Debugging: How to find the errors ....

As some of you may have discovered, there are times when -- despite your carefully crafted pseudocode -- you make a mistake in translating your algorithm into actual Scilab code. We call these mistakes *bugs*. The process of removing them from your program is called *debugging*.

You can find many, many references and guides to debugging all over the Net; going to Google and typing **debugging techniques** returns about 1,340,000 items. There are good chapters on debugging in some of the references for the course.

Let me show you just one, very simple technique that I use frequently. It isn't particular to Scilab, but can be used in just about any programming language.

A buggy program

Consider the following program, which tries to find the root via the bisection technique, but fails.

- root_bisect.sci with a bug

```
function root = root_bisect(lower_bound, upper_bound, epsilon, func)
//
// Find the root of the given function within the given interval,
// using the bisection method.
//
// Arguments:
//
//     lower_bound     (input)    lower bound of interval
//                                on which to search for root
//
//     upper_bound     (input)    upper bound of interval
//
//     epsilon         (input)    when fractional change in root
//                                reaches this limit, stop
//
//     func            (input)    name of function whose root
```

```
//                              we seek -- should take
//                              a single argument
//
//     root         (output)    argument at which the given
//                              function has value zero
//
// MWR 3/25/2002
//

// sanity checks
if (lower_bound >= upper_bound)
  error('lower_bound must be less than upper_bound');
end
if (epsilon <= 0)
  error('epsilon must be greater than zero');
end

// if there is no guaranteed root between the given bounds,
// then quit with an error message
y_low = feval(lower_bound, func);
y_high = feval(upper_bound, func);
if (sign(y_low) == sign(y_high))
  error('function has same sign at both bounds -- quitting');
end

done = 0;
iterations = 0;
old_mid = lower_bound;
while (done == 0)
  y_low = feval(lower_bound, func);
  y_high = feval(upper_bound, func);

  // Pick the half of the interval in which the root must exist
  // 'mid' is going to be next root candidate
  mid = (upper_bound - lower_bound)/2.0;
  y_mid = feval(mid, func);
  if (sign(y_mid) == sign(y_low))
    // root is in upper half
```

```
    lower_bound = mid;
  else
    // root is in lower half
    upper_bound = mid;
  end

  // now, check to see if we are close enough
  if (mid == 0)
    // we can't calculate fractional change here, so use criterion
    // that absolute value of the change in candidates is less than epsilon
    if (abs(mid - old_mid) < epsilon)
      done = 1;
    end
  else
    frac_change = abs((mid - old_mid)/mid);
    if (frac_change < epsilon)
      done = 1;
    end
  end

  // we need to save a copy of the current root candidate, so we
  // can calculate the fractional change since the previous one
  old_mid = mid;

  iterations = iterations + 1;
end

// when we get here, the latest root candidate is in 'mid'
root = mid;

endfunction
```

Now, if we run the program with a very simple function funcz.sci

```
function y = funcz(x)
y = x*x - 5;
```

on the interval [0, 5], we ought to find the root sqrt(5) = 2.236. But when we try, what happens?

Obviously, something is wrong. But what? And where in the source code? Can you find the error? It's pretty hard when all you have to use is the source code itself.

Print and pause to find the bug

Here's a second version of exactly the same program -- it still contains the error. However, this version includes some extra code which is designed to track it down.

- root_bisect.sci with bug and extra debugging equipment

```
function root = root_bisect(lower_bound, upper_bound, epsilon, func)
//
// Find the root of the given function within the given interval,
// using the bisection method.
//
// Arguments:
//
//     lower_bound     (input)    lower bound of interval
//                                on which to search for root
//
//     upper_bound     (input)    upper bound of interval
//
//     epsilon         (input)    when fractional change in root
//                                reaches this limit, stop
//
//     func            (input)    name of function whose root
//                                we seek -- should take
//                                a single argument
//
//     root            (output)   argument at which the given
//                                function has value zero
//
// MWR 3/25/2002
```

```
//

// set this to 1 to watch diagnostic messages as the function runs
verbose = 1;

// sanity checks
if (lower_bound >= upper_bound)
  error('lower_bound must be less than upper_bound');
end
if (epsilon <= 0)
  error('epsilon must be greater than zero');
end

// if there is no guaranteed root between the given bounds,
// then quit with an error message
y_low = feval(lower_bound, func);
y_high = feval(upper_bound, func);
if (sign(y_low) == sign(y_high))
  error('function has same sign at both bounds -- quitting');
end

done = 0;
iterations = 0;
old_mid = lower_bound;
while (done == 0)
  y_low = feval(lower_bound, func);
  y_high = feval(upper_bound, func);

  if (verbose > 0)
    mprintf('iter %5d   %11.7e %9.4e   %11.7e %9.4e ', ...
      iterations, lower_bound, y_low, upper_bound, y_high);
    // pause;
  end

  // Pick the half of the interval in which the root must exist
  // 'mid' is going to be next root candidate
  mid = (upper_bound - lower_bound)/2.0;
  y_mid = feval(mid, func);
```

```
if (sign(y_mid) == sign(y_low))
  // root is in upper half
  lower_bound = mid;
else
  // root is in lower half
  upper_bound = mid;
end

// now, check to see if we are close enough
if (mid == 0)
  if (verbose > 0)
    mprintf('  abs change %9.4e ... ', abs(mid - old_mid));
  end
  // we can't calculate fractional change here, so use criterion
  // that absolute value of the change in candidates is less than epsilon
  if (abs(mid - old_mid) < epsilon)
    if (verbose > 0)
      mprintf(' stop \n');
    end
    done = 1;
  else
    if (verbose > 0)
      mprintf(' keep going \n');
    end
  end
else
  frac_change = abs((mid - old_mid)/mid);
  if (verbose > 0)
    mprintf('  %9.4e \n', frac_change);
  end
  if (frac_change < epsilon)
    if (verbose > 0)
      mprintf(' stop \n');
    end
    done = 1;
  else
    if (verbose > 0)
      mprintf(' keep going \n');
```

```
      end
    end
  end

  // we need to save a copy of the current root candidate, so we
  // can calculate the fractional change since the previous one
  old_mid = mid;

  iterations = iterations + 1;
end

// when we get here, the latest root candidate is in 'mid'
root = mid;


endfunction
```

Look near the top of the main loop. There's a new bit of code that looks like this:

```
  if (verbose > 0)
    mprintf('iter %5d   %11.7e %9.4e   %11.7e %9.4e ', ...
      iterations, lower_bound, y_low, upper_bound, y_high);
    pause;
  end
```

What does it do?

- If the variable called **verbose** is equal to zero, this new bit of code is not executed at all.
- But if **verbose** is set to 1 (for example), then the program will print out the value of the lower and upper bounds of the current range, plus the values of the function at each bound ...
- .... and, after printing the information, the **pause** statement causes the program to wait for the user to type "return" and press the enter key before it continues

Please copy this program to your local disk, rename it to **root_bisect.sci**, and run it. Watch what happens.

Now that you've seen the output of the program, can you explain why it isn't finding the root properly? Can you now find the place in the program which is most likely to contain the error?

Leaving debugging mode

It is often useful to include extra **mprintf** statements in your program as you are developing it. Sometimes, you may add a LOT of these debugging "print" lines. What happens when it's time to turn the program in?

- One option is to go into the source code and delete all the extra **mprintf** statements. But you might accidentally delete some code that you really need. And what happens if you need to go back and add new features to your program?
- Another option is to use the comment characters // to "comment out" the extra lines; that way, they are still present and can be re-activated later.
- But I think the best way (at least for this course) is to add a special "control variable", which is defined ONCE at the start of your program. It controls all the "extra" **mprintf** statements, as well as any **pause** or other debugging material. By changing just a single line:
- ```
  // set this to 1 to watch diagnostic messages as the function runs
  verbose = 1;
  ```

  you can switch the program from its "debugging" state to its "production" state, or back.

Try editing your version of the program: change the **verbose** variable to zero, and run it again. Now what happens?

Source: http://spiff.rit.edu/classes/phys317/lectures/debug/debug.html
  ```