**14:440:127– Introduction to Computers for Engineers**

Notes for Lecture 13

Rutgers University, Spring 2010 Instructor- Blase E. Ur

# 1 Using Multiple Cores / Distributed Computing

`http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/f3-6010.html` contains a good primer for this section since it's too new to be covered in your book.

In recent years, the trend has been to create processors (CPUs) with multiple cores. It started with dual (2) core processors. Now, 4 cores are quite common, and the thought is that this trend will continue. Supercomputers now have thousands of cores and processors. However, Matlab runs on just one processor/core by default. To change this, you need to use the parallel computing toolbox that comes with recent (professional, not student) versions of Matlab. In Matlab, we're used to writing loops like this:

```
z = zeros(1,1000000);
tic;
for a = 1:1000000
     z(a) = isprime(a);
end
toc

%%% Elapsed time is 73.615442 seconds
```

However, we can invoke the parallel computing toolbox through a two step process and end up nearly cutting the time it takes to run that code in half. I'm going to try the same operation on my laptop's dual core processor, but first tell Matlab we're using parallel computing tools by using the *matlabpool* command, and then using *parfor* instead of *for* to create a parallel for loop:

```
matlabpool open local 2  % this tells Matlab I'm using 2 cores
                         % on my local processor
z = zeros(1,1000000);
tic;
parfor a = 1:1000000
     z(a) = isprime(a);
end
toc
matlabpool close;
%%% Elapsed time is 37.690607 seconds
```

We can similarly distribute the responsibilities for matrices among cores by using the *codistributed* function:

```
matlabpool open local 2  % this tells Matlab I'm using 2 cores
                         % on my local processor
x = 1:1000000;
xd = codistributed(x);
yd = cos(xd);  % yd will also be codistributed between cores
matlabpool close;
```

Finally, we can run code in parallel on the cores using an *spmd...end* block, which stands for single program, multiple data:

```
tic; h = rand(10000,1000); a = toc
    a = 34.61

clear h;
matlabpool open local 2;
spmd; tic; h = rand(10000,1000); a = toc; end;
matlabpool close;
    a(1) = 0.2541
    a(2) = 0.2885
```

However, note that starting up the parallel toolbox (matlabpool, spmd, and similar commands) is INTENSELY SLOW right now. Hopefully, they'll fix that in later versions of Matlab. Even as it is now, though, if you have a lot of number crunching to do in the middle of your program and have a multi-core processor, consider using parallel computing tools.

# 2 Recursion

There's a very interesting technique in computer programming called recursion, in which you write a function that calls a variation of itself to get the answer. Eventually, this string of functions calling itself stops at some base case, and the answers work their way back up. Of course, in order to understand recursion, you must understand recursion. (That's a joke).

As a first example, let's consider calculating a factorial. If you want to calculate $n!$ (n factorial), you could write this as $n! = n * (n-1)!$. In other words, **"n factorial"** is actually just **n times "n-1 factorial"**. Thus, we've defined the factorial of n using factorial again. In other words, "to calculate the factorial (of n), calculate the factorial (of n-1)."

Eventually, you need to get down to a base case. What should your base case be? Well, 1! is 1, so make that your base case. So how do you write this as Matlab code? Often, you'll include the base cases first.

```
function y = myfactorial(x)
if(x==1)
   y = 1;
else
   y = x * myfactorial(x-1);
end
```

```
function y = myfactorial(x)
y = 1;
for z = x:-1:1
     y = y*z;
end
```

Why do we use recursion? Well, it's sometimes very simple to define a problem recursively, particularly when creating distributed algorithms in networks, or performing searches, or that sort of thing. Pretty much all of the problems we're showing you in lecture can be done about as easily with iterative (performing something over and over again) algorithms using loops, but not all algorithms can easily be written iteratively.

Let's now look at how you can use recursion to calculate a term in the Fibonacci sequence. Recall that the first two terms of the Fibonacci sequence are 1, and all other terms are the sum of the previous two terms. We'll make $F(1) = 1$ and $F(2) = 1$ our base cases. Those are the cases where we trivially know the answer!

If you wanted to find the fifth term of the Fibonacci sequence, F(5), you could say that:

```
F(5) = F(4) + F(3).
So now, we need to know what F(4) and F(3) are.
F(4) = F(3) + F(2).
So now, we need to know what F(3) is.  F(2) is 1, by the base case.
F(3) = F(2) + F(1).
Nice, F(2) = 1, and F(1) = 1, by our base cases.

Thus, we work backwards:
F(3) = F(2) + F(1) = 1 + 1 = 2.
F(4) = F(3) + F(2) = 2 + 1 = 3.
F(5) = F(4) + F(3) = 3 + 2 = 5.
```

In English, we've essentially said that the "nth fibonacci number is the (n-1)st fibonacci number plus the (n-2)nd fibonacci number." Again, we've defined a fibonacci number in terms of... well, other fibonacci numbers. We keep calling our Fibonacci number function for smaller and smaller numbers until we got to the base cases. We define our base cases as the first and second fibonacci number— they're both 1, which we know pretty trivially. Then, we worked backwards, filling in the blanks. So how do you write this as Matlab code? Again, include the base cases first.

```
function y = FibonacciN(n)
if(n==1 | n==2)
   y = 1;
else
   y = FibonacciN(n-1) + FibonacciN(n-2);
end
```

```
function y = FibonacciN(n)
f(1) = 1;
f(2) = 1;
for z = 3:n
     f(z) = f(z-1) + f(z-2);
end
y = f(n);
```

## 2.1 Recursion- Reverse a String

Let's write a recursive function that takes a string of characters (or a vector) and returns it reversed! Our base case, which is the one for which we know a trivial answer, is when we only have one character. Just return that! (You can also define the base case as the case where you have the empty string, which has some slight advantages). Otherwise, we'll return the flipped version of the second through last characters (by recursively calling *flipit*), with the current first character on the end.

```
function out = flipit(s)
if(length(s)==1)
    out = s;
else
    out = [flipit(s(2:(end)))   s(1)];
end
```

```
function out = flipit(s)
L = length(s);
for z = L:-1:1
     out(L-z+1) = s(z);
end
```

## 2.2 Recursion- Just Print a String Reversed

Now, Let's write a recursive function that takes a string of characters (or a vector) and prints it out, reversed! Our base case, which is the one for which we know a trivial answer, is when we only have one character. Just print that! Otherwise, we'll ask Matlab to keep recursively calling *flipit2*. After calling *flipit2* for the smaller value (and waiting for it to return an answer), we'll print out the character on the end.

```
function [ ] = flipit2(x)
if(length(x)==1)
    fprintf('%s',x);
else
    flipit2(x(2:end));
    fprintf('%s',x(1));
end
```

```
function [ ] = flipit2(x)
for z = length(x) : -1 : 1
     fprintf('%s',x(z))
end
fprintf('\n');
```

## 2.3 Recursion- Find Zero Crossing

Let's say we have a function that we know crosses the x-axis exactly once between x = a and x = b. How can we find the x such that f(x) = 0 recursively? Well, we'll keep dividing the interval in half and looking for a 'sign change,' which means that y goes from positive to negative or vice versa within that interval.

To formalize this notion into code, first note that the midpoint between a and b is $x = \frac{a+b}{2}$. Now, compare f(a) with $f(\frac{a+b}{2})$. If these values have different signs (one is positive and one is negative), then the 'zero-crossing' is between a and $\frac{a+b}{2}$. However, if they both have the same sign, the 'zero-crossing' must be betweeen $\frac{a+b}{2}$ and b. In either case, we've eliminated half of the x points. We keep repeating this same process until we're looking at a lower bound and upper bound for our interval that are sufficiently close together, say within 0.001 of each other. At that point, we know the correct answer with at most 0.001 error!

```
function [v] = findzero(f,a,b)
% recursively finds x within 0.001 of the correct answer
% such that f(x) = 0.  Output v is [lowerBound upperBound]
% assumes that f only crosses zero once between x=a and x=b
midpt = (a+b)/2;
if(feval(f,a)==0) % endpoint is answer
    v = [a a];
    return
elseif(feval(f,b)==0)  % endpoint is answer
    v = [b b];
    return
elseif(feval(f,midpt)==0)  % midpoint is answer
    v = [midpt midpt];
    return
elseif((b-a)<.001) % within tolerance
    v = [a b];
    return
elseif(sign(feval(f,a)) == sign(feval(f,midpt)))
    v = findzero(f,midpt,b); % looks for crossing in latter half
else
    v = findzero(f,a,midpt); % looks for crossing in first half
end
```

We can test this by creating the anonymous function `z = @(x) (x-5).^2-3` and then running `findzero(z,0,5)`

## 2.4 Recursion- Stars Example

Recall the stars problem we solved with nested loops, where your output looks like:

*
**
***
etc...

Let's write a recursive version of this example.

```
function y = stars(x)
if(x==1)
    y = '*';
else
    y = [ stars(x-1) '*' ];
end
fprintf('%s\n',y);
```

You would call this by typing something like *stars(12);...* don't forget the semicolon, or you'll display ans = (the last set of stars). Now, think about why this works... each time we call stars( ), we'll eventually output something (y) and print something (also y). Because we keep calling stars(x-1), nothing gets printed until we get to the base case, x=1, at which point one star is printed. stars(1) returns one star to stars(2), which adds on an extra star and prints that out. Now, stars(2) returns 2 stars to stars(3), which adds on an extra star and prints that out.

```
function y = stars(x)
for r = 1:x
    for c = 1:r
         fprintf('*');
    end
    fprintf('\n');
end
```

## 2.5 Hardcore Recursive Stars

For a quick detour, let's write a recursive version of the stars algorithm that goes from a lot of stars to 1 star in successive lines. In other words, this is the backwards version of the stars problem. We'll attack this problem recursively by having the function call itself for successively smaller and smaller inputs. When we get to 1, the base case, we return a star followed by a newline. As that returns to stars(2), we just take what stars(1) return (up to the closest newline), add a star, and then append what stars(1) returned. We follow this pattern, taking the output added by stars(x-1) beyond what stars(x-2) had, adding a star up front, and then appending the output of stars(x-1):

```
function y = stars(x)
if(x==1)
    y = ['*' char(13)];
else
    z = stars(x-1);
    L = min(find(z==char(13)));
    y = ['*' z(1:L) z];
end
```

## 2.6 Recursive Sudoku

Yes, I clearly hate you and want to remind you of Sudoku. Our recursive idea here is that we'll try to insert a number that works. Once we find a number that works, we'll recursively call our function for this new puzzle. Our base case will be having no numbers to return, which means we're done, and we'll send back the puzzle. Otherwise, if we get stuck, we send back the empty matrix, which is our signal that we ran into a dead end.

```
function out = sudokurecursive(x)
[zr zc] = find(x==0);
if(length(zr)==0)
    out = x;
    return;
end
out = [];
for guess = 1:9
    x(zr(1),zc(1)) = guess;
    currentgroup = x((3*(ceil(zr(1)/3))-2):(3*(ceil(zr(1)/3))),
                          (3*(ceil(zc(1)/3))-2):(3*(ceil(zc(1)/3))));
    if(sum(x(zr(1),:)==x(zr(1),zc(1)))~=1 | sum(x(:,zc(1))==x(zr(1),zc(1)))~=1 |
                           sum(currentgroup(:)==x(zr(1),zc(1)))~=1)
        continue
    end
    out = sudokurecursive(x);
    if(~isempty(out))
        return
    end
end
```

# 3 Progressively Slower Audio

For the final example in this course, we're going to return to our Beyonce obsession and show how different elements of Matlab can be used to progressively slow down a song. Our idea will be to read in the audio file, break it into parts, and then slow down each part by an increasing amount.

How do we slow down each part? Recall that digital audio is merely a series of samples, or the amplitudes of the sound wave taken at evenly spaced intervals. Generally, for CD quality sound (44.1 kHz, or 44,100 samples per second), the sample number can be the x axis and the amplitudes of the samples can be the y axis. Thus, for each second, the x points would be 1:44100. Now, we're going to cheat. We'll pretend that we're not looking at samples 1:44100, but instead samples linspace(1,66150,44100). In other words, we still have 44100 points on our graph, but the x points (time) now go from 1 to 66,150 rather than from 1 to 44,100, stretching out the x axis. However, some of those sample numbers are not integers and we want to have a sample amplitude for every integer from 1 to 66,150. Thus, we use cubic splines to smoothly interpolate and now have samples for every integer point in this stretched range. We add that on to the end of the samples we have so far and move to the next point.

```
[y f] = wavread('single.wav');
intervals = 50;
newy = y(1:(fix(length(y)/10)))';  % unmodified
y = y((fix(length(y)/10)+1):end); % to be modified
L = fix(length(y)/intervals); % length of each interval
for z = 1:intervals
    totalx = fix(L/(1-(z-1)/24)); % the largest x point after stretching
    if(totalx>(L*5)) % Don't go more than 1/5th of the speed
        break
    end
    newy = [newy spline(linspace(1,totalx,L),y(((z-1)*L+1):(z*L)),1:totalx)];
end
newy((end-f):end) = linspace(1,0,length(newy((end-f):end))).*newy((end-f):end); % fade out
p = audioplayer(newy,f);
play(p)
```