aprendtech.com >> blog >> this post

If you have trouble viewing this, try the pdf of this post. You can download the code used to produce the figures in this post.

Optional arguments for Matlab functions

The standard way to handle optional arguments in a Matlab function is to put them at the end of the call list and only include them if you want to change them. This has a lot of problems. First, what do you do if you want to change an argument that is somewhere in the middle of the list but leave the rest unchanged? Some would suggest just putting commas for the unchanged arguments but as far as I know the behavior is undefined. Does this substitute an empty variable '[]' for the other arguments? Also, this makes it hard to read the code. You have to count up the number of commas to figure out which argument is being changed.

Finally, it is hard to program to populate the argument list leading to spaghetti code like this

```
1  function f(arg1,arg2,arg3)
2
3    if nargin < 3
4      arg3 =    'some default';
5    end
6
7    if nargin < 2
8        arg2 = 'another default';
9    end
```

Using cell array to populate arguments

In two posts in her blog (here and here), Loren Shure suggests using a cell array to populate the arguments. Here is the version that handles inputs in the middle of the list:

```
function y = somefun2AltEmptyDefs(a,b,varargin)
% Some function that requires 2 inputs and has some optional inputs.

% only want 3 optional inputs at most
numvarargs = length(varargin);
if numvarargs > 3
    error('myfuns:somefun2Alt:TooManyInputs', ...
        'requires at most 3 optional inputs');
end

% set defaults for optional inputs
optargs = {eps 17 @magic};

% skip any new inputs if they are empty
newVals = cellfun(@(x) ~isempty(x), varargin);

% now put these defaults into the valuesToUse cell array,
% and overwrite the ones specified in varargin.
optargs(newVals) = varargin(newVals);
% or ...
% [optargs{1:numvarargs}] = varargin{:};

% Place optional args in memorable variable names
[tol, mynum, func] = optargs{:};
```

This is nice but you still have the problem of counting commas to figure out which argument is changed. Also, you would have to put error checking the arguments into separate code, which separates the processing into two pieces leading to possible inconsistencies between the parts of the code.

Named optional arguments

My preference is to use the named optional arguments style shown below:

```
retval = f(reqarg1,'optionalarg2',arg2,'optionalarg3',arg3);
```

This style has possible required arguments first followed by possible pairs of 'optional_argument_name',argum
A single argument_name can be used for flags. This makes the code much more readable and is more flexible than the comma method. But how to implement it?

Using inputParser

One possibility is a recent addition to Matlab called $inputParser$. Here is a link to description on the Mathworks site. I have not used it but from the description I see several problems. First, validating the arguments requires specifying an additional function, either an anonymous function, which has limited capability, or a separate subfunction that again separates the error checking from the processing, which I think is error-prone. Another problem is that it is a recent addition so it limits the users of your code to those with versions that support it. Finally, it is a black box and Mathworks proprietary. I could not find the source code for it.

Using the switch approach

We finally come to my favorite, which I use in many of the functions I distribute with this blog. This implements the named optional arguments style in an easy to understand block of code that processes and validates the arguments at the same time. An example of its use is shown below for my $CTrecon$ function.

```
1     % handle the optional arguments---first process the required arguments
2  nreqargs = 1;
3  if nargin<nreqargs
4     fprintf('syntax: [img,H] = CTrecon(prj,varargin)');
5     error('%d is too few arguments',nargin);
6  end
7    % process the required arguments and define the defaults
8  [nlines, nangles] =size(prj);
9  angles = linspace(0,180,nangles+1);
10 angles = angles(1:(end-1) ); % make sure angles do not wrap around
11 interpolation = 'linear';
12 filter_type = 'Ram-Lak';
13 freq_cutoff = 1;
14
15     % populate the optional arguments if present
16 if(nargin>nreqargs)
17   i=1;
18   while(i<=size(varargin,2))
19      switch lower(varargin{i})
20      case 'angles';    angles=varargin{i+1};
21         if numel(angles) ~= nangles
22            error('CTrecon: length of angles array %d must equal size(prj,2) %d
23         end
24          i=i+1;
25      case 'interpolation';          interpolation=varargin{i+1};
26          if ~ischar(interpolation)
27            error('CTrecon: interpolation parameter must be a string');
28         end
29          i=i+1;
30      case 'filter_type';          filter_type=varargin{i+1};
31          if ~ischar(interpolation)
32            error('CTrecon: filter_type must be a string');
33         end
34          i=i+1;
35      case 'freq_cutoff';          freq_cutoff=varargin{i+1};
36          if (freq_cutoff<0)  || (freq_cutoff>1)
37            error('CTrecon: freq_cutoff %g must be between 0 and 1', freq_cutof
38         end
39          i=i+1;
40     otherwise
41       error('CTrecon: Unknown argument %s given',varargin{i});
42      end
43      i=i+1;
44   end
45 end
```

The $CTrecon$ function has one required argument and four optional arguments. The first part from lines 1-14 checks the required arguments and defines the default values. The while loop starting at line 18 goes through the additional arguments passed to the

function. The switch statement processes the optional arguments one per case. Each case block loads the argument from $varargin$ and then has user defined validation of the argument. These are together so there is no problem with consistency. The code is straightforward and can be readily modified by the user. For example to require case sensitive arguments, do not use the $lower()$ function in line 19 and adjust the string for each case block. This approach does not use any proprietary functions so it will work with almost all versions of Matlab and Octave.

The example above did not include a switch variable to handle a logical condition such as a $verbose$ option. In that case implement the variable without the $i = i + 1$ as shown below:

```
1  verbose = false;
2
3  ...
4  while(i<=size(varargin,2))
5       switch lower(varargin{i})
6
7  case 'verbose';    verbose = true;
8  case ...
```

Conclusion

I have described several different methods to handle optional arguments. In general, I think the method should be easy to understand so future users of your code can modify and extend it. It should allow you to test and validate the inputs so you can give feedback to the user for improper inputs. Finally, the processing and the validation should be done at the same time to reduce the chance of inconsistencies when the code is modified. I think the switch method satisfies these criteria and I use it in most of my functions.

—Bob Alvarez

Last editedMarch 15, 2013

References