Chapter 05 - Solving Equations Symbolically

Table of Contents

We can ask MATLAB to try to solve equations two different ways. MATLAB can sometimes obtain a symbolic solution by manipulating the symbols in the equation(s) much like you would do with pencil and paper in an introductory math class. Another way is to find a numerical answer which may be approximate. Generally the second approach is good when the equation is very hard or impossible to solve symbolically.

Solving One Equation Symbolically

Suppose you want to find the solutions to the equation

```
4*x-5=7
```

Matlab can solve this with the `solve` command. First we symbolically define our variable `x` and then apply the command. Try this. Note the use of the `==`. Matlab is currently undergoing a change to bring it in line with most computer programming languages. Now you see when **checking** for equality use `==`. There are places in Matlab where you can still geta way with one equals sign for checking equality (historical reasons) but I recommend against it.

```
syms x
solve(4*x-5==7)
```

```
ans =

3
```

That was easy! MATLAB found the solution.

Important!

Matlab hasn't assigned the variable `x` here, it's just told you what the symbolic variable `x` would be to solve the equation.

Here's another; notice that we don't need to `syms x` because Matlab already knows that `x` is symbolic.

```
solve(x^2+4*x-21)
```

```
ans =

 -7
  3
```

You can also ask MATLAB to solve equations that involve arbitrary constants. A nice example is to try to find the general solutions to the generalized form of the quadratic equation `a*x^2+b*x+c=0`. What do you think the solutions should look like? Let's see if you're right...

```
syms x a b c
solve(a * x ^ 2 + b * x + c == 0)
```

```
ans =

 -(b + (b^2 - 4*a*c)^(1/2))/(2*a)
 -(b - (b^2 - 4*a*c)^(1/2))/(2*a)
```

Do those solutions look familiar? You may have to mentally re-arrange them to make them look the way you're used to seeing them!

Which Variable to Solve For?

How does Matlab know which variable to solve for? It tries to follow common-sense respecting what we usually solve for. Since the variable `x` is so common it solves for that one unless we tell it otherwise:

```
syms a x
solve(a*x-x^2==2)
```

```
ans =

 a/2 - (a^2 - 8)^(1/2)/2
 a/2 + (a^2 - 8)^(1/2)/2
```

If you prefer you can explicitly tell it to solve for `x`:

```
solve(a*x-x^2==2,x)
```

```
ans =

 a/2 - (a^2 - 8)^(1/2)/2
 a/2 + (a^2 - 8)^(1/2)/2
```

If we wished to solve for `a`; we'd have to tell Matlab:

```
solve(a*x-x^2==2,a)
```

```
ans =
```

```
(x^2 + 2)/x
```

Solving a System of Equations Symbolically

You can use the `solve` command for a whole system of equations as well. For example, suppose we are trying to find the solution to the following system of equations:

```
3*x+4*y+z-7=1
x-y-15=2
6*x-2*y-5*z+11=3
```

We can try to use the `solve` command to do this by feeding it all of the equations at once, separating them with commas:

```
syms x y z
solve(3*x+4*y+z-7==1,x-y-15==2,6*x-2*y-5*z+11==3)
```

```
ans =

  struct with fields:

    x: [1×1 sym]
    y: [1×1 sym]
    z: [1×1 sym]
```

Hmmmm. That output is a bit disappointing! To enable us to see the actual answers conveniently, we can introduce an intermediate variable that will store the whole "vector" of answers. Then we can view the values individually. Here's how that might look:

```
syms x y z
MyAnswers = solve (3*x+4*y+z-7==1,x-y-15==2,6*x-2*y-5*z+11==3)
```

```
MyAnswers =

  struct with fields:

    x: [1×1 sym]
    y: [1×1 sym]
    z: [1×1 sym]
```

Now we can ask to see each of the three values individually, as follows:

```
MyAnswers.x
```

```
ans =

26/3
```

```
MyAnswers.y
```

```
ans =

-25/3
```

```
MyAnswers.z
```

```
ans =

46/3
```

Another (perhaps better) way to accomplish the same thing is by assigning the value of the computation to a vector comprised of variables like this:

```
[x y z] = solve (3*x+4*y+z-7==1,x-y-15==2,6*x-2*y-5*z+11==3)
```

```
x =

26/3


y =

-25/3


z =

46/3
```

Notice that the answers are displayed as fractions rather decimal expansions. Does that surprise you? It actually makes sense because Matlab solved the equations symbolically. If you solved these equations by hand, the last operation you would do in calculating `z` would be to take `185` and divide it by `13`. Although you could try to evaluate this as a decimal why should you? The expression `185/13` is exact. If this were written as a decimal it would be `14.230769230769230769230769...`. We could only write down an approximation this way and besides -- it looks really hideous!

Order of The Variables in the Solution

The `solve` command returns a vector with entries in alphabetical order. For example

```
solve(x+y==0,x-y+2==0)
```

will return the vector whose first entry is `x` and whose second entry is `y`. If you do something like

```
[y x]=solve(x+y==0,x-y+2==0)
```

then what happens is that `y` gets assigned to the `x` solution and `x` gets assigned to the `y` solution. This is not what you want.

To unconfuse things make sure that your entries in the vector on the left are in alphabetical order! For example if your variables are `dogs` and `cats` then *don't* do:

```
[dogs cats] = ...
```

instead do

```
[cats dogs] = ...
```

Failure

This symbolic approach works well for some problems. Unfortunately there is a large class of problems that will defy attempts to solve them this way. For example:

```
solve(log(x)+x+1==0)
ans =

wrightOmega(-1)
```

Matlab has used some really fancy confusing math to try to solve this and has given back a very confusing answer. Instead of taking this approach, this equation can be solved using a numerical method which will yield a very good approximation to the precise answer. We'll see this soon.

Published with MATLAB® R2016b