Solving an Equation

http://people.sc.fsu.edu/∼jburkardt/isc/week09
lecture_16.pdf

....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 05 July 2011

EQUATIONS

- **Introduction**
- Project Discussion
- When does Cos(X) $=$ X?
- The Bisection Idea
- A Sample Bisection Solver
- Bracketing the Solution
- Lab Exercise #8

INTRO: Schedule

Next Class:

- The Secant and Newton Methods

Assignment:

- Today: in-class lab exercise #8
- Thursday: Programming Assignment #6 is due.
- Tuesday, July 12: 1 paragraph project proposal is due.

INTRO: Projects, Problems, Programs

For the month of July, you will need to choose and work on a final project. This is a feature of the FSU Computer Competency Requirement. I will suggest some topics, and give you some guidelines.

We have learned enough C++ to be “dangerous”. That is, it’s a good time to stop learning about pieces of the language, and to focus on how they are put together to solve problems. This is a chance to understand better some of the things we think we’ve learned.

One example of a common scientific task involves solving an equation, or, more accurately, getting a good estimate of a sol[illegible]on to a nonlinear equation. We will seek a program to help us.

INTRO: Solving a Nonlinear Equation

We will start by looking at some examples of this kind of problem, and look at a simple idea that can help us, as long as we are able to make a rough guess as to where the answer is.

We will put this idea into a C++ code.

Then we will think about how we can reuse this code. That will involve turning it into a function, which takes in an equation and spits out a solution. To do that we have to answer the question:

How can we make an equation become an input quantity?

EQUATIONS

PROJECTS: A Course Requirement

We are required to do a final project for this course.

Each student needs to pick a topic, investigate it, write a program that examines some feature of the topic, write a short report and make an oral presentation.

These presentations will be made on our last day of class, Tuesday, August 2nd.

To start the process, I need each of you to turn in, by Tuesday, July 12, a paragraph describing a project you will work on.

I have talked with a few students who already had an idea about a project to work on, but I understand many of you may have n ideas about what to do.

PROJECTS: Possible Topics

I have a one page outline of the project guidelines for you, followed by a list of 15 possible topics you might choose. Most of these topics are associated with an article or short writeup, and I have printed one copy of each of those, as well.

I invite you to look over the articles to find something that interests you.

Some of the articles seem long and complicated. However, in each one, I believe there is a simple idea that can be investigated and programmed. I am willing to help each of you get started on forming a project from one of these articles, especially during my office hour time on Friday.

Remember, I need a decision from each of you by next Tuesday.

EQUATIONS

COS(X): A Coincidence?

If we look at a table of cosines, we notice something interesting happening a little bit after $x = 0.7$:

```
   X            COS(X)
 --------     --------
 0.700000     0.764842
 0.710000     0.758362        <-- X < COS(X)
 0.720000     0.751806
 0.730000     0.745174
---------------------
 0.740000     0.738469
 0.750000     0.731689
 0.760000     0.724836        <-- X > COS(X)
 0.770000     0.717911
 0.780000     0.710914
 0.790000     0.703845
 0.800000     0.696707
```

COS(X): A Coincidence?

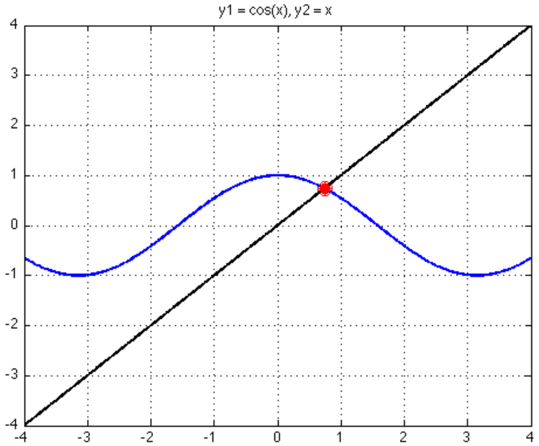
Noticing that at some point, x becomes larger than $\cos(x)$, we can ask ourselves, is it possible that for some value of x, it is true that $\cos(x) = x$ exactly?

If you think about it, you probably know that $\cos(x)$ is a continuous function, and, for our purposes, one property of a continuous function is that its graph cannot have any holes or jumps.

And that means that if a mountain climber **A** is walking down "Mount Cosine", and a lazy person **B** is gliding up "Escalator X", then if at some point **A** is above **B**, and later the positions are reversed, there must have been a time when **B** caught up to **A**, at which point $\cos(x) = x$.

COS(X): Compare Graphs of COS(X) and X

When is it true that $\cos x = x$?



COS(X): A Graph is Quick but Expensive and Crude

Now the graph makes it look obvious that this must happen.

However, a graph is not always the best way to answer this kind of question, because:

- we had to know where to look;
- we had to call the function about 500 times to get a picture;
- the accuracy of the answer is at best 1 decimal place for this picture;
- to get more accuracy, we'd have to do a new plot, on a finer scale, with 500 more values (and again, and again).

COS(X): Look for Crossings

We looked at this problem in an earlier homework exercise. Starting in [0,1], we chose 11 points, and looked for the crossing:

```
 1   0.000000   1.000000
 2   0.100000   0.995004
 3   0.200000   0.980067
 4   0.300000   0.955336
 5   0.400000   0.921061
 6   0.500000   0.877583
 7   0.600000   0.825336
 8   0.700000   0.764842   <-- X < COS(X)
--------------------
 9   0.800000   0.696707   <-- X > COS(X)
10   0.900000   0.621610
11   1.000000   0.540302
```

We zoomed in to [0.7, 0.8] with 11 more points, and so on.

COS(X): Create a function F1(X) = COS(X) - X

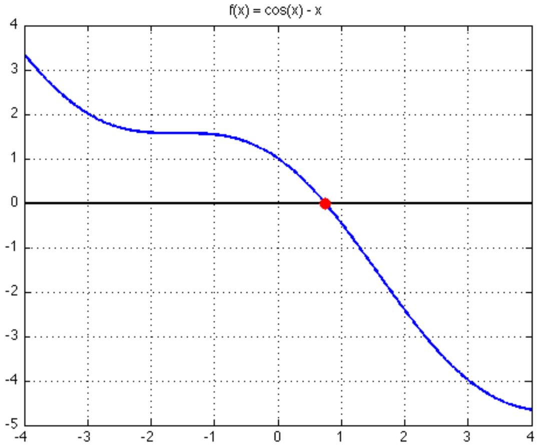
Now I'm going to try to make it easier to find crossings, by considering a function $f_1(x) = cos(x) - x$:

```
  1  -1.000000
  2  -0.895004
  3  -0.780067
  4  -0.655336
  5  -0.521061
  6  -0.377583
  7  -0.225336
  8  -0.064842  <-- F1(X) < 0 (same as X < COS(X) )
--------------------
  9   0.103293  <-- F1(X) > 0 (same as X > COS(X) )
 10   0.278390
 11   0.459698
```

It's easier to look for a change in sign in one list!

COS(X): Graph of F1(X) = COS(X) - X

Rewriting $\cos(x) = x$ as $f_1(x) = \cos(x) - x$ gives new perspective. Where does the graph cross the x axis?



COS(X): Graph of F1(X) = COS(X) - X

By itself, knowing the value X for which COS(X) = X is not important. But it is an example of a kind of calculation that goes on all the time in scientific computing, called **the solution of a nonlinear equation**.

A workable strategy for many examples of this problem is:

1. rewrite your problem as a function that must equal 0;
2. find an interval **[a,b]** where the function changes sign;
3. cut the interval in half after evaluating the midpoint;
4. repeat step 3 until satisfied.

This procedure is known as the method of bisection.

EQUATIONS

BISECT: Step 1 Requires a Function

Let's think about the steps involved in the bisection method, using our example problem $cos(x) = x$ as a guide.

Step 1: **Rewrite your problem as a function that must equal 0.**

It seems easy to do this:

$$f_1(x) = cos(x) - x$$

but of course, now we probably also need to implement this as a C++ function!

BISECT: The Cosine Function

```
double f1 ( double x )
{
  double value;

  value = cos ( x ) - x;

  return value;
}
```

BISECT: Step 2 Requires a Change of Sign Interval

Having described our problem as a function $f_1(x)$, we are ready for step 2:

Find an interval [a,b] where the function changes sign.

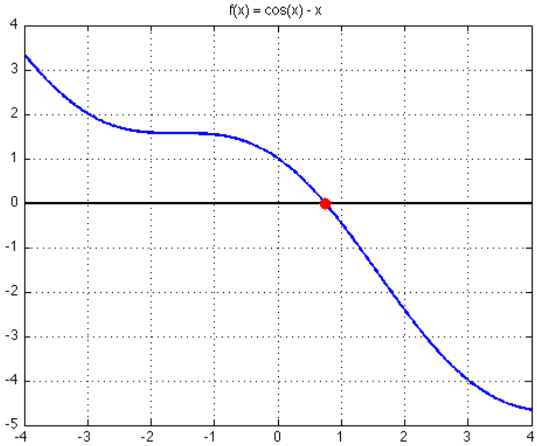
There are many ways that this can be done. We really just need two values at which the function has opposite sign. We might have stumbled on such values, or guessed them from the equation, or used the computer to sample some values, or even looked at a graph.

A change of sign interval is very important. It tells us there is a solution, and it "fences in" the location of a solution.

If we graphed $f_1(x)$ over the range [-5,+5], it would make sen[illegible] to choose the interval [0,1] as our **[a,b]**. (For our example, we'[illegible] pretend we picked [-10,+10] instead...)

BISECT: F1(X) = COS(X) - X

Rewriting $\cos(x) = x$ as $f_1(x) = \cos(x) - x$ gives new perspective:



BISECT: Step 3 Cuts the Interval in Half

Step 3 says:

Cut the interval in half after evaluating the midpoint.

What we are saying here is this. Rather than picking 11 points in the interval **[a,b]** to sample the function, just pick the midpoint, $c = \frac{a+b}{2}$, and evaluate $f_1(c)$. Now either $f_1(c)$ matches the sign of $f_1(a)$, or of $f_1(b)$. Whichever point has the same function value sign as **c** indicates the half of the interval we can discard. The remaining half is still a change of sign interval, and is guaranteed to contain a solution.

BISECT: First Bisection

To see what this means in practice, assume we started out with **[a,b]** = [-10,+10]. Our initial table looks like this:

```
             A                B
    X:   -10.0           +10.00
 F1(X):    9.16          -10.83
```

Setting $c = \frac{a+b}{2} = 0$ for which $f_1(c) = 1.0$. Now our table is:

```
             A       C        B
    X:   -10.0     0.00  +10.00
 F1(X):    9.16    1.00  -10.83
```

The value of $f_1(c)$ tells us that the solution must be in the interval $[c, b]$. So if we replace the value of a by c, then it is again true that our solution is somewhere in the (smaller) interval $[a, b]$.

BISECT: Second Bisection

After one step, our table looks like this:

	A	B
X:	0.00	+10.00
F1(X):	1.00	-10.83

The average of a and b is $c = 5$, for which $f1(5) = -4.71$.

	A	C	B
X:	0.00	5.00	+10.00
F1(X):	1.00	-4.71	-10.83

and so $f_1(c)$ tells us that solution is in the left half interval $[a, c]$
If we replace the value of b by c, it is again true that our sol
is somewhere in the (smaller) interval $[a, b] = [0,5]$.

BISECT: Step 4: When Do We Stop?

Step 4 says **Repeat step 3 until satisfied.**

Well, what does that mean? Are we getting satisfied?

On each step, our interval is cut in half. That means we are getting a better and better estimate of the location of the answer.

Moreover, while the function value at the midpoint will not necessarily go down smoothly, once the interval becomes small we expect the function to decrease regularly.

So things are getting better, but we most likely will never actually reach a solution for which $f_1(x) = 0$ exactly. This is simply a reality of computing.

Therefore, the important issue for step 4 is <u>when do we stop?</u> start with, we'll keep things simple and expect the absolute value of the function to get pretty small.

BISECT: What Parts Become a Program?

It really seems like this idea can be automated. So rather than doing any more computing by hand, let's think about what we did, and what parts we could turn over to an algorithm.

The user has to describe the function.

The user needs to specify the initial interval **[a,b]**.

After that, the bisections are automatic.

The check for when to stop should also be some kind of automatic decision.

So steps 3 and 4 of the bisection method can be easily packag
into a C++ procedure, if the user takes care of steps 1 and 2

EQUATIONS

- Introduction
- Project Discussion
- When does Cos(X) $=$ X?
- The Bisection Idea
- **A Sample Bisection Solver**
- Bracketing the Solution
- Lab Exercise #8

SOLVER: A Function to Solve F(X)=0

We are going to try to write a function that will find a solution to an equation for us. We'll think of that equation as being written in the form $f(x) = 0$.

Our C++ function needs a name, and a description of the input and output. Let's call this C++ function **bisect1()**. *(The "1" is because it's our first try, and we'll try again later!)*

The output, obviously, will be the solution estimate.

For input, the least we have to specify is **a** and **b**, the left and right endpoints of the search interval. The user function is also a sort of input, but (for now) it's not really the same kind of inp that **a** and **b** are, so we treat it differently.

SOLVER: A Function to Solve F(X)=0

So our declaration might be:

```
double bisect1 ( double a, double b );
```

and we expect a typical use to look like this:

```
a = 0.0;
b = 10.0;
x = bisect1 ( a, b );
```

We are assuming, of course, that the (mathematical) function $f(x)$ we are trying to zero out is defined by the user in a separate (C++) function, called **f()** and declared as:

```
double f ( double x );
```

SOLVER: A Sketch of the Function

The **bisect1()** code, which works with a function **f()**, might be:

```
  double bisect1 ( double a, double b )
  {
    double f ( double x );                <-- This declaration could go at the top of the file instead.
    double c;
    while ( true )
    {
//
//  C = midpoint.
//
      c = ( a + b ) / 2.0;                <-- This begins "step 3"
//
//  Is F(C) extremely close to 0?
//
      if ( fabs ( f ( c ) ) < 0.000001 )  <-- This is our "step 4"
      {
        break;
      }
//
//  If F(C) is opposite in sign to F(A), C replaces B.
//
      if ( f ( c ) * f ( a ) < 0.0 )      <-- This completes step 3.
      {
        b = c;
      }
      else
      {
        a = c;
      }
    }
    return c;
  }
```

SOLVER: Making A Program

Suppose we try the solver on our cosine function **f1(x)**. My copy of **bisect1()** must be changed to call **f1()** rather than **f()**...

```
# include <cstdlib>
# include <iostream>
# include <cmath>

using namespace std;

double bisect1 ( double a, double b );   <-- Declare it up here, once and for all.
double f1 ( double x );                  <-- Declare it up here, once and for all.

int main ( )
{
  double a = -10.0, b = +10.0, c;

  c = bisect1 ( a, b );

  cout << "\n";
  cout << "BISECT1 returned solution estimate C = " << c << "\n";
  cout << "F1(C) = " << f1 ( c ) << "\n";

  return 0;
}
```

...text of bisect1() function goes here, changed to call f1()...
...text of f1() function goes here...

SOLVER: Using the Solver

I will modify the solver to print out the midpoint **c** and **f1(c)** at each step:

```
Step   C          F1(C)
----  --------  ----------
 1    0.0         1
 2    5.0        -4.71634
 3    2.5        -3.30114
 4    1.25       -0.934678
 5    0.625       0.185963
 6    0.9375     -0.345695
 7    0.78125    -0.0712161
 8    0.703125    0.0597003
 9    0.742188   -0.00519571
10    0.722656    0.0273953
11    0.732422    0.0111353
12    0.737305    0.0029786
13    0.739746   -0.00110635
14    0.738525    0.000936676
15    0.739136   -8.47007e-05
16    0.738831    0.000426022
17    0.738983    0.000170669
18    0.739059    4.29864e-05
19    0.739098   -2.08566e-05
20    0.739079    1.1065e-05
21    0.739088   -4.89575e-06
22    0.739083    3.08466e-06
23    0.739086   -9.05543e-07    <-- F1(C) < 0.000001, so we stop.
```

SOLVER: Improvements

There are several ways our first program could be improved:

- we require the function to be named **f(x)**;
- the user might want more or less accuracy;
- if the interval is small enough, that might also be a good indication that we can stop;
- we might also want to limit the total number of steps taken;
- the program doesn't check that there is a change of sign between **f(a)** and **f(b)**.

We will look at making some of these improvements in version 2 of the program!

EQUATIONS

- Introduction
- Project Discussion
- When does Cos(X) $=$ X?
- The Bisection Idea
- A Sample Bisection Solver
- **Bracketing the Solution**
- Lab Exercise #8

BRACKET: Find Change of Sign

In order to get our bisection solver started, we need a pair of values **a** and **b** for which **f(x)** changes sign. Sometimes, it's possible to look at the equation and guess two values that work. But even then, we need to verify our guess by computing the value of the function.

It may be helpful, therefore, to have a program called **bracket()** which can evaluate our function for us interactively.

The simplest version of such a program might simply ask us to type in values **x** at which the function should be evaluated.

BRACKET: Sample Session for F1(X) = COS(X) - X

On the web page, there's a program called **bracket.cpp** which does this for the function $f_1(x) = \cos(x) - x$. That means the program includes a copy of the **f1()** function, and waits for us to type in values **x** to try out. When we're satisfied, we type CTRL-D to end the input.

```
BRACKET:
  Enter a value X, to receive a function value F(X).
  Terminate with a CTRL-D or end-of-file.
5
  F(5) = -4.71634
4
  F(4) = -4.65364
3
  F(3) = -3.98999
2
  F(2) = -2.41615
1
  F(1) = -0.459698
0
  F(0) = 1           <-- positive value at x=0!
0.5
  F(0.5) = 0.377583
0.6
  F(0.6) = 0.225336
CTRL-D
```

EQUATIONS

EXERCISE: Lambert's Function

The function $W()$ is known as *Lambert's function*. It doesn't have a simple formula. However, we can ask when Lambert's function is equal to a specific value, such as 1,000. In that case, we can write our question as seeking the value x so that

$$x * e^x = 1000$$

Our goal is to use the bisection method to find a solution of this equation.

EXERCISE: Step 1

Step 1 is to rewrite this problem as a function, which we can call $f_6(x)$:

$$f_6(x) = x * e^x - 1000$$

To solve the problem, we will have to write a corresponding C++ function

double f6 (double x)

which evaluates our function. Note that the value e^x can be computed in C++ by typing **exp (x)**, and that, because we are calling the **exp()** function, your program will need the statement

```
# include <cmath>
```

EXERCISE: Step 2

Step 2 requires us to find two values **a** and **b** for which the function changes sign.

The interactive program **bracket()** can be used for this purpose. However, to use it, you must include a copy of your function **f6()**. Since **bracket()** assumes your function is called **f()**, you need to change the name of your function to **f()**, or change **bracket()** to call **f6()**.

Start at 0, and go up by 1's, and you will probably find a change of sign quickly.

EXERCISE: Steps 3 and 4

Steps 3 and 4 are the part we let the **bisect1()** code do for us.

Change the **bisect1()** code by

- inserting your values for **a** and **b**;
- adding your **f6()** function, and again;
- changing **f()** to **f6()** or vice versa;

Now run the program to get an estimate for the solution.

Once your program has printed its results, please show your work to Detelina so you can get credit for the exercise!