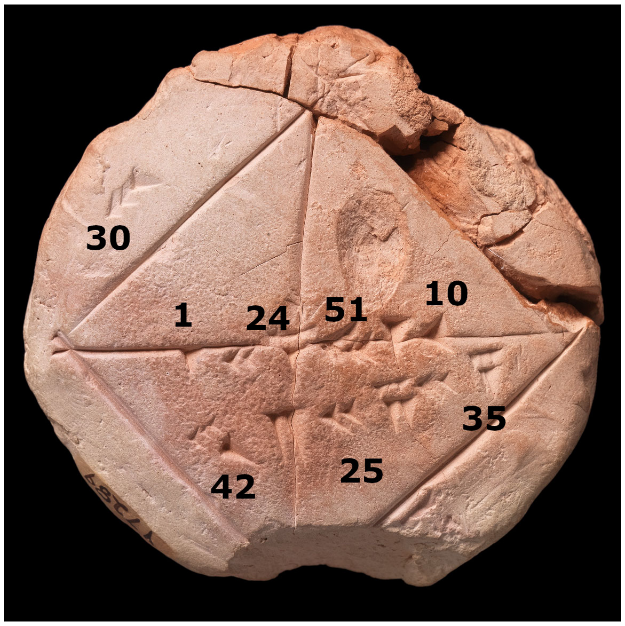
Iteration
Mathematical Programming with Python

https://people.sc.fsu.edu/∼jburkardt/classes/mpp_2023/iteration/iteration.pdf



Class notes made by a Babylonian math student.

$\sqrt{2} \approx 1 + \frac{24}{60} + \frac{51}{60^2} + \frac{10}{60^3}$

An isoceles right triangle with side 30 has a diagonal of length

$d = 30 * \sqrt{2} \approx 42 + \frac{25}{60} + \frac{35}{60^2}$

Iteration

- *Much of mathematics is described by functions $y = f(x)$ that turn input x to output y;*
- *We think of the function as a one-step answer machine: $A = \pi r^2$;*
- *Function iteration applies the function to its output over and over;*
- *For a starting point x, we compute $x, f(x), f(f(x)), ..., f^n(x)...$;*
- *This approach has applications in approximation, randomization, polynomial families;*
- *We don't really want to iterate forever, so we need rules on when and why to stop.*

1 Babylonian Algebra Class

The Babylonians developed mathematical tools for their interests in accounting, surveying, and astronomy. The Babylonians used a base 60 system to represent numbers. Clay tablets have been discovered that give hints of what they calculated, and how. They knew, for example, how to calculate $\sqrt{2}$, or rather, they knew how to estimate it to a desired accuracy. Their method is surprisingly simple, and involves iteration. It's not known how they discovered this method, but we can certainly implement it in Python, and afterwards explore why it works.

`Purpose:`

```
  To determine the square root of x
Initialize:
  Start with r = x
Iterate as often as desired:
  Replace r by the average of r and x/r.
```

Let's try this method to compute the square root of 2. The instruction to iterate "as often as desired" will be interpreted to mean that we will stop as soon as $|2 - r^2| < 1.0e - 6$.

Our code might look like this:

```
n = 0

while ( True ):

  if ( n == 0 ):
    r = x
  else:
    r = ( r + x / r ) / 2.0

  e = abs ( x - r * r )
  if ( e < tol ):
    break

  n = n + 1
```

Notice that for this iteration, we don't specify in advance how many steps we will take. We do try to keep track of this number by updating the value of `n`, but our `while()` loop uses the condition `True` which means, just keep looping forever. Since we definitely don't want to loop forever, we have to escape by specifying some condition that we are waiting for. In our case, we want $|x - r^2| < tol$, and when we see that has happened, we `break` from the loop.

The other thing to notice is that, instead of initializing r outside the loop, we use an `if()` statement, which combines the initialization and update formulas into a single block.

The table below shows how our iteration proceeds when searching for the square root of 2:

n	r	x/r	$x - r^2$
0	2.0	1.0	2.0
1	1.5	1.3333333333333333	0.25
2	1.4166666666666665	1.411764705882353	0.006944444444444198
3	1.4142156862745097	1.41421143847487	6.007304882427178e-06
4	1.4142135623746899	1.4142135623715002	4.510614104447086e-12

This result was surprisingly quick. Looking up the actual value to 16 digits, we get $\sqrt{2} \approx 1.414213562373095$ so we've done pretty well.

Some questions naturally arise:

- Is there some explanation for why this method works?
- Would this same procedure work for $x = 10$, or for $x = 1,000,000$?
- Why does r seem to decrease on every step, while x/r rises?
- Can we get 100 digits of $\sqrt{2}$ this way?
- Can we devise a similar method for, say, cube roots?

For some help with the first question, consider the following: On each iteration, we are averaging two numbers, r and x/r, with $x/r <= r$, and whose product is x. Our next estimate for the square root is the average of these two values, so it must be between x/r and r. That is, the next value of r must be smaller than the current one, and x/r must increase. This suggests that on each step we are actually computing an interval $[x/r, r]$ that is shrinking. If it is shrinking in the right way, then we know that r will converge to $\sqrt{2}$.

2 Decimal / Sexagesimal Conversion

The illustration of the Babylonian tablet indicates that $\sqrt{2}$ was approximated by the sexagesimal value 1.24,51,10 where the digits represent coefficients of powers of 60. Since we are used to base 10, this representation is pretty mysterious. We could evaluate it by writing a program to compute:

```
value = 1 + 24 / 60 + 51 / 60**2 + 10 / 60**3
```

However, let's suppose we have discovered a Babylonian tablet containing a sequence of such values, and we need to convert them all. How could we go about this?

We will assume that we are given an array `d[]` of integers, the base 60 digits of a number in Babylonian form. To keep things simple, we will assume that every such number is a real number strictly between 0 and 60. (Why does this make things simple?).

```
Purpose:
  To evaluate sexagesimal digits, assuming 1 <= value < 60
Initialize:
  Start with value = 0 and base = 1
Iterate n times, counting with i
  Add d[i] * base to value
  Divide base by 60
```

The number of digits we are given may vary, so we have to be more flexible in our programmming:

```
value = 0.0
base = 1.0
for c in d:
  value = value + c * base
  base = base / 60
```

If the `for()` loop looks strange to you, you could replace that code by:

```
value = 0.0
base = 1.0
n = len ( d )
for i in range ( 0, n ):
  value = value + d[i] * base
  base = base / 60
```

The important thing to understand is how we make sure that the first coefficent is divided by 1, the second by 60, the third by 60^2 and so on, as long as necessary. We do this by updating the value of `base` at the end of each iterative step.

You might even think of writing the following code, which is equivalent...as long as our special assumption is true, that the value is between 0 and 60:

```
value = 0.0
for c in d:
  value = value + c / 60**i
```

But let's suppose now that we want to allow the Babylonian information to be greater than 60, or less than 1. Here are two examples:

```
1000000 = 4,37,46.40 = 4 * 60^3 + 37 * 60^2 + 46 * 60 + 40
0.01    = 0.00,36    = 36 * 60^(-2)
```

In order to handle values like these, we need one more piece of information, the value of p, the power of 60 associated with the first digit. For 1,000,000, the value of p is 3, and for 0.01, the power is -2. For the examples we looked at before, the value of p is 0.

So now, let's consider a revised code that accepts an array of digits `d[]` and the value `p`, the power of 60 associated with the first digit. We can easily adjust our code with a single modification:

```
value = 0.0
base = 60**p
for c in d:
  value = value + c * base
  base = base / 60
```

This is probably the simplest way to handle the computations we are interested in.

3 Bisection

Let's return to the problem of evaluating $\sqrt{2}$ and think of it in a new way. We will invent a convenient function $f(x)$, which, for any input x, tells us how well we have approximated $\sqrt{2}$:

$$f(x) = x^2 - 2$$

We have changed our point of view, so that now we are interested in finding a "root" of the equation $f(x) = 0$.

We may know roughly where the solution is, say somewhere between $x = 0$ (too low!) and $x = 5$ (much too large). But notice that $f(0) = -2$ and $f(5) = 23$. Assuming that $f(x)$ is a continuous function, then $[0, 5]$ is a *change-of-sign interval* for $f(x)$. This means $f(x)$ has to rise from a negative value to a positive value over that interval. And that means there must be at least one value x inside that interval such that $f(x) = 0$. In other words, a solution exists (we knew that) and it's in the interval $[0, 5]$.

How can we use this information? Let's call our two interval endpoints a and b. In a way similar to the Babylonians, let's look at the average of our two values, $c = \frac{a+b}{2}$, which in this case is $c = 2.5$. There are three possibilities:

$$f(c) \begin{cases} = 0, & \text{and we have found our solution;} \\ > 0, & \text{so our solution is in } [a, c]; \\ < 0, & \text{so our solution is in } [c, b]. \end{cases}$$

In fact, $f(c) = 4.25$, so our solution must be in the interval $[0, 2.5]$.

Let's rename things, so that the letter b now stands for our updated right endpoint, $b = 2.5$. We can try the averaging trick again, to get a new value $c = 1.25$, for which $f(c) = -0.4375$. This means that the root must lie in the interval $[a = 1.25, b = 2.5]$

You should get the feeling (correctly) that we can continue this process indefinitely. Except for the unlikely event of hitting the solution exactly, on every step we average the two endpoints, evaluate the function at the average, and use the sign of that function value to determine how to reduce our interval in half.

Because our interval is getting smaller, we know we are *converging* towards the solution. Presumably, two things are getting smaller during our iteration:

- the interval size $b - a$ is cut in half on every step;
- the function value $f(c)$ should begin to decrease once we are close enough.

This gives us several choices for how to terminate the iteration:

- if the interval size $b - a$ is smaller than some tolerance;
- if the function value $f(c)$ is smaller than some tolerance.
- if the number of iterations is too large.

The first choice is the preferred way to handle things, since we can guarantee that the interval shrinks at a given rate, and so we can be sure to approximate the location of the root to a desired tolerance.

```
Purpose:
  To approximate x such that f(x) = 0
Initialize:
  Determine a and b that define a change of sign interval for f(x)
  Choose a tolerance t for the interval size
Iterate until |b-a| < tol
  Let c be the average of a and b
  Evaluate f(c)
  If f(c) matches the sign of f(a),
    replace a by c
  Otherwise
    replace b by c
Return
  x = average of a and b
```

Assuming that we have found an initial change of sign interval $[a, b]$, chosen an interval size tolerance t, and defined our function $f(x)$, we can code this as follows,

```
while ( ( b - a ) < tol ):
  c = ( a + b ) / 2.0
  fc = f ( c )
  if ( sign ( c ) == sign ( a ) ):
    a = c
  else:
    b = c
return ( a + b ) / 2.0
```

4 Fixed Point Iteration

We can think of a function $f()$ as a kind of machine that accepts an input x and returns an output $F(x)$. This suggests the simple diagram $x \xrightarrow{f} f(x)$. Of course, if $f(x)$ is just a number, perhaps we can feed it back into the machine, computing $f(f(x))$. And if we can do that once, perhaps we can iterate:

$$x \xrightarrow{f} f(x) \xrightarrow{f} f(f(x)) \xrightarrow{f} f(f(f(x))) \xrightarrow{f} ...$$

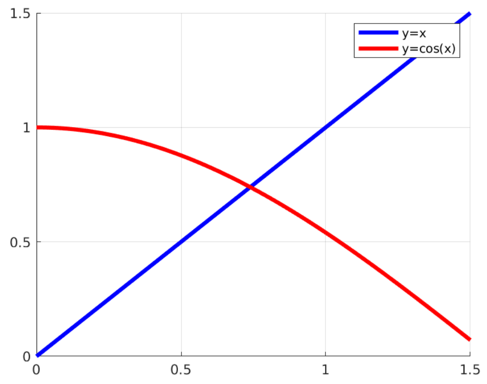
On the old hand-held calculators, you could choose a number x and then repeatedly hit the `SQRT` or `LOG` key, for instance, and see what happens. Many such sequences simply go to zero.

Suppose we start with a number x and hit the `COS` key repeatedly. I started with $x = 1$, and here is what I got:

```
0  1.0
1  0.5403
2  8.8576
3  0.6543
```

```
 4  0.7935
 5  0.7014
 6  0.7640
 7  0.7221
 8  0.7504
 ...
20  0.7392
21  0.7390
22  0.7391
23  0.7391
```

We seem to have discovered an iteration that jumps up and down in smaller increments as it converges towards a value that is approximately $x = 0.7391$. Things become a little more clear if we plot the functions $y_1 = x$ and $y_2 = \cos(x)$:

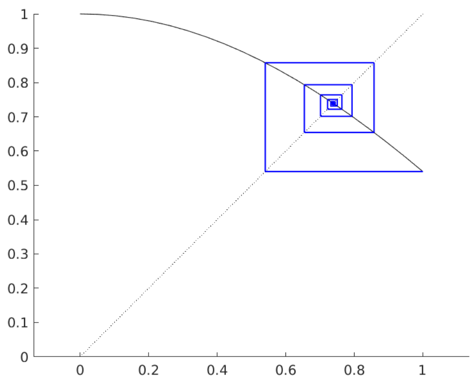


The functions $y_1 = x$ and $y_2 = \cos(x)$ intersection around $x = 0.7391$.

This plot tells us that there is a solution to the equation $x = cos(x)$. But how did we get there? The process we employed is known as a *fixed point iteration*. Suppose we seek a solution of an equation, and we can rewrite it as $x = f(x)$. In our case, the equation would be some version of $x = cos(x)$ or $cos(x) - x = 0$. Then we try the iteration that starts from some given value x_0 and computes a sequence of values by repeatedly computing $x_{i+1} = f(x_i)$. Under certain conditions, this process will converge to a solution to the given equation.

We can try to illustrate the fixed point iterative process by a *cobweb plot*. This is done by plotting the sequence of line segments beginning with:

$$(x_0, x_1) \rightarrow (x_1, x_1)$$
$$(x_1, x_2) \rightarrow (x_2, x_2)$$
$$...$$
$$(x_{n-1}, x_n) \rightarrow (x_n, x_n)$$



A cobweb plot shows our iteration converging to a fixed point of $x = \cos(x)$.

We can outline an algorithmic form for the fixed point iteration:

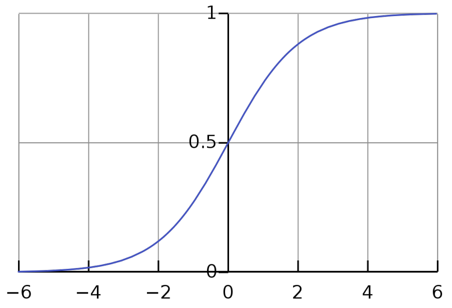
```
Purpose:
  To solve an equation that can be written as $x=f(x)$.
Initialize:
  Choose a starting value for $x$.
  Choose nmax, a maximum number of steps
    OR
  Choose a tolerance t1 for |x-f(x)|
    OR
  Choose a tolerance t2 for |x-xold|
Iterate :
  Set xold = x
  Set x = f(x)
  if nmax <= n finish
  if |x - f(x)| < t1 finish
  if |x - xold| < t2 finish
```

As you can see, there are several tests we can make to decide whether to halt the iteration. A fixed point iteration is only guaranteed to converge under certain conditions; even then, the rate of the convergence may be very slow. So limiting the number of steps is a good idea. One or both of the other conditions should be checked as well.

5 The Logistic Map

It's time for a little vocabulary.

The]itlogistic function $y = \frac{1}{1+e^{-x}}$) is a handy device for modeling quantities that start out very small, have a growth spurt, and then level off as they approach a maximum value of 1. The graph of the function is somewhat like the sigmoid shape of the `arctan(x)` and the error function `erf(x)`.



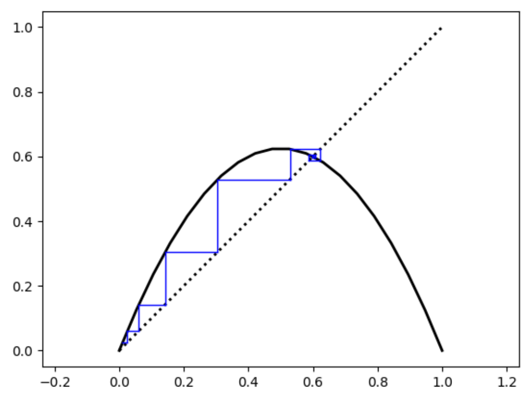
The `logistic equation` is a a differential equation $y' = y(1-y)$ which is often used to describe the evolution of a population in an environment which has a limited supply of food. The population at first rises dramatically but then slows as it approaches the maximum sustainable level. In fact, the ODE solution simply traces the graph of the logistic function.

The `logistic map` is a discrete version of the logistic equation. Starting with some value y_0 in the unit interval, we repeatedly apply the transformation $y_i = ry_{i-1}(1-y_{i-1})$, where $0 \le r \le 4$ is a constant whose effect on the solutions will be interesting to observe.

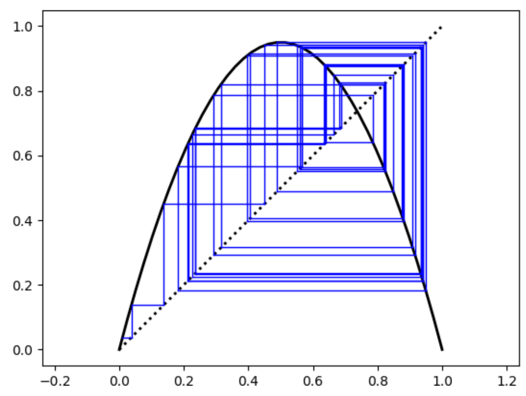
```
Purpose:
  Compute n applications of a logistic map
Initialize:
  r = ?, a value between 0 and 4
  i = 0
  y(0) = y0, a value between 0 and 1
Iterate n times
  increment i
  y(i) = r * y(i-1) * ( 1.0 - y(i-1) )
Plot:
  the function y=r*x(1-1)
  the diagonal line y=x
  (y(0),y(0)) to (y(0),y(1))  vertical
  (y(0),y(1)) to (y(1),y(1))  horizontal
  (y(1),y(1)) to (y(1),y(2))  vertical
  (y(1),y(2)) to (y(2),y(2))  horizontal
  ...
  (y(n-1),y(n)) to (y(n),y(n)) horizontal
```

As we saw in the fixed point examples, we expect the function plot and the diagonal line to cross at at least one point, which represents a solution. Then we expect the cobweb lines to spiral in towards a solution point in a convergent fashion.

The logistic function and logistic equation seem very regular and well behaved. For low values of r, the logistic map works like the fixed point examples we have already looked at. For any starting point in $[0, 1]$, the iterates gradually work their way towards a solution of $x = r\,x\,(1-x)$. Here, for instance, is the cobweb plot for a logistic map iteration with $x_0 = 0.01$ and $r = 2.5$:



However, as the value of r increases, the convergence of the iterates slows down, and then some vary strange things happen. For some values of r, the iterates "dance around" the solution, but keep a certain distance away, in a sort of periodic behavior. In other cases, the iterates jump all over, seeming to produce a chaotic scribble. In fact, the logistic map is one of the earliest examples of a system that can be gradually driven to chaotic behavior. Because of its simplicity, it has been analyzed extensively, and has provided insight into how seemingly deterministic systems can switch from smooth behavior to irregularity. Here is a plot of 40 steps of the logistic map for $r = 3.8$:



6 The Arithmetic Geometric Mean

Suppose we have two real positive values x and y. Their *arithmetic mean* is $a = \frac{x+y}{2}$, while their geometric mean is $g = \sqrt{ab}$. In general, these two quantities are distinct, and $g \leq a$. Lagrange found a useful new quantity, the *arithmetic geometric mean* or AGM, which essentially combines these two values using an iteration, as follows:

```
Purpose:
  Estimate the AGM of x and y
Initialize:
  i = 0
  a(0) = x
  b(0) = y
Iterate
  increment i
  a(i) = arithmetic mean of a(i-1), b(i-1)
  g(i) = geometric mean of a(i-1), b(i-1)
  if ( | a(i) - g(i) | < tol ) exit the iteration
Return:
  arithmetic mean of a(i), g(i)
```

It's fairly easy to set up a Python code `agm(x,y)` corresponding to this algorithm. Once we've done that, here is the result of computing AGM(12,4):

i	a	b
0	12.0	4.0
1	8.0	6.928203230275509
2	7.464101615137754	7.4448388728167965
3	7.454470243977275	7.454464021982545
4	7.454467132979911	7.454467132979261
5	7.454467132979586	7.454467132979586

We see that the a values decrease while the g values increase, until they meet with 16 digit accuracy, after just 5 iterations.

Lagrange seems to have "discovered" the AGM while he was investigating the evaluation of a quantity known as Legendre's elliptic integral of the first kind:

$$I = \int_0^{\frac{\pi}{2}} \frac{1}{\sqrt{a^2\cos^2(\theta)) + b^2\sin^2(\theta)}} d\theta$$

He was able to show that the value of I was unchanged if a and b were replaced by their arithmetic and geometric means, respectively. But this replacement process could be repeated indefinitely, so that, in the limit, we have

$$I = \int_0^{\frac{\pi}{2}} \frac{1}{\sqrt{agm(a,b)^2\cos^2(\theta)) + agm(a,b)^2\sin^2(\theta)}} d\theta = \int_0^{\frac{\pi}{2}} \frac{1}{agm(a,b)} d\theta = \frac{\pi}{2\,agm(a,b)}$$

which means that the difficult-looking integral can be evaluated immediately.

There is also a quantity known as the geometric-harmonic mean, or `GHM(x,y)`. We first need to define the harmonic mean $h(x,y) = \frac{2}{\frac{1}{x}+\frac{1}{y}}$. Now, to compute the geometric-harmonic mean, we start with $g_0 = x, h_0 = y$ and then carry out an iteration similar to what we did for `AGM(x,y)`, so that $g_i = g(g_{i-1}, h_{i-1})$ and $h_i = h(g_{i-1}, h_{i-1})$.

7 A Perfect Shuffle

In a perfect shuffle, a deck of 52 cards is split into two piles, which are then interleaved. in a perfect out-shuffle, the resulting deck still has the top card on top and the bottom card on the bottom. If the original cards are numbered by position from $k = 0$ to 51, then after a perfect outshuffle, the card at position k will have number $s(k)$:

$$s(k) = \begin{cases} \frac{k}{2} \text{ if } k \text{ is even;} \\ \frac{k+51}{2} \text{ if } k \text{ is odd;} \end{cases}$$

We can illustrate this situation as follows:

```
 0    1    2    3    4    5    6  ...   24    25     first pile
   26   27   28   29   30   31   32  ...   50    51  second pile
---------------------------------------------------
 0 26  1 27  2 28  3 29  4 30  5 31  6 32 ... 24 50 25 51  new shuffled deck
```

And you can see that the card in position $k = 7$ has the number $s(k) = 29$.

Suppose we have a deck of cards, which we have initialized as follows:

```
deck = list ( range(52) )
```

and we want to apply a perfect outshuffle to it. We can program that using the formula above, as follows:

```
deck2 = np.zeros ( 52 )

for i in range ( 0, 52 ):
  if ( ( i % 2 ) == 0 ):
    j = i // 2
  else:
    j = ( i + 51 ) // 2
  deck2[i] = deck[j]
```

Alternatively, we can do the more suggestive code:

```
deck2 = np.zeros ( 52 )
deck2[0:52:2] = deck[0:26]
deck2[1:52:2] = deck[26:52]
```

Now suppose that we implement one of these two procedures as a function `outshuffle(deck)`, which returns a shuffled version of the input deck. Then we can explore how well a perfect shuffle mixes up the cards.

We might start by setting

```
deck = list ( range(52) )
deck = outshuffle ( deck )
```

Now suppose we have heard that something interesting happens if we apply 8 perfect outshuffles to our deck. Instead of typing 8 `outshuffle()` calls, we use iteration:

```
deck = list ( range(52) )
for i in range ( 0, 8 ):
  deck = outshuffle ( deck )
print ( deck )
```

and we find something surprising, which may also be of interest to magicians, who are perfectly capable of perfect shuffles!

8 Exercises

1. Using the ideas of the Babylonian square root computation, can you think of a similar approach that will compute estimates for the cube root of a number?
2. Evaluate the number whose Babylonian representation is 1.24,51,10 and compare it to the square root of 2.
3. Write a code that takes a real number r and produces the first n base 60 digits in its representation. To start with, you may assume $1.0 < r < 60$, and you should test your code to get the first $n = 5$ sexagesimal digits for $r = \pi$.
4. If you can get the previous exercise to work, consider how you would improve your code so that it can handle cases like $r = 0.01$, $r = 1000000$, $r = 0$ and $r = -100$.

5. Suppose you plan to use the bisection method to find a root $x*$ of a version of the Kepler equation: $x - 2\sin(x) = t5$. Your initial interval is [0,10], and you want to estimate the value of $x*$ with a tolerance of $t = 10^{-6}$. You can actually predict in advance the number of steps n you will need. What is the value of n for this problem? What is a general formula for this question?
6. The fixed point iteration for a root of $x = cos(x)$ converges to $x \approx 0.7391$ when we used an initial value of $x = 1$. Can you find starting points for which the iteration converges to a different value? Can you find starting points for which the iteration does not converge at all? What fact about the range of the cosine function makes this question somewhat easier to answer?
7. Write a program to carry out fixed point iteration; seek a root of the equation $x = 1 + 1/2sin(x)$ starting at $x = 0$; then seek a root of the equation $x = 3 + 2sin(x)$ starting at $x = 3$. The fixed point method only works on one of these cases. There is a simple test which explains why one problem is solvable and the other is not. Find out what this test is, and apply it to your two cases.
8. Create a cobweb plot for the "tent iteration", which has the form $x = 2 * |x - 0.5|$. Your plot should look chaotic. Is there some small change you can make to the right hand side which will result in more regular behavior? Remember that your formula must still produce a new value x in the interval $[0, 1]$. What would happen with $x = 4 * (x - 0.5)^2$, for instance?
9. Suppose that we have two positive distinct real values a, b. Show that their arithmetic mean must be greater than their geometric mean. You might start by noting that $0 < (a - b)^2$.
10. Write a code $ghm(x, y)$ that can compute the geometric-harmonic mean of two numbers. Demonstrate by example that the sequences of g and h values converge monotonically to a common value; Demonstrate that $ghm(x, y) = \frac{1}{agm(\frac{1}{x}, \frac{1}{y})}$.
11. A perfect *inshuffle* is similar to a perfect outshuffle, except that the new deck has the ordering 26 0 27 1 28 2 29 3 30 4 31 5... 50 24 51 25. Write a program that implements a perfect inshuffle. We found something remarkable happens to a deck after 8 perfect outshuffles. Does this also happen after 8 perfect inshuffles? If not, does it happen eventually? Hint: you'll need fewer than 100 shuffles to see this.