2 ZEROS OF FUNCTIONS

Contents

```
% make sure the workspace is clear
if ~exist('__code__','var') ; clear ; end
% reduce needless whitespace
format compact
% reduce irritations (pausing and buffering)
more off
% start a diary (in the actual lecture)
%diary lecture3.txt
```

LESSON SUMMARY

- This lesson explains how to solve a single nonlinear equation in one variable, for example $f(x) = 0$ for some given function $f(x)$.

- There are two key steps in solving an equation for an unknown x:

  1. Create a Matlab function file that returns (outputs) the *error* in the equation as a function of the unknown x. The error is in general the difference between the left- and right-hand sides. So if the equation is $f(x) = 0$, then the error is just function $f(x)$.

  2. Let Matlab function `fzero` find a location where the error is zero, i.e. a solution or root. To do so, you must tell `fzero` what your function is that gives the error in the equation. The function name must be preceded by an @ character. You must also either *(a)* give `fzero` a starting value for x near which it needs to search for the root you want; or *(b)* give `fzero` an x-range in which to search for the root.

- Giving `fzero` a starting value for x is inefficient and can fail. It is much safer to give `fzero` a range in which to search. As long as you ensure that the error is of opposite sign at the two end points of the range you give it, `fzero` is *guaranteed* to find a root. So if you also ensure that the range is small enough that there is only one root inside it, `fzero` will find that root.

- To identify a suitable range (or starting value) for x, you will want to make a plot of your error function versus x. The Matlab `plot` function can make the graph for you if you give it a lot of closely spaced x-values and corresponding error-values. In Matlab, the x-values will form an "array", and so will the error-values.

- Neat, readable, and understandable plots will also need `title`, `legend`, `grid`, `xlabel`, `ylabel`, `set(gca,...,...)`, ... commands as appropriate.

- The plot will then allow you to ballpark the x-values where the error is zero, the roots of the equation. You can then use each ballparked value as a starting value for `fzero`, or identify a suitable range around each ballparked value.
- Function `fzero` can only handle functions of a single variable. (The variable does not have to be called x, but there cannot be more than one of them.) So if you, say, want to solve an equation like $f(x, y) = 0$ for x given a value for y, there is a problem. In Matlab you can fix this problem using the concept of "anonymous functions".
- You will also need to learn to print out the numbers you find in a neat and prescribed format using the Matlab `fprintf` function. Function `fprintf` allows you to specify exactly how you want the number formatted; for example, how many digits behind the decimal point to print.

Key areas of the online book

Before the lecture, in the online book do:

- 3.7 Basic output, `fprintf`: all.
- 3.8 Floating-point formatting in `fprintf`: all.
- 4.1 Introduction to arrays: complete the section.
- 4.2 Row arrays: do PA 4.2.1.
- 4.3 Constructing row arrays: do PA 4.3.1-4.
- 4.8 Functions to create arrays: do PA 4.8.1 and CA 4.8.1.
- 5.5 Functions and 1D arrays: skip the CAs.
- 9.1 Simple plotting: all.

Note: while the online book has a section 18.5 on finding zeros of functions, it is overly mathematical and provides much less explanation of what is going on than I do below. I suggest to stay clear of section 18.5.

THE PROBLEM WE WANT TO SOLVE

The main purpose of this lesson is to explain how to solve a single nonlinear equation in one variable. Of course, for a quadratic equation, i.e.

$$f(x) = ax^2 + bx + c = 0$$

there is a simple formula for the two solutions (or "roots"). This lesson is for the case that you do not know how to solve the equation analytically, or even how many roots there are.

In particular, as an example we will solve the equation

$$\sin(\omega) = -k\omega\cos(\omega)$$

for the unknown ω (or `omega` in Matlab).

It can be shown that the solutions (roots) ω to this equation are the nondimensional frequencies of vibration of a string with one end rigidly attached and the other end flexibly attached. The constant k is a nondimensional stiffness of the flexible attachment.

But you do not have to know all that. All you need to know is how to solve an equation like the one above using Matlab. The above equation does not have an analytical solution. But the Matlab function `fzero` can be used to solve it numerically.

THE ERROR IN THE EQUATION AS A FUNCTION

If we should have that

$$\sin(\omega) = -k\omega\cos(\omega)$$

then if ω is not right, there will be an error in the equation. We can take this error to be the difference between the left- and right-hand sides of the equation:

$$\text{error} = \sin(\omega) + k\omega\cos(\omega)$$

Note that this error is a function of ω. And only if the error is zero do we have a correct value of ω. So, basically we need to find out the values of ω where the above function of ω is zero. Matlab function `fzero` can help us do this.

But to use `fzero`, we must first put the function above in a Matlab function file. To keep it as simple as possible, for now we will assume that k equals 1. A reasonable name for our function is therefore `FreqEqError1` (for "Frequency-Equation Error for $k = 1$"). The *minimal* contents of the corresponding function file `FreqEqError1.m` is then:

```
function errorEq = FreqEqError1(omega)

errorEq = sin(omega) + omega.*cos(omega);

end
```

Note above that proper frequencies are in radians, not degrees. Also note that the point in `.*` is needed since `fzero` may input an array of ω values. And note that the semicolon is also definitely needed; else the function will print out every error it evaluates.

Play a bit with the function

```
% see whether Matlab can see the function
%help FreqEqError1

% for omega=0 the error is zero but there is no sound!
error0=FreqEqError1(0)

% for omega=1 the error is not zero, so omega=1 is _not_
% a frequency of vibration of this string
error1=FreqEqError1(1)

% how about 2?
error2=FreqEqError1(2)

% how about 1.9 or 2.1?
error1p9=FreqEqError1(1.9)
error2p1=FreqEqError1(2.1)
```

```
error0 = 0
error1 =  1.3818
error2 =  0.077004
error1p9 =  0.33205
error2p1 = -0.19697
```

There seems to be a root between 1.9 and 2.1! (Or more precisely, between 2 and 2.1.)

PLOT TO UNDERSTAND THE PROBLEM BETTER

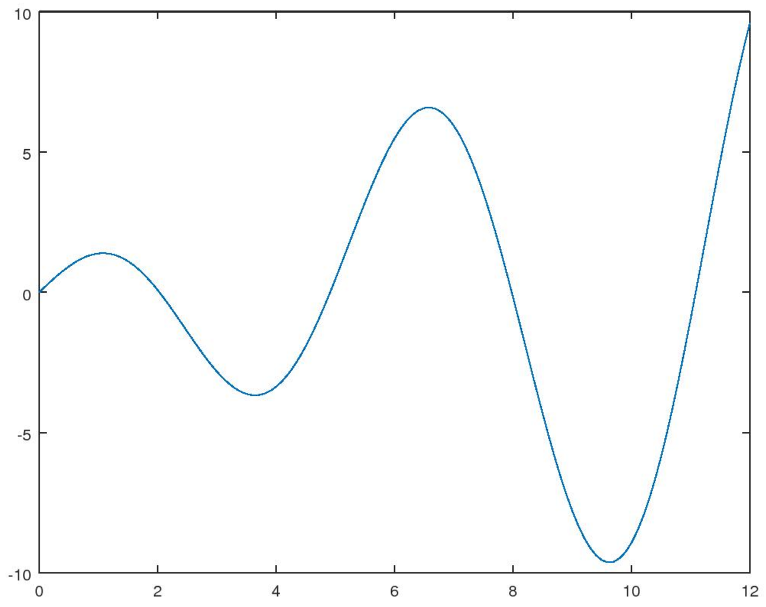
Somehow we must find the locations where function `FreqEqError1` is zero. That is not that straightforward. So maybe we should first plot the function.

Plot the error for 0 < omega < 12

The Matlab 'plot' function can plot a curve if you give it enough points on the curve.

```
% generate 121 omega values between 0 and 12
omegaPlot=[0:0.1:12];

% this makes omegaPlot a row of numbers
%omegaPlot

% a simpler way to do the same thing
omegaPlot=linspace(0,12,121);
%omegaPlot
```

```
% compute the corresponding errors
errorPlot=FreqEqError1(omegaPlot);
%errorPlot

% plot the error versus omega
plot(omegaPlot,errorPlot)
```



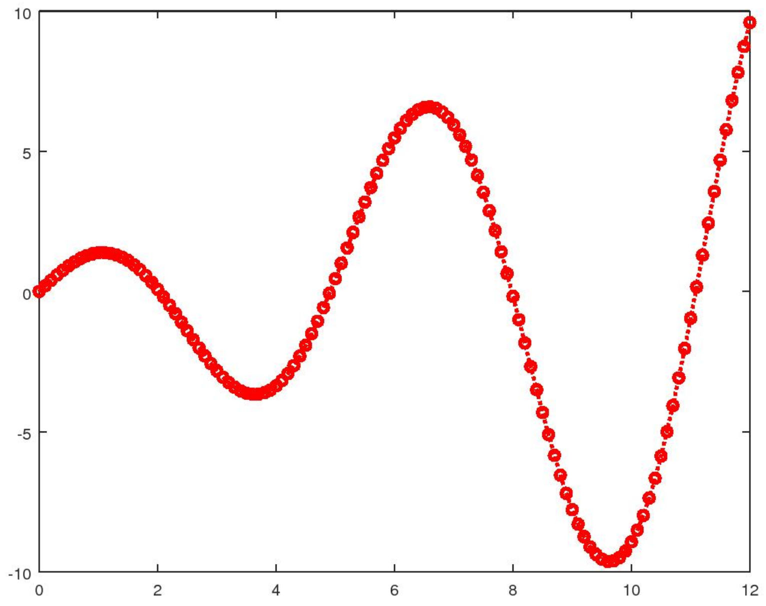
Try improving the plot

```
% to find out how to modify the plot
%help plot
```

(or you can google 'matlab chart line properties'.)

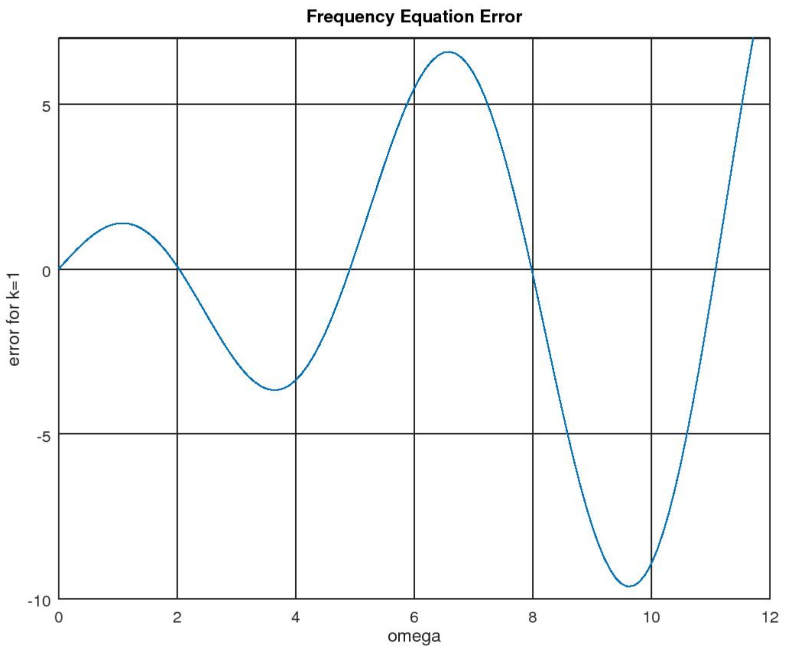
```
% colon: dashed line, o: circle symbols, r: red line
plot(omegaPlot,errorPlot,':or','LineWidth',2)
```

If you count the circles you should find that there are 121 exactly.



Try, try, try again

```
% redo from scratch
plot(omegaPlot,errorPlot)

% explicitly set the x- and y-axes limits
axis([0 12 -10 7])

% add a grid
grid on

% add labels on the x- and y-axes
xlabel('omega')
ylabel('error for k=1')

% add a title
title('Frequency Equation Error')

% put the x-axis at y=0
%set(gca)
set(gca,'xaxislocation','origin')
```



There is clearly a root somewhere near 2. And there are other roots too: near 5, 8, 11, ...

ACCURATE VALUES FOR THE FREQUENCIES

To keep it simple, let's keep k=1 for now and find the lowest frequency, the one near 2, first. This can be done using Matlab function `fzero`.

Finding the value of the lowest frequency

We want to find the lowest positive frequency `omega`, call it `omega1`, where function `FreqEqError1` is zero. We already concluded from the graph that the value of `omega1` will be close to 2.

Matlab can find zeros ("roots") of functions using the `fzero` library function.

Warning: The word "root" as used here has nothing to do with a square root. It simply means the position where a function is zero. For example if $f(x) = 0$ when $x = x_1$, then x_1 is a "root" of the equation $f(x) = 0$.

When using `fzero`:

1. As its first input argument you must specify the function you want to be zero; `FreqEqError1` in this case. The function name must be preceded by

an @ character. (The reason has to do with the fact that a function name is not a normal input argument. For now just remember that you need an @.)

2. As the second input argument of `fzero`, you must specify either a rough guess for the root you want or a range in which it is located.

Giving a range is safer and more efficient than giving an initial guess. If you give an initial guess, `fzero` will *try* to find an range that contains the nearest root by itself. However, this may fail; probably you understand more about the function than `fzero` does.

```
% Get a clue how to use fzero first
%help fzero

% our approximate guess for the first root
omegaGuess1=2
% let fzero search for the exact root from this guess
omega1=fzero(@FreqEqError1,omegaGuess1)
```

```
omegaGuess1 =  2
omega1 =  2.0288
```

This value happens to be OK. But it might just as well have failed. And it is inefficient.

The *best* way is to tell fzero to search in a small range that contains only the root we want, like from 2 to 2.1. Note from the values given earlier that the errors are of opposite sign at 2 and 2.1; so the error **must** be zero somewhere in between 2 and 2.1.

```
% the range in which to search for the root
omegaRange1=[2 2.1]
% check that the end point errors are of different sign
endPointErrors=FreqEqError1(omegaRange1)
```

```
omegaRange1 =
   2.0000   2.1000
endPointErrors =
   0.077004  -0.196967
```

That seems to be OK!

```
% let fzero search for the exact root in this range
omega1=fzero(@FreqEqError1,omegaRange1)
```

```
omega1 =  2.0288
```

The result is the same as before. But this method was absolutely safe!

How about the other frequencies?

How about the frequencies near omega = 5, 8, 11, ...? That is going to be messy. So let's look a bit better at the plot first.

```
% redo the plot from scratch (for publishing purposes)
plot(omegaPlot,errorPlot)
axis([0 12 -10 10])
grid on
xlabel('omega')
ylabel('error for k=1')
title('Frequency Equation Error')
set(gca,'xaxislocation','origin')

% set the tick marks at odd multiples of pi/2
set(gca,'xtick',[pi/2:pi:12])
% 2016b+ Matlab may instead use function xticks

% change the numbers on the tick marks
set(gca,'xticklabel',...
        {'\pi/2' '3\pi/2' '5\pi/2' '7\pi/2'})
% 2016b+ Matlab may instead use function xticklabels
```

Find a lot more frequencies now

It seems that all the correct frequencies are a bit bigger than the odd multiples of $\pi/2$. So we can probably use each odd multiple of $\pi/2$ as a starting point for finding the corresponding frequency.

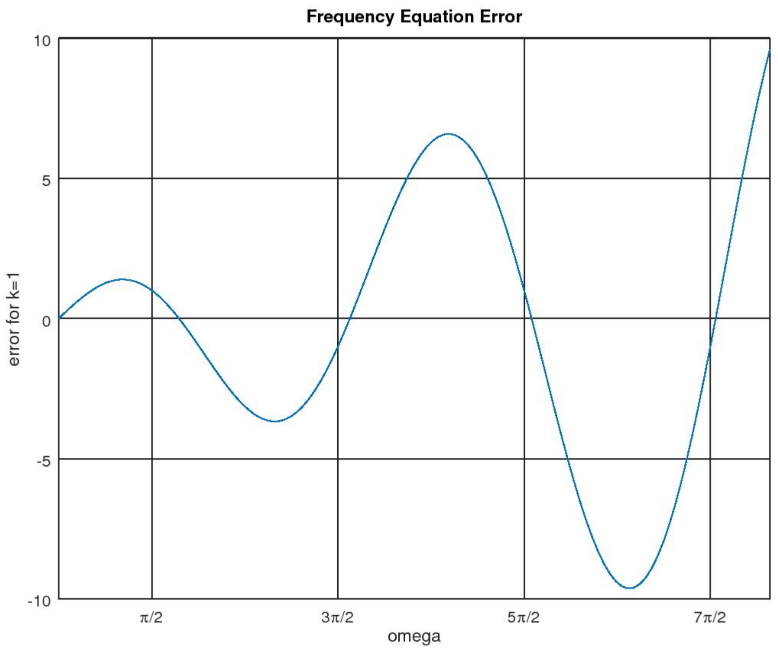
Or much better, we can use the odd multiple and the next odd multiple as a search range which will give the corresponding frequency for sure!

```
% let's try it out
omegaRange1=[pi/2 3*pi/2]
omega1=fzero(@FreqEqError1,omegaRange1)
```

```
omegaRange1 =
    1.5708    4.7124
omega1 =    2.0288
```

Yes, that produced the correct root again

```
% the second frequency, near 5
omegaRange2=[3*pi/2 5*pi/2]
omega2=fzero(@FreqEqError1,omegaRange2)
```



```
omegaRange2 =
   4.7124     7.8540
omega2 =   4.9132
```

```
% the third frequency, near 8
omegaRange3=[5*pi/2 7*pi/2]
omega3=fzero(@FreqEqError1,omegaRange3)
```

```
omegaRange3 =
     7.8540     10.9956
omega3 =   7.9787
```

```
% the fourth frequency, near 11
omegaRange4=[7*pi/2 9*pi/2]
omega4=fzero(@FreqEqError1,omegaRange4)
```

```
omegaRange4 =
   10.996     14.137
omega4 =   11.086
```

The difference from the start of the range, 7 pi/2, is less than 1% now!

To less than a percent error, we may approximate the remaining frequencies as $9\pi/2$, $11\pi/2$, $13\pi/2$, $15\pi/2$, ...

HOW ABOUT IF THE STIFFNESS IS NOT 1??

So far we assumed that k was 1 in

$$\sin(\omega) = -k\omega\cos(\omega)$$

What if it is not? Surely we cannot create a new function for every possible value of k??

Add the stiffness k as a function argument

To solve this problem, we can create a function that accepts k as a second input argument. Then we can use that function for *any* k we want. We will call this function `FreqEqError`. And *this time* we will put proper comments in the file (absolutely required for your homework solutions!):

```
function errorEq = FreqEqError(omega,k)

%
% Function used to find the natural frequencies of a
% string that has one end rigidly attached to the musical
% instrument but the other end attached to a flexible
% strip.
%
%             errorEq = FreqEqError(omega,k)
%
% Input:
%    omega: The frequency to test, in radians.
%    k:     The bending stiffness of the strip.
%    These parameters are suitably nondimensionalized in
%    a way not important here.
%
% Output:
%    errorEq: Zero if omega is a correct frequency (tone)
%             of the string, nonzero if it is not.
%
% Advanced analysis taught in Analysis in Mechanical
% Engineering II shows that the equation the frequencies
% must satisfy is:
%            sin(omega) = - k omega cos(omega)
% So if the frequency is not right, the error in the
% equation (difference between the left and right hand
% sides) is:
%       errorEq = sin(omega) + k omega cos(omega)
```

```
%

% set the return variable equal to the error
errorEq = sin(omega) + k*omega.*cos(omega);
% omega is in radians and the .* and ; are really needed

end
```

Using an anonymous function

Function `fzero` can only find zeros of functions with a *single* input argument. `FreqEqError` has two, `omega` and `k`, so there is no way to use function `FreqEqError` directly in `fzero`.

Instead we must tell Matlab to create a *new* function with a single input argument `omega`. This function should have the desired value of `k` already hidden inside. Then we can give that function to `fzero`.

The convenient way to do that is to tell Matlab to create an anonymous (nameless) function `(omega)` of `omega`. (It is called anonymous because there is no name like `f` or `sin` or `log` or whatever in front of the `(omega)`.) This anonymous function should for a given `omega` return `FreqEqError(omega,k)`, if we have already set the desired value of `k` earlier.

To define the anonymous function, follow the function specification `(omega)` by its definition in terms of `FreqEqError`:

```
    (omega) FreqEqError(omega,k)
```

This can be specified as first input argument of `fzero` if preceded by an @ character.

```
% let's first try it for our old value k = 1
k=1
omegaRange1=[0.5*pi 1.5*pi]
omega1=fzero(@(omega) FreqEqError(omega,k),omegaRange1)
omegaRange2=[1.5*pi 2.5*pi]
omega2=fzero(@(omega) FreqEqError(omega,k),omegaRange2)
```

```
k =   1
omegaRange1 =
    1.5708    4.7124
omega1 =   2.0288
omegaRange2 =
    4.7124    7.8540
omega2 =   4.9132
```

Seems to work OK.

How about another value of k now?

```
% try k = 2 now
k=2

% compute the new frequencies as before for this k
omegaRange1=[0.5*pi 1.5*pi]
omega1=fzero(@(omega) FreqEqError(omega,k),omegaRange1)
omegaRange2=[1.5*pi 2.5*pi]
omega2=fzero(@(omega) FreqEqError(omega,k),omegaRange2)
omegaRange3=[2.5*pi 3.5*pi]
omega3=fzero(@(omega) FreqEqError(omega,k),omegaRange3)
omegaRange4=[3.5*pi 4.5*pi]
omega4=fzero(@(omega) FreqEqError(omega,k),omegaRange4)
```

```
k =   2
omegaRange1 =
   1.5708    4.7124
omega1 =   1.8366
omegaRange2 =
   4.7124    7.8540
omega2 =   4.8158
omegaRange3 =
    7.8540    10.9956
omega3 =   7.9171
omegaRange4 =
   10.996    14.137
omega4 =   11.041
```

Seems to work OK.

Read: More on @: function handles and quick functions

Why do we need to put an @ in front of a function that we provide as an input argument to another function (in particular to `fzero` in our case)?

The basic reason is that a function is not a normal input argument. A normal input argument of a function may be explicit data, (like the 4 in `sqrt(4)`, say) or the name of a variable whose data is passed on to the function (like the `x` in `sqrt(x)`, say) or some combination of the two. In any case, what is passed to the function is some sort of *data*. The name of a function is not data for Matlab, it is something special.

However, if you precede a function name or anonymous function definition by an @ character, it *creates* data; data on the function. This data is called a "function handle". The data is not, of course, a simple number (which Matlab would call a "double"). Instead if the function is in a function file on disk, the function handle consists of, among others, the name of the function and the location on disk where it is stored. If the function is an anonymous one, the function handle consists of the definition of the anonymous function, as well as the values of any constants the definition used, like `k` in our case.

In either case, the function handle data is passed to `fzero`. So all is well again, as far as Matlab is concerned: `fzero` receives *data* of some kind, a function handle, in its first input argument. It does *not* receive a function name. (Since an anonynmous function *has* no name, the latter would not be possible anyway.)

But how does having data on the function help `fzero`? Well, there is one additional point. If a *variable* contains a function handle, Matlab allows you to use the name of that variable to *evaluate* the function. It is as if the name of the variable was a name for the function. So whoever wrote function `fzero` could use the name of the first input argument of `fzero` as if it was the name of the function that should be zero.

Why am I telling *you* all this? Well, this can also be helpful for other purposes too, in particular for quickly defining a "function". If you store the handle of an anonymous function in a variable, then the name of that variable acts much like a name of the anonymous function.

For example, the next would work fine to get the first frequency:

```
% set a value for k
k=1
% put the anonymous function handle in a variable
funHandle=@(omega) FreqEqError(omega,k)
% print out a value of FreqEqError for testing
FreqEqError(2,k)
% check that funHandle(2) gives the *same* value
funhandle(2)
% OK, so now let fzero find the first root
omega1=fzero(funHandle,[0.5*pi 1.5*pi])
```

(Note that there is no @ before funHandle; funHandle is not *really* the name of a function, it is just a variable that contains a function handle. But do not worry too much about that; if you do put an @, Matlab will complain loudly, and you know.)

So why did I not tell you to do things the above way in the first place? Well, one big reason is that the way I told you to do it, the first argument always starts with an @. That is easy to remember. The other big reason is that after the above, the following would *not* work correctly.

```
k=2
omega1=fzero(funHandle,[0.5*pi 1.5*pi])
```

After you change `k` you *must* recreate `funHandle`; this is *not* automatic. (Although you can only see it clearly in Octave, the function handle saves the value of `k`, not the name of the variable.) So I think it is simpler and safer to just avoid saving function handles in variables. (But admittedly, it may be somewhat less efficient, as the function handle must be recreated every time. Not a big deal for us.)

PRINT OUT THE FREQUENCIES NICELY

The `fprintf` function allows you to print out numbers in your own way. Use it as

```
        fprinf('FORMATSTRING',VARIABLES)
```

The `FORMATSTRING` contains the literal text you want to print, as well as a "formatting operator" for each numerical variable whose value is to be printed. The formatting operators are:

- %i: integer (%d also works)
- %f: floating point number
- %e: floating point number in exponential notation
- %E: floating point number in Exponential notation
- %g: either %f or %e, depending on the number

WARNING: *Normally you need to end FORMATSTRING with* `\n`*. Otherwise the line does not end.*

In the first following `fprintf` statement,

- the first `%f` gets replaced by the value of `k`
- the `%i` gets replaced by the frequency number 1
- the second `%f` gets replaced by the value of `omega1`

and similar for the other three `fprintf` statements.

```
% print out the found frequencies formatted
fprintf('for k =%f, omega%i equals: %f\n',k,1,omega1)
fprintf('for k =%f, omega%i equals: %f\n',k,2,omega2)
fprintf('for k =%f, omega%i equals: %f\n',k,3,omega3)
fprintf('for k =%f, omega%i equals: %f\n',k,4,omega4)
```

```
for k =2.000000, omega1 equals: 1.836597
for k =2.000000, omega2 equals: 4.815842
for k =2.000000, omega3 equals: 7.917053
for k =2.000000, omega4 equals: 11.040830
```

It is *illegal* in this class to put actual data numbers in `FORMATSTRING`! (Well, I guess it would be defensible to put the 1, 2, 3, and 4 explicitly in FORMATSTRING.)

The above is not yet good enough. You *must take control of the formatting!* To specify the precise way a number should be printed, use:

- %PRINTPOSITIONSi for integers
- %PRINTPOSITIONS.DIGITSBEHINDPOINTf for floating point numbers
- ...

In the current case, that works out to:

- Use %.0f to print k with zero digits behind the point
- Use %1i to print the root number in a single print position
- Use %5.2f to print the frequency with 2 digits behind the decimal point and 5 print positions total, so that, say, 1.836... prints as [SPACE]1.84 and 11.040... as 11.04. This makes the decimal points align.

```
% print out the first frequency
fprintf('for k = %.0f, omega%1i equals: %5.2f\n',...
         k,1,omega1)
```

```
for k = 2, omega1 equals:  1.84
```

Note that %f did perform rounding of 1.836....

Also, the 1 in %1i does nothing; if Matlab sees that the number is 10 or more, it will use two print positions anyway. So you may as well leave that one away:

```
% print out the remaining frequencies
fprintf('for k = %.0f, omega%i equals: %5.2f\n',...
         k,2,omega2)
fprintf('for k = %.0f, omega%i equals: %5.2f\n',...
         k,3,omega3)
fprintf('for k = %.0f, omega%i equals: %5.2f\n',...
         k,4,omega4)
```

```
for k = 2, omega2 equals:  4.82
for k = 2, omega3 equals:  7.92
for k = 2, omega4 equals: 11.04
```

Note that the decimal points now line up.

ADDITIONAL REMARKS

To find the location of the smallest or largest value of a function instead of a zero value, you could find a zero for the derivative. Alternatively, you can directly search for a minimum by using Matlab function `fminbnd` instead of `fzero`. To search for a maximum, search for a minimum of minus the function.

If you have more than one equation for more than one variable, things get messier. Try `fsolve` or `fminunc`.