

Experiments in Matlab: Iteration

John Burkardt
burkardt@vt.edu

Advanced Research Computing
Virginia Tech

.....

10:00-11:00, 01 March 2017

.....

Slides available at:

https://secure.hosting.vt.edu/www.arc.vt.edu/class_note/

Overview: The Intended Audience

This presentation is an informal introduction to iteration and Matlab.

It shows how the idea of iteration is used in computing, and how Matlab can implement iteration using `for` and `while` loops.

*If you are familiar with Matlab loops, this presentation **is not** for you!*

If you have a copy of Matlab or Octave, you can follow along on the programming examples. Otherwise, you can watch the demonstrations on the screen.

For information on obtaining a copy of Matlab:

<http://www2.ita.vt.edu/software/student/products/mathworks/matlab/>

Octave is available at <https://www.gnu.org/software/octave/>

Overview: Book and Programs by Cleve Moler

Cleve Moler, the author of Matlab, has published an electronic textbook of 20 chapters, called “**Experiments in Matlab**”.

Each chapter is intended to investigate a simple, interesting topic, while demonstrating certain features of Matlab.

Any of the chapters. or the whole ebook, may be downloaded from <https://www.mathworks.com/moler/exm/chapters.html>

A toolbox and interactive application can be downloaded as well.



















We won't need those tools for this simple presentation.

Overview: Chapters of Experiments in Matlab

- 0 Preface
- 1 Iteration
- 2 Fibonacci Numbers
- 3 Calendars and Clocks
- 4 Matrices
- 5 Linear Equations
- 6 Fractal Fern
- 7 Google PageRank
- 8 Exponential Function
- 9 T Puzzle
- 10 Magic Squares
- 11 TicTacToe Magic
- 12 Game of Life
- 13 Mandelbrot Set
- 14 Sudoku
- 15 Ordinary Differential Equations
- 16 Predator-Prey Model
- 17 Orbits
- 18 Shallow Water Equations
- 19 Morse Code
- 20 Music

Overview: GUI for Experiments in Matlab

The Toolbox allows you to interactively explore the topics:

 fern	 biorhythm	 clockex	 lifex	 rabbits
 pagerank	 wiggle	 t_puzzle	 tictactoe	Alb backslash
 predprey	 mandelbrot	 durerperm	 waterwave	 expgui
 sudoku	 orbits	 morse	 pianoex	exmlogo helpwin exm helpwin exmgui close

Overview: Iteration

The definition of insanity is doing the same thing over and over and expecting different results.."

– (The most overused cliché, according to Salon magazine.)

Iteration: *in the context of computer programming, is a process wherein a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met.*

–Techopedia

Reasonable conclusion: **Computer programming = Insanity?**

Golden: Iteration Plan

Let's try a very simple example of iteration, which will end up computing a number called the Golden Ratio. We will do the same thing over and over, and **I do expect different results!**

Start up Matlab, and then type

- $x = \textit{put some number here}$
- $x = \text{sqrt} (1 + x)$
- Use the **up-arrow** key to recall this command;
- Use the **return** key to execute it again;
- At first, you should get a different answer each time. Then...?

Golden: Iteration Results

```
>> x = 123.456
```

```
x = 123.46
```

```
>> x = sqrt ( x + 1 )
```

```
x = 11.156
```

```
>> x = sqrt ( x + 1 )
```

```
x = 3.4865
```

```
>> x = sqrt ( x + 1 )
```

```
x = 2.1181
```

```
>> x = sqrt ( x + 1 )
```

```
x = 1.7658
```

```
>> x = sqrt ( x + 1 )
```

```
x = 1.6631
```

```
>> x = sqrt ( x + 1 )
```

```
x = 1.6319
```

```
>> x = sqrt ( x + 1 )
```

```
x = 1.6223
```

```
>> x = sqrt ( x + 1 )
```

```
x = 1.6194
```

```
>> x = sqrt ( x + 1 )
```

```
x = 1.6184
```

```
>> x = sqrt ( x + 1 )
```

```
x = 1.6182
```

```
>> x = sqrt ( x + 1 )
```

```
x = 1.6181
```

```
>> x = sqrt ( x + 1 )
```

```
x = 1.6180
```

```
>> x = sqrt ( x + 1 )
```

```
x = 1.6180
```


Golden: Comments

No matter what your starting number, your iteration should tend to go to the value 1.6180, known as the golden ratio, or ϕ or “phi” ..

You should try this with several different starting values; the results usually settle down in about 10 steps.

If you know a little math, it will surprise you to know that you can even start with x being a negative value. Your intermediate results will now be complex numbers, but even then, they will gradually approach 1.6180!

Golden: Mathematics Can Help Us

What does it mean that our results have stopped changing?

In computing, the equals sign indicates an **assignment statement**, such as

$$a = 7$$

$$b = c + 2$$

$$x = \text{sqrt} (1 + x)$$

That is, the number, variable or formula on the right hand side represents a value. Put that value into the variable on the left. That's why, in computing, it can make sense to say:

$$x = x + 1 \quad \leftarrow \text{replace } x \text{ by the value } x + 1$$

In mathematics, this last statement would be nonsense. In mathematics, the equal sign asserts that the left and right quantities (already, and forever) have the same values.

Golden: Mathematics Can Help Us

But our Matlab statement seems to become a Mathematics statement!

Instead of looking at $x = \text{sqrt}(1 + x)$ as a Matlab assignment statement, we can now think of it as mathematics statement, because now x does seem to already be the same as the right hand side.

But if we can use mathematics, we can actually work something out:

$$x = \sqrt{1 + x}$$

$$x^2 = 1 + x \quad (\text{square both sides})$$

$$x^2 - x - 1 = 0 \quad (\text{move everything to left})$$

$$x = \frac{1 \pm \sqrt{5}}{2} \quad (\text{quadratic formula})$$

for which the values are **1.618033988749895** and -0.618033988749895.

Golden: Matlab Can Show More Digits

Usually, Matlab only displays about 5 digits of the result; however, it is really working with 16 decimal digits. You can ask Matlab to show you the gory details by typing

```
format long
```

Now when you carry out an iteration, it takes longer for all the digits to match...because there are more digits. If you don't want to see the full details anymore, you can go back to the shorter output form:

```
format short
```

Golden: The Same Statements, Different Data

The line `x = 123.46` is an assignment statement. On the left hand side is a **name**, `x`, and on the right is a value, or a formula, or the name of another variable, whose value is to be associated with `x`.

Once the name `x` has a value, it can be used in formulas, where its name will be replaced by the value it holds.

At any time, a new value can be stored into `x` using an assignment.

So `x = sqrt (x + 1)` is interpreted as assuming that `x` already has a value, to be added to 1, and then the square root taken. The result replaces the old value in `x`.

The result of our computation changes on every step, even though we are doing the exact same operation. The point is, **we're doing the same operation, but to data that changes.**

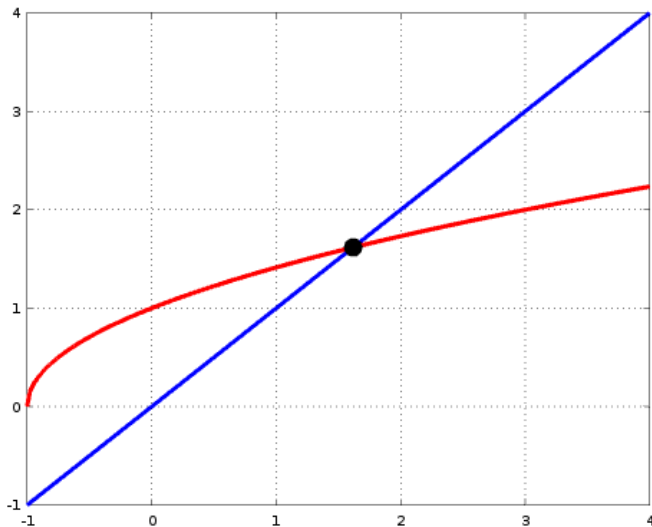
Golden: Insight from Matlab Graphics

Another way to think about $x = \sqrt{1+x}$ is to think of the right and left hand side as formulas for any value of x . If we plot the two formulas $y = x$ and $y = \sqrt{1+x}$, then any crossing point is a solution.

```
phi = ( 1 + sqrt ( 5 ) ) / 2;      <-- We know the solution

x = -1.0 : 0.02 : 4.0;            <-- Check x from -1 to 4
y1 = x;                            <-- Left hand formula
y2 = sqrt ( 1 + x );              <-- Right hand formula
plot ( x, y1, '-', x, y2, '-', phi, phi, 'o' )
```

Golden: Formulas Agree where Curves Cross



Golden: More Matlab Facts

A simple Matlab plot command has the form `plot(x,y)`, where `x` and `y` are *vectors*, that is, lists of values.

The command `plot(x,y,'-')` connects data values to form a curve. The command `plot(x,y,'.')` simply displays them as dots.

We can plot several sets of data with one plot command.

We want to plot our formulas for `x` values between -1 and 4. The Matlab command `x=-1:0.02:4` creates a list of values in that range, with a spacing of 0.02.

The commands `y1=x` and `y2=sqrt(1+x)` evaluate the formulas at each `x` value, storing the results in `y1` and `y2`.

Type `help plot` for more information about the Matlab plot command.

Automation: Iterating **FOR** So Many Steps

Rather than hit the up-arrow and Return keys 10 times to carry out our iteration, we can use Matlab's **for** loop. To repeat a single command 10 times, we write:

```
for i = 1 : 10
    command
end
```

So we can save a lot of typing by writing:

```
x = 123.456
for i = 1 : 10
    x = sqrt ( 1 + x )
end
```

Automation: Iteration Results

```
>> x=123.456
x = 123.46
>> for i = 1 :10
    x = sqrt(1+x)
end
    <-- Matlab waits til the end statement
        before executing the loop.

x = 11.156
x = 3.4865
x = 2.1181
x = 1.7658
x = 1.6631
x = 1.6319
x = 1.6223
x = 1.6194
x = 1.6184
x = 1.6182
    <-- Looks like 10 steps was not enough.
```

Automation: Iterating **WHILE** a Condition Holds

If we want our iteration to settle down, we could simply run it for more steps, as in `for i = 1 : 20`.

If we know that we should keep going as long as x is not the same as $\text{sqrt}(1+x)$, then Matlab allows us to write an iteration loop that repeats exactly that way, using the `while` statement, controlled by a **condition**.

```
while ( some condition is true )  
    one or more commands to be repeated  
end
```

Automation: Expressing a Condition

A condition is usually a comparison between two numbers or formulas.
The most common conditions have the form

<code>(x == y)</code>	<code>x</code> is exactly equal to <code>y</code> ;
<code>(x ~= y)</code>	not equal to <code>y</code> ;
<code>(x < y)</code>	less than <code>y</code> ;
<code>(x <= y)</code>	less than or equal to <code>y</code> ;
<code>(x > y)</code>	greater than <code>y</code>
<code>(x >= y)</code>	greater than or equal to <code>y</code> .

For our iteration, the condition could be `(x ~= sqrt (1 + x))`

Automation: A While Loop Iteration

To run our iteration using a while loop, we write

```
x = 123.456
while ( x ~= sqrt ( 1 + x ) )
    x = sqrt ( 1 + x )
end
```

It may seem like this loop would take just one step, and then stop. But remember, in computing, the equals sign is an assignment statement.

We're not saying that x equals $\sqrt{1 + x}$; we're replacing x by this value. Now that we have a new value of x , the value of $\sqrt{1 + x}$ also changes. And so x and $\sqrt{1 + x}$ are probably still not equal. In fact, it is really a little bit surprising that these two values do gradually get extremely close, and, on a computer, actually equal.

Automation: Suppress Output with Semicolon

Now that we expect our loop to run just long enough to get the right result, we probably don't need to see all the intermediate results. You can stop Matlab from printing the result of a statement by putting a **semicolon** on the end.

Of course, then we don't see anything from the loop. To see the final values of `x` and `sqrt(1+x)`, we can list them at the end.

```
x = 123.456
while ( x ~= sqrt ( 1 + x ) )
    x = sqrt ( 1 + x );
end
x
sqrt ( 1 + x )
```

Automation: Iterate WHILE a Tolerance Not Met

Instead of iterating until the two formulas are exactly equal, we can stop when they are close. The value `eps(x)` represents the accuracy of numbers near `x`.

```
x = 123.456
while ( abs ( x - sqrt ( 1 + x ) ) > eps ( x ) )
    x = sqrt ( 1 + x );
end
x
eps ( x )
```

The `abs()` function takes the absolute value.

Depending on how many digits of accuracy we want, we could speed up the iteration by using a tolerance of `1000 * eps (x)`, for example.

Square Roots: A Second Iteration Example

In our next example of an iteration, we don't get the same final result each time. Instead, we have been asked to find a number x which is the square root of the number y . It turns out that an iteration will find the answer for us.

Start up Matlab, and then type

- y = *put the number whose square root is desired;*
- x = *put any nonnegative guess here*
- $x = (x + y/x) / 2$
- Use the **up-arrow** key to recall this command;
- Use the **return** key to execute it again;
- Repeat as long as x keeps changing.
- When done, type $x*x$ or x^2 and compare to y

Square Roots: Iteration Using Up Arrow

```
>> y = 123.45
y = 123.45
>> x = 1
x = 1
>> x = (x + y/x) / 2    (We won't print the repetitions)
x = 62.225
x = 32.104
x = 17.975
x = 12.421
x = 11.180
x = 11.111
x = 11.111
x = 11.111
>> x*x
ans = 123.45
```

Square Roots: Iteration Using WHILE

```
y = 123.45;
x = 1;
while ( abs ( x * x - y ) > eps ( y ) )
    x = ( x + y / x ) / 2;
end
%
% When loop is done, print X, and error Y-X*X
%
x
y - x * x
```

Comments start with the **percent** sign.

Hailstones: Iteration Example 3

The **hailstone** game starts with a whole number, and divides it by 2 if even, or triples and adds 1 if odd, then repeats until the value 1 is reached.

This process **always** seems to reach the value 1, but the mathematician Lothar Collatz could never prove this. Here is a set of commands that carry out the process:

- 1 If n is 1, stop;
- 2 If n is even, replace it by $n/2$;
- 3 If n is odd, replace it by $3*n+1$;
- 4 Go back to step 1.

Hailstones: Example Done By Hand

Suppose we start with the number 17?

17 is odd, so $17 \rightarrow 3 \cdot 17 + 1 = 52$;
52 is even, so $52 \rightarrow 52/2 = 26$;
26 is even, so $26 \rightarrow 26/2 = 13$;
13 is odd, so $13 \rightarrow 3 \cdot 13 + 1 = 40$;
40 is even, so $40 \rightarrow 40/2 = 20$;
20 is even, so $20 \rightarrow 20/2 = 10$;
10 is even, so $10 \rightarrow 10/2 = 5$;
5 is odd, so $5 \rightarrow 3 \cdot 5 + 1 = 16$;
16 is even, so $16 \rightarrow 16/2 = 8$;
8 is even, so $8 \rightarrow 8/2 = 4$;
4 is even, so $4 \rightarrow 4/2 = 2$;
2 is even, so $2 \rightarrow 2/2 = 1$;
1 is 1, so we stop!

Hailstones: Observations

For every starting number Collatz tried, the process ended at 1.

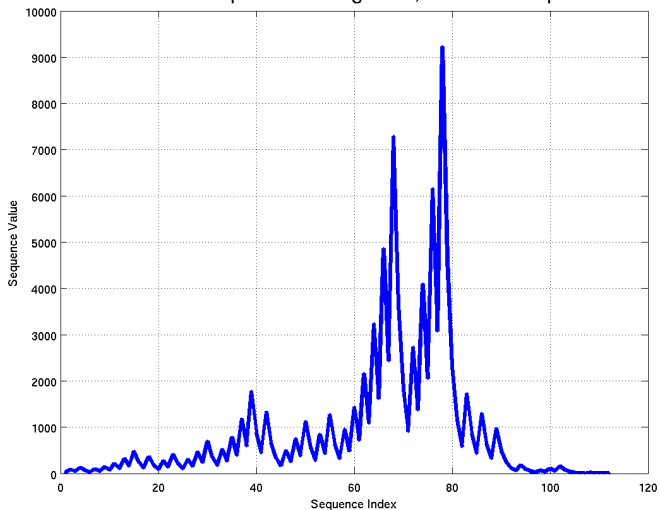
He asked his graduate students to try more examples, which showed the same result. He tried proving mathematically that this must be the case, but he could not. This **Collatz conjecture** remains unproven to this day.

Since the numbers in the sequence fly up and down irregularly before landing at 1, they are often called the *hailstone numbers*. An interesting question is, for any starting value n , how many iterations does it take before the sequence reaches 1 (or the hailstone hits the ground?)

For 17, this value is 12; for 1, this value is 0.

Hailstones: The Case of $N=27$

Collatz sequence starting at 27, takes 112 steps!



Hailstones: Design an Iteration

Computing the hailstone sequence can be done with an iteration; Given a starting value of n , we repeatedly apply the rules until we reach a value of 1. The commands we carry out will be a little more complicated, though!

```
n = 17;
while ( n ~= 1 )
    if ( mod ( n, 2 ) == 0 )      <-- if n is even, do this
        n = n / 2;
    else                        <-- otherwise, do this
        n = 3 * n + 1;
    end
end
n                                <-- print final value of n
```

Hailstones: The Matlab IF Statement

In order to carry out the hailstone process, we needed to use the `mod(a,b)` function, which returns the remainder when the number `a` is divided by `b`. **`mod (n, 2)`** is 0 if `n` is even, and 1 if it is odd.

We also had to use a pair of `if` and `else` statements, which used the results of the `mod()` calculation to decide which action to perform on `n`.

A simpler command involves just an `if` statement, and a more complicated command might involve a three-way test of `if`, `elseif` and `else` statements.

Hailstones: We want to report number of steps!

Our code is correct, but doesn't tell us the number of steps we take. We need to add a variable to count this, and print that at the end.

```
n = 17                                <-- print n at the beginning
step = 0;                             <-- counter starts at 0
while ( n ~= 1 )
    if ( mod ( n, 2 ) == 0 )           <-- if n is even, do this
        n = n / 2;
    else                               <-- otherwise, do this
        n = 3 * n + 1;
    end
    step = step + 1;                  <-- counter goes up by one
end
step                                  <-- print steps at end
```

Hailstones: Now we're cooking!

Now, by modifying the starting value n , we can do some explorations.

The starting value 27 is pretty interesting.

On the other hand, any power of 2 (2, 4, 8, 16, 1024...) is uninteresting, because...why?

If you are ambitious, you can think about how to use a pair of loops, one inside the other, to search for the longest Collatz sequence for a starting value between 1 and some value n_{max} .

BISECTION: Iteration Example 4

Let's go back to the mysterious iteration that had the form

$$x = \text{sqrt} (x + 1)$$

This started out as a computer science assignment statement, but the iteration reached a point where the right and left sides were equal, so that this became a mathematical statement, for a particular value of x .

We were also able to find this value of x by doing some mathematical reasoning that involved the quadratic formula.

However, there is another way to find solutions to an equation, known as the **bisection method**. It produces a useful approximate answer for almost any problem, as long as we have a pair of starting guesses of the right kind.

BISECTION: Find Two Bad Solutions

For convenience, we rewrite our equation as a function $f(x)$:

$$f(x) = x - \text{sqrt} (x + 1)$$

Our search for a solution to the original equation now means we are searching for a value x that makes $f(x) = 0$.

Now we need two starting values of x , such that $f(x)$ is negative in one case and positive in the other. To keep the arithmetic simple, I suggest we look at $x = 0$ and $x = 8$, because then the square roots work out:

$$f(0) = 0 - \text{sqrt} (0 + 1) = -1$$

$$f(8) = 8 - \text{sqrt} (8 + 1) = 5$$

Again, for convenience, we will denote by n the value 0, where the function is negative, and p will stand for the value 8, where the function is positive.

BISECTION: Look in the Middle

Now it's reasonable to expect that $f(x)$, which is negative at $n = 0$ and positive at $p = 8$ must vary smoothly in between, and that means there should be some point between n and p where the function is zero.

Why don't we compute the **midpoint**, call it m , and check $f(m)$?

$$m = (n + p) / 2 = (0 + 8) / 2 = 4$$
$$f(m) = 4 - \text{sqrt}(5) = 1.7639\dots$$

We did not find our solution, but we did find that m was a positive point, and if we replace our old value of $p = 8$ by $p = 4$, we have reduced the size of the interval we are searching.

We can repeat the process, looking at the point m that is halfway between $n = 0$ and $p = 4$, and as we do this, even if we never get an exact solution, our interval will shrink. As it shrinks, we will get better and better approximate solutions.

BISECTION: Express as an Iteration

Here is how the bisection method could be expressed as an iteration:

```
n = 0;
p = 8;
for i = 1 : 10
    m = ( n + p ) / 2;
    fm = m - sqrt ( m + 1 );
    if ( fm < 0.0 )
        n = m;
    else
        p = m
    end
end
end
```

Instead of taking 10 steps, we could use a **while** statement to iterate:

```
while ( 0.0000001 < abs ( p - n ) )
```

BISECTION: First 10 Steps

The first 10 iterations already get us close to the correct answer.

M	F(M)
4.0000000	1.76393202250021
2.0000000	0.267949192431123
1.0000000	-0.414213562373095
1.5000000	-0.0811388300841898
1.7500000	0.0916876048223001
1.6250000	0.00481482539803491
1.5625000	-0.0382810593582121
1.5937500	-0.0167623408406408
1.6093750	-0.00598099791501072
1.6171875	-0.000584888193098099e-04

Summary

Iteration is a kind of repetition; it doesn't simply compute the same number over and over; instead, it applies the same procedure over and over to numbers that gradually change in response.

Iterations can be used to make a plot, by moving step by step through a sequence of values.

Iterations can be carried out a given number of times: a **for loop** or until some desired state is reached: a **while loop**.

Iterations can “converge”, that is, the changes to the data can become so small that the the left and right hand sides are almost equal.

Iterations can be used to search for solutions to mathematical formulas.

Exercises

The following exercises are suggested experiments for you to practice the idea of using Matlab loops to carry out iterations.

You can also find many interesting experiments at the end of Chapter 1 of Moler's book!

Exercise #1: Normalize a Floating Point Number

Given a nonzero number x , we can write it in scientific notation as

$$\text{sign} * \text{mantissa} * 10^{\text{exponent}}$$

where

- sign is $+1$ or -1
- $1.0 \leq \text{mantissa} < 10.0$
- exponent is an integer

Thus, -123.45 has $\text{sign} = -1$, $\text{mantissa} = 1.2345$, and $\text{exponent} = 2$.

Write some Matlab commands that determine the values of sign , mantissa , and exponent for a given number x .

I would use one loop to handle values of x that are 10.0 or larger, followed by a second that handles values of x that are smaller than 1 .

Exercise #2: Random Sum

The Matlab function `rand()` returns a random number between 0 and 1.

Suppose that you are playing a slot machine, and that every time you pull the lever, the amount you win is determined by calling `rand()`. Start out with a variable `winnings` set to zero, and then write a loop that keeps playing until you have won at least 10 units.

If you repeat this exercise, the number of plays will probably be different. You can guess what the average number of plays would be. You could also write a second loop, **around the first loop**, which would let you repeat the experiment 100 times and average the number of steps necessary to win one game.

Exercise #3: Binary Digits

We are familiar with the idea of binary representation of integers. Thus, the number 19 has the binary representation 10011, which happens to mean that $19=1*16+0*8+0*4+1*2+1*1$.

You can compute the binary digits of a positive integer n using an iteration:

As long as n is not zero, you should repeatedly:

- print '1' and replace n by $(n-1)/2$ if n is odd;
- print '0' and replace n by $n/2$ if n is even.

The binary digits will come out in reverse order, so for 19, you will see the values 1, 1, 0, 0, 1 in that order.

Try your iteration for values of n of 19, 31, and 170.

Conclusion

Our simple examples suggest that, at least in computing, it really is possible to do the same thing again and expect a different (maybe better) answer. Iteration is a key technique in numerical work.

The Matlab `for` and `while` commands make it possible to carry out iterations that produce approximations of mathematical functions, solve nonlinear equations, smooth out video images, or track the paths of molecules in a gas.

To see how iterations are used, look at the chapters of Experiments in Matlab on the Exponential function, the Fibonacci Sequence, the Fractal Fern, The Game of Life, the Mandelbrot Set, Orbits, and Sudokus.

In the future, we hope to offer short workshops on other chapters from Cleve Moler's book "Experiments in Matlab!"

Exercise #1

```
if ( x < 0.0 )
    sign = -1.0;
else
    sign = +1.0;
end

exponent = 0;
mantissa = abs ( x );

while ( 10.0 <= mantissa )
    mantissa = mantissa / 10.0;
    exponent = exponent + 1;
end
```

...continued...

Exercise #1

...continued...

```
while ( mantissa < 1.0 )
    mantissa = mantissa * 10.0;
    exponent = exponent + 1;
end

sign
mantissa
exponent
```

Note that this code will fail if $x = 0$!

Exercise #2

```
winnings = 0;
while ( winnings < 10 )
    winnings = winnings + rand ( );
end
winnings
```

Add code to count the steps.

```
winnings = 0;
steps = 0;
while ( winnings < 10 )
    winnings = winnings + rand ( );
    steps = steps + 1;
end
winnings
steps
```


Exercise #2

Play the game 100 times, and compute the average number of steps:

```
step_ave = 0;
for i = 1 : 100

    winnings = 0;
    steps = 0;
    while ( winnings < 10 )
        winnings = winnings + rand ( );
        steps = steps + 1;
    end
    step_ave = step_ave + steps;

end
step_ave = step_ave / 100;
step_ave
```

Exercise #3

```
n = 19;

while ( 0 < n )
    if ( mod ( n, 2 ) == 0 )
        '0'                                <-- A simple way to print '0'
        n = n / 2;
    else
        '1'                                <-- or print '1'.
        n = ( n - 1 ) / 2;
    end
end
```