delay (+ 5 6))
;Returns; #<struct:promise>
;(force (delay ( + 5 6)))
;11

;Call Promise
(let

```

```

    (
      (delayed (delay (+ 5 6)) )
    )
    (force delayed)
  )
;Returns 11

```

;These are the Header Files, sort of the Core Functions that Are Used Later on.

```

(define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream x y)
     (cons x (delay y)))))

```

;Just the First element of the Stream

```

(define head
  (lambda(stream)
    (car stream)))

```

;Forces Evaluation of the Cdr of the Stream

```

(define tail
  (lambda(stream)
    (force(cdr stream)))))

```

;Adds two Streams (in Reality it only adds the First Element of Both Streams, but Promises to add the Rest of the Stream later, by delaying it)

```

(define add
  (lambda (s1 s2)
    (cons-stream (+ (head s1) (head s2) )      (add (tail s1) (tail s2)))))

```

;Gets the First N Characters of any Stream, by recursively Forcing evaluation of the Stream one at a time.

```

(define get
  (lambda(stream n)
    (cond
      ((= n 0) '())
      (else (cons (head stream) (get (tail stream) (- n 1)))))))

```

;Gets the Nth Character of any Stream, by discarding all characters until it reaches the Nth character. N here serves as a counter.

```

(define getn
  (lambda(stream n)
    (cond
      ((= n 0) (head stream))
      (else (getn (tail stream) (- n 1))))))

```

;Define a List of Infinite Ones

```

(define ones (cons-stream 1 ones))

```

;Gets Integers from n Onwards (Example 1)

```

(define integers
  (lambda(n)
    (cons-stream n (integers (+ n 1)))))

```

;Prime Nos (Example 2)

```

(define (divisible? x y) (= (remainder x y) 0))

```

```

(define sieve
  (lambda(stream)
    (cons-stream (head stream) (sieve (filter (lambda(x)
      (not (divisible? x (head stream)))) (tail stream))))))

```

```

(define filter
  (lambda(p lst)
    (cond

```

```
((null? lst) '())
((p (head lst)) (cons-stream (head lst) (filter p (tail lst))))
(else (filter p (tail lst))))))

(define primes (sieve (integers 2))) ;;Prime Seed Value of 2

(get primes 5)
;(2 3 5 7 11)

;Fibonacci Series (Example 3)
(define fibs
  (cons-stream 0 (cons-stream 1 (add (tail fibs) fibs))))

(define golden
  (lambda (n)
    (/ (getn fibs (+ n 1)) (getn fibs n))))
```

History

- 3rd March, 2010: Initial post

License


This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author



Kanhar Munshi



Software Developer (Senior) Microsoft
United States 

****Update**:** I now work at Microsoft. See more here: <https://kanhar.github.io/>

***Update*:** I now work as a software developer at Investment Bank, working on mostly proprietary stuff.

Kanhar Munshi currently works as a consultant for the New York City Department of Health and Environmental Surveillance building, deploying and maintaining ASP.NET web applications, while also serving in the role of an assistant database administrator.

He is currently involved in the building of a Large Scale implementation of a Normalized Data Store, Data Warehouse for the Department of Health.

His interests, include C#, running, table tennis, triathlons and going on long day hikes.

Comments and Discussions

 **0 messages** have been posted for this article Visit <https://www.codeproject.com/Articles/60374/Generation-of-n-Digits-of-the-Golden-Ratio-Phi> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Article Copyright 2010 by Kanhar Munshi
Everything else Copyright © [CodeProject](#), 1999-2021

Web04 2.8.20210820.1