Complicated Sinusoidal Signals

by Mark A. Yoder with some tweaking by others

Objectives

A variety of interesting waveforms can be expressed as sums of complex exponentials of different frequencies. The pulse trains used in communication systems, speech waveforms, and the waveforms produced by musical instruments can be modelled in this way. In this lab we will explore some analytical techniques for generating waveforms by summing complex exponentials.

The two main objectives of this lab are:

1. Learn how to write functions in MATLAB, and
2. Write a function that will sum several complex exponential terms.

Background

A single sinusoid can be written as the sum of two complex exponentials, like this:

$$\begin{aligned} x(t) &= A\cos(2\pi f_0 t + \phi) \\ &= \frac{A}{2}e^{j\phi}e^{j2\pi f_0 t} + \frac{A}{2}e^{-j\phi}e^{-j2\pi f_0 t} \end{aligned} \tag{1.1}$$

When only a single frequency is involved, it is convenient to drop the time-varying exponential term, and work with the phasor $Ae^{j\phi}$. In this lab we will synthesize more complicated waveforms composed of sums of complex exponential signals, each with a different frequency f_k:

$$x(t) = \sum_{k=-N}^{N} c_k e^{j2\pi f_k t}, \tag{1.2}$$

where $c_k = A_k e^{j\phi_k}$ is a complex phasor amplitude. In this case we cannot drop the time-varying terms $e^{j2\pi f_k t}$. If $f_{-k} = -f_k$, and $c_{-k} = c_k^* = A_k e^{-j\phi_k}$, then pairs of complex exponentials combine to give sinusoids like this:

$$\begin{aligned} c_k e^{j2\pi f_k t} + c_{-k} e^{j2\pi f_{-k} t} &= A_k e^{j\phi_k} e^{j2\pi f_k t} + A_k e^{-j\phi_k} e^{-j2\pi f_k t} \\ &= 2A_k \cos(2\pi f_k t + \phi_k) \end{aligned}$$

In this case $x(t)$ will be a real-valued signal.

Harmonically Related Sinusoids: Fourier Series

There is an important special case where $x(t)$ is the sum of $2N + 1$ complex exponentials whose frequencies f_k are all multiples of one "fundamental" frequency f_0:

$$f_k = kf_0 \quad \text{(Harmonic frequencies).}$$

The sum of $2N + 1$ complex exponentials given by Eq. (1.2) becomes

$$x(t) = \sum_{k=-N}^{N} c_k e^{j2\pi k f_0 t} \tag{1.3}$$

in this case. If $c_{-k} = c_k^*$, then each positive-frequency complex exponential combines with the corresponding negative-frequency complex exponential to give a sinusoid. Then $x(t)$ will be the sum of a constant and N harmonically related sinusoids. In general, a signal $x(t)$ given by (1.3) is periodic with period $T_0 = 1/f_0$. The frequency f_0 is called the *fundamental frequency*, and T_0 is called the *fundamental period*. The right hand side of Eq. (1.3) is called a *Fourier series.*

Pre-Lab

The goal is to give you a jump start in writing functions in MATLAB. Read the "MATLAB Programming Tips" and then do the following before coming to lab. Record the results in your lab notebook and hand in a photocopy of your pre-lab at the start of the class before lab.

(a) Find the mistake(s) in the following functon:

```
function x=cosgen(f,dur)
%COSGEN Function to generate a cosine wave
%  usage:
%    x=cosgen(f,dur)
%    f=desired frequency
%    dur=duration of the waveform in seconds.
%
t=[0:1/(20*f):dur]; %gives 20 samples per period
y=abs(2*pi*f*t);
```

(b) Find the mistake(s) in the following function:

```
function [sum,prod]=sumprod(x1,x2)
%SUMPROD Function to add and multiply two complex numbers
%  usage:
%    [sum,prod]=sumprod(x1,x2)
%    x1=a complex number
%    x2=another complex number
%    sum=sum of x1 and x2
%    prod=product of x1 and x2
%
sum=z1+z2;
prod=z1*z2;
```

(c) Write a function that performs the same task as the following without using a `for` loop.

```
function Z=expand(x,ncol)
%EXPAND Function to generate a matrix Z with identical columns
%  equal to an input vector x
%  usage:
%    Z=expand(x,ncol)
%    x=thi input vector that comprises the identical columns of Z
%    ncol=the number of desired columns
%
x=x(:); % this turns the input vector x into a column vector
for k=1:ncol
  Z(:,k)=x;
end
```

(d) Write a function that performs the same task as the following without using a `for` loop.

```
function Z=replacez(A)
%REPLACEZ Function that replaces the positive elements of a matrix with 1
%  usage:
%    Z=replacez(A)
%    A=input matrix whose positive elements are to be replaced with 1
```

```
%
[M,N]=size(A);
for i=1:M
  for j=1:N
    if A(i,j)>0
      Z(i,j)=1;
    else
      Z(i,j)=A(i,j);
    end
  end
end
```

MATLAB hint: The MATLAB logical operators may be helpful in completing this exercise. Use `help` to understand what these commands do.

Now test part d to demonstrate that it works. Record the matrices `A` and `Z` in your lab notebook.

Procedure

Synthesis of Waveforms

Write an m-file that will synthesize a waveform in the form of Eq. (1.2). Write the function without using `for` loops. The first few statements of the m-file should look like:

```
function [x,t]=sumexp(f,c,fs,dur)
%SUMEXP Function to synthesize a sum of complex exponentials
%  usage:
%    [x,t]=sumexp(f,c,fs,dur)
%    f=vector of frequencies (values can be negative or positive)
%    c=vector of complex amplitudes (phasors):  Amp*e^(j*phase)
%    fs=the sampling rate in Hz
%    dur=total time duration of signal
%    x=the returned waveform x(t)
%    t=the time array, returned to facilitate plotting
%
%    Note:  f and c must be the same length.  c(1) corresponds to
%      frequency f(1), c(2) corresponds to frequency f(2), etc.
%
```

In order to use this m-file to synthesize a Fourier series, you would simply choose the entries in the frequency vector to be integer multiples of the desired fundamental frequency. Try the following test and plot the result:

```
[x,t]=sumexp([-60 -40 -20 20 40 60],[-1 1 -1 -1 1 -1],200,1.5);
```

Test Waveforms

Each of the following waveforms can be synthesized with a simple call to the function `sumexp`. Plot a short section of the signal to observe its characteristic shape. If your computer can play sounds, change the fundamental to 1 kHz and listen to 0.8 seconds of the signal using `sound(x,fs)`.

Waveform 1

Try your m-file with the fundamental $f_0 = 25$ Hz, $f_k = kf_0$, and

$$c_k = \left\{ \begin{matrix} \dfrac{j}{k} & k \text{ an odd integer} \\ 0 & k \text{ an even integer} \end{matrix} \right\} \text{ for } k \geq 0, \text{ and } c_{-k} = c_k^*. \qquad (1.4)$$

Specify the duration to get three periods of the waveform.

Make plots for three different cases: N=5, 10, and 25, where there are $2N + 1$ complex exponentials (including terms of zero amplitude). Use a three-panel subplot to show all three signals together. Explain how the period of the synthesized waveform is related to the fundamental frequency.

What wave shape is approximated with this sum of complex exponentials as you add more terms? Explain what happens as $N \to \infty$. Do you notice anything unusual about the waveform as it converges?

Waveform 2

Now try the coefficients

$$c_k = \left\{ \begin{array}{ll} \dfrac{1}{k^2} & k \text{ an odd integer} \\ \dfrac{2}{k^2} & k = 2, 6, 10, 14, \text{ etc.} \\ 0 & k \text{ a multiple of 4} \end{array} \right\} \text{ for } k \geq 0, \text{ and } c_{-k} = c_k^*. \qquad (1.5)$$

Choose the fundamental frequency to be $f_0 = 25$ Hz. Compute the signal for three cases: N=5, 10, and 25, and plot all three functions together using a three-panel subplot. What wave shape is approximated with this sum of complex exponentials? Explain how the period of the synthesized waveform is related to the fundamental frequency.

Report

Record the results of all of your work in one partner's lab notebook. Tape any printout (graphs, for example) into the notebook as specified in the Lab Manual for the course. (You must provide your own tape.) Be sure that all members of your lab group sign the lab notebook, and hand the notebook in at the end of lab.

MATLAB Programming Tips

This section presents a few programming tips that should help improve your MATLAB programs. For more ideas and tips, list some of the functions (m-files) in the toolboxes of MATLAB using the `type` command. Copying the style of other programmers is always an efficient way to improve your own knowledge of a computer language. In the hints below, we discuss some of the most important points involved in writing good MATLAB code. These comments assume that you are an experienced programmer and at least an intermediate user of MATLAB.

Matrix Operations

The default notation in MATLAB is matrix. Therefore, some confusion can arise when trying to do pointwise operations. Take the example of multiplying two matrices `A` and `B`. If the two matrices have compatible dimensions, then `A*B` is well defined. But suppose that both are 5×8 matrices ant that we want to multiply them together element by element. In fact, we can't do matrix multiplication between two 5×8 matrices. To obtain point-wise multiplication we use the "point-star" operator `A.*B`. In general, when "point" is used with another arithmetic operator it modifies that operator's usual matrix definiton to a point-wise one. Thus we have `./` and `.^` for point-wise division and exponentiation. For example, `x=(0.9).^[0:49]` generates an exponential of the form a^n for $n = 0,1,2,\ldots,49$.

A Review of Matrix Multiplication

Recall that before a matrix *A* can multiply another matrix *B*, the number of columns in *A* must equal the number of rows in *B*. For example, if *A* is *m*×*n*, then *B* must be *n*×*l*. Obviously, matrix multiplication is not commutative, as the product $B \cdot A$ is undefined for $l \neq m$.

To generate the first element in the product of two matrices *A* and *B*, simply take the first row of *A* and multiply it *point by point* with the first column of *B*, then sum. For example, if

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}, \text{ and } B = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{bmatrix},$$

then the first element of $C = A \cdot B$ would be

$$c_{1,1} = a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1}.$$

The second element $c_{1,2}$ of *C*, is found by taking the first row of *A* and multiplying it point by point with the *second* column of *B* and then summing. The element $c_{2,1}$ is found by taking the second row of *A* and multiplying it point by point with the first column of *B*, then summing. Finally, element $c_{2,2}$ is found by multiplying the second row of *A* point by point with the second column of *B* and then summing. The result would be:

$$\begin{aligned} C &= \begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix} \\ &= \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} + a_{1,3}b_{3,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} + a_{2,3}b_{3,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} + a_{2,3}b_{3,2} \end{bmatrix}. \end{aligned}$$

Some useful results are as follows:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_1 & a_1 & a_1 \\ a_2 & a_2 & a_2 & a_2 \\ a_3 & a_3 & a_3 & a_3 \end{bmatrix}$$

and

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = a_1 + a_2 + a_3 + a_4.$$

There may be times when you want to multiply matrices element by element, that is $c_{i,j} = a_{i,j} \cdot a_{i,j}$. Obviously the matrices would have to be the same size to do this. In MATLAB this is accomplished using the .* operation. For example if *A* and *B* are both 2×2:

$$C = A.*B = B.*A = \begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1}b_{1,1} & a_{1,2}b_{1,2} \\ a_{2,1}b_{2,1} & a_{2,2}b_{2,2} \end{bmatrix}$$

Also remember in MATLAB:

$$\exp(A) = \begin{bmatrix} \exp(a_{1,1}) & \exp(a_{1,2}) & \cdots & \exp(a_{1,n}) \\ \exp(a_{2,1}) & \exp(a_{2,2}) & \cdots & \exp(a_{2,n}) \\ \vdots & \vdots & \ddots & \vdots \\ \exp(a_{m,1}) & \exp(a_{m,2}) & \cdots & \exp(a_{m,n}) \end{bmatrix}$$

and

$$\cos(A) = \begin{bmatrix} \cos(a_{1,1}) & \cos(a_{1,2}) & \cdots & \cos(a_{1,n}) \\ \cos(a_{2,1}) & \cos(a_{2,2}) & \cdots & \cos(a_{2,n}) \\ \vdots & \vdots & \ddots & \vdots \\ \cos(a_{m,1}) & \cos(a_{m,2}) & \cdots & \cos(a_{m,n}) \end{bmatrix}.$$

Avoid FOR Loops

There is a temptation among experienced programmers to use MATLAB in the same fashion as a high-level language like C. However, this leads to very inefficient programming whenever `for` loops are used to do operations over the elements of a vector, e.g. summing the elements of a vector. Instead, you must look for the MATLAB functions or vector operations that will do the same operation with a function call - in the case of summing there is a MATLAB function called `sum`.

A `for` loop is extremely inefficient in MATLAB because MATLAB is an interpreted language. Macro operations such as matrix multiplies are about as fast as micro-operations such as incrementing an index, because the overhead of interpreting the code is needed in both cases. The bottom line is that `for` loops should only be used as a last resort, and then probably only for

control operations, not for computational reasons. More than likely, 90% of the `for` loops used in ordinary MATLAB programs could be replaced with equivalent, and faster, vector code.

Writing a MATLAB Function

You can write your own functions and add them to the MATLAB environment. These functions are just a type of m-file and are created as an ASCII file via a text editor. The first word in the m-file must be the keyword `function` to tell MATLAB that this file is to be treated as a function with arguments. On the same line as the word `function` is the calling template that specifies the input and output arguments of the function. The filename for the m-file must end in `.m` and the filename will become the name of the new command for MATLAB. For example, consider the following file:

```
function y=last(x,L)
%LAST get last L points of x
%  usage:
%    y=last(x,L)
%  where:
%    x=input vector
%    L=number of points to get
%    y=output vector
N=length(x);
if(L>N)
  error('input vector too short')
end
y=x((N-L+1):N);
```

If this file is called `last.m`, the operation could be invoked from the MATLAB command window by typing

```
a=last(rand(11,1),7);
```

The effect will be to generate a vector of eleven random numbers and then get the last seven. Note that the variable names inside the function are all local variables and they disappear after the function completes. The argument names `y`,`x`, and `L` are dummy names which are passed values when the function is invoked.

Creating Your Own Functions

We will show that most functions can be written according to a standard format. Consider a clip function m-file that takes two input arguments (a signal vector and a scalar threshold) and returns an output signal vector. You can use an editor to create an ASCII file `clip.m` that contains the following statements:

Eight lines of comments at the beginning of the function will be the response to `help clip`.

First step is to figure out matrix dimensions of `x`

Input could be row or column vector

Since `x` is local, we can change it without affecting the workspace.

Create output vector

```
function y=clip(x,Limit)
%CLIP saturate mag of x[n] at Limit
%
%  usage: y=clip(x,Limit)
%
%    x=input signal vector
%    Limit=limiting value
%    y=output vector after clipping
%
[nrows ncols]=size(x);
if (ncols~=1 & nrows ~=1) % neither
  error('CLIP: input not a vector')
end
Lx=max([nrows ncols]);    % length
for n=1:Lx              % Loop over entire vector
  if(abs(x(n))>Limit)
    x(n)=sign(x(n))*Limit; % saturate
  end
end
y=x;                    % copy to output vector
```

We can break down the mfile `clip.m` into four elements:

1. *Definition of Input/Output:* Function m-files must have the word `function` as the very first thing in the file. The information that follows `function` on the same line is a declaration of how the function is to be called and what arguments are to be passed. The name of the function should match the name of the m-file; if there is a conflict, it is the name of the m-file which is known to the MATLAB command environment.

   Input arguments are listed inside the parentheses following the function name. Each input is a matrix. The output argument (a matrix) is on the left side of the equals sign. Multiple output arguments are also possible, e.g., notice how the function `size(x)` in `clip.m` returns the number of rows and number of columns into separate output variables. Square brackets surround the list of output arguments. Finally, observe that there is no explicit return of the outputs; instead, MATLAB returns whatever value is contained in the output matrix when the function completes. The MATLAB function `return` just leaves the function, it does not take an argument. For `clip` the last line of the function assigns the clipped vector to `y`.

   The essential difference between the function m-file and the script m-file is dummy variables versus permanent variables. The following statement creates a clipped vector `wclipped` from the input vector `win`.

   ```
   >> wclipped=clip(win,0.9999);
   ```

   The arrays `win` and `wclipped` are permanent variables in the workspace. The temporary arrays created inside `clip` (i.e., `y`, `nrows`, `ncols`, `Lx`, and `i`) exist only while `clip` runs; then they are deleted. Furthermore, these variable names are local to `clip.m` so the name `x` could also be used in the workspace as a permanent name. These ideas should be familiar to anyone with experience using a high-level computer language like C or PASCAL.

2. *Self Documentation:* A line beginning with the % sign is a comment line. The first group of these in a function is used by MATLAB's help facility to make m-files automatically self-

documenting. That is, you can now type `help clip` and the comment lines from *your* m-file will appear on the screen as help information!! The format suggested in `clip.m` follows the convention of giving the function name, its calling sequence, definition of arguments and a brief explanation.

3. *Size and Error Checking:* The function should determine the size of each vector/matrix that it will operate on. This information does not have to be passed as a separate input argument, but can be extracted on the fly with the `size` function. In the case of the `clip` function, we want to restrict the function to operating on vectors, but we would like to permit either a row ($1 \times L$) or a column ($L \times 1$). Therefore, one of the variables `nrows` or `ncols` must be equal to one; if not we terminate the function with the bail-out function `error` which print a message to the command window and quits the function.
4. *Actual Function Operations:* In the case of the `clip` function, the actual clipping is done by a `for` loop which examines each element of the `x` vector for its size versus the threshold `Limit`. In the case of negative numbers, the clipped value must be set to `-Limit`, hence the multiplication by `sign(x(n))`. This assumes that `Limit` is passed in as a positive number, a fact that might also be tested in the error testing phase.

Avoid FOR Loops

Since MATLAB is an interpreted language, certain common programming habits are intrinsically inefficient. The primary one is the use of `for` loops to perform simple operations over an entire matrix or vector. *Whenever possible* you should try to find a vector function (or the composition of a few vector functions) that will accomplish the same result rather than writing a loop. For example, if the operation were summing up all the elements in a matrix, the difference between calling `sum` and writing a loop that looks like FORTRAN code can be astounding - the loop is unbelievably slow because of the interpretive nature of MATLAB. Consider the following three methods for matrix summation:

Double loop needed to index all matrix entries

```
[Nrows,Ncols]=size(x)
xsum=0.0;
for m=1:Nrows
  for n=1:Ncols
    xsum=xsum+x(m,n);
  end
end
```

`sum` acts on a matrix to give the sum down each column

```
xsum=sum(sum(x));
```

`x(:)` is a vector of all elements in the matrix

```
xsum=sum(x(:));
```

The last two methods rely on the built-in functon `sum` which has different characteristics depending on whether its argument is a matrix or a vector (this is called “operator overloading”). To get the third (and most efficient) method, the matrix `x` is converted to a column vector with the colon operator. Then one call to `sum` will suffice.

Repeating Rows or Columns

Often it is necessary to form a matrix by repliating a value in the rows or columns. If the matrix is to have all the same values, then functions such as `ones(M,N)` and `zeros(M,N)` can be used. But when you want to replicate a column vector `x` to create a matrix that has eleven identical colunms, you can avoid a loop by using the outer-product matrix multiply operation. The following MATLAB code fragment will do the job:

```
X=x*ones(1,11)
```

If `x` is a length *L* column vector, then the matrix `X` formed by the outer product is *L*×11. Try it!

Vectorizing Logical Operations

The `clip` function offers a different opportunity for vectorization. The `for` loop in that function contains a logical test and might not seem like a candidate for vector operations. However, the logical operators in MATLAB apply to matrices. For example, a greater-than test applied to a 3×3 matrix returns a 3×3 matrix of ones and zeros.

```
>> x=[1  2 -3
      3 -2  1
      4  0 -1];
>> mx=x>0 % check the greater-than condition
mx =
     1 1 0
     1 0 1
     1 0 0
>> y=mx.*x % multiply by masking matrix
y =
     1 2 0
     3 0 1
     4 0 0
```

The zeros mark where the condition was false; the ones denote true. Thus, when we multiply `x` by the masking matrix `mx`, we get a result that has all negative elements set ot zero. Note that the entire matrix has been processed without using a loop.

Since the saturation done in `clip.m` requires that we change the large values in `x`, we can implement the `for` loop with three array multiplications. This leads to a vectorized saturation operator that works for matrices as well as vectors:

```
y=x.*(abs(x)<=Limit)+Limit*(x>Limit)-Limit*(x<Limit);
```

Three different masking matrices are needed to represent the three cases of negative saturation, positive saturation, and no action. The additions correspond to the logical OR of these cases. The number of arithmetic operations needed to carry out this statement is 3*N* multiplications and 2*N* additions, where *N* is the total number of elements in `x`. On the other hand, the statement is interpreted only once.

Exercise

Write a variation of the `clip` function with three input arguments: the matrix `x` and two thresholds, an upper limit and a lower limit.

Creating an Impulse

Another simple example is given by the following trick for creating an impulse signal vector:

```
n=[-20:80];
impulse=(n==0);
```

This result could be plotted using `stem(n,impulse)`. In some sense, this code fragment is perfect because it captures the essence of the mathematical formula that defines the impule as only existing when $n = 0$.

$$\delta[n] = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases}$$

The Find Function

An alternative to masking is to use the `find` function. This is not necessarily more efficient, it just gives a different approach. The `find` function will determine the list of indices in a vector where a condition is true. For example, `find(x>Limit)` will return the list of indices where the vector is greater than the `Limit` value. Thus we can do saturation as follows:

```
y=x;
jkl=find(y>Limit);
y(find(y>Limit))=Limit*ones(size(jkl));
jkl=find(y<Limit);
y(jkl)=-Limit*ones(jkl);
```

The `ones` function is needed to create a vector on the right-hand side that is the same size as the number of elements in `jkl`.

Seek to Vectorize

The dictum "avoid `for` loops" is not always an easy path to follow, because it means the algorithm must be cast in vector form. We have seen that this is not particularly easy when the loop contains a logical test, but such loops can still be "vectorized" if masks are created for all possible conditions. This does result in extra arithmetic operations, but they will be done efficiently by the internal vector routines of MATLAB, so the final result should still be much faster than an interpreted `for` loop.

Composition of Functions

MATLAB supports the paradigm of "functional programming" which in the language of system theory is equivalent to cascading systems. Consider the following equation which can be implemented with one line of MATLAB code.

$$\sum_{n=1}^{L} \log\left(|x_n|\right)$$

Here is the MATLAB equivalent:

```
sum(log(abs(x)))
```

Programming Style

"May your functions be short and your variable names long." Each function should have a single purpose. This will lead to short simple modules that can be composed together with other functions to produce more complex operations. Avoid the temptation to build super functions with many options and a plethora of outputs. MATLAB supports long variable names - up to thirty-two characters. Take advantage of this feature to give variables descriptive names. In this way, the number of comments littering the code can be drastically reduced. Comments should be limited to help information and documentation of tricks used in the code.

The COLON Operator

One essential part of MATLAB that is needed to avoid `for` loops is the colon toation for selecting parts of matrices. The help for `:` is given below:

```
» help colon

 :  Colon.
    J:K  is the same as [J, J+1, ..., K].
    J:K  is empty if J > K.
    J:D:K  is the same as [J, J+D, ..., J+m*D] where m = fix((K-J)/D).
    J:D:K  is empty if D > 0 and J > K or if D < 0 and J < K.

    COLON(J,K) is the same as J:K and COLON(J,D,K) is the same as J:D:K.

    The colon notation can be used to pick out selected rows, columns
    and elements of vectors, matrices, and arrays.  A(:) is all the
    elements of A, regarded as a single column. On the left side of an
    assignment statement, A(:) fills A, preserving its shape from before.
    A(:,J) is the J-th column of A.  A(J:K) is [A(J),A(J+1),...,A(K)].
    A(:,J:K) is [A(:,J),A(:,J+1),...,A(:,K)] and so on.

    The colon notation can be used with a cell array to produce a comma-
    separated list.  C{:} is the same as C{1},C{2},...,C{end}.  The comma
    separated list syntax is valid inside () for function calls, [] for
    concatenation and function return arguments, and inside {} to produce
    a cell array.  Expressions such as S(:).name produce the comma separated
    list S(1).name,S(2).name,...,S(end).name for the structure S.

    For the use of the colon in the FOR statement, See FOR.
    For the use of the colon in a comma separated list, See VARARGIN.
```

The colon notation works from the idea that an index range can be generated by giving a start, a skip, and then the end. Therefore, a regularly spaced vector of integers is obtained via

```
iii=start:skip:end
```

Without the `skip` parameter, the increment is 1. Obviously, this sort of counting is similar to the notation used in FORTRAN DO loops. However, in MATLAB you can take it one step further by combining it with a matrix. If you start with the matrix `A`, then `A(2,3)` is the scalar element located at the second row and third column of `A`. But you can also pull out a 4×3 submatrix via `A(2:5,1:3)`. If you want an entire row, the colon serves as a wild card, i.e. `A(2,:)` is the second row. You can even flip a vector by just indexing backwards: `x(L:-1:1)`. Finally, it is sometimes necessary to just work with all the values in a matrix, so `A(:)` gives a column vector that is just the columns of `A` concatenated together. More general "reshaping" of the matrix `A` can be accomplished with the `reshape(A,M,N)` function. For example, a 5×4 matrix `B` can be reshaped into a 2×10 matrix via

```
Bnew=reshape(B,2,10)
```

Debugging an M-file

Since MATLAB is an interactive environment, debugging can be done by examining variables in the workspace. Type `help debug` for more information.