7 SYMBOLIC MATH

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 lectureN.txt

% load the symbolic package (NEEDED FOR OCTAVE)
pkg load symbolic
syms initpython
sympref display flat
```

LESSON SUMMARY

Normally Matlab deals with numbers, where a variable like `x` stands at any given time for one particular stored number. Instead, symbolic mathematics deals with abstract symbols, in which a symbolic variable like `x` could be any number.

To declare variables to be symbolic ones, use

```
syms VARIABLE VARIABLE ...
```

Given a ratio like 1/3 (an integer divided by an integer), normally Matlab would numerically evaluate it and store it as 0.3333... This is not exact. To make the expression 1/3 symbolic, use

```
sym('1/3').
```

This prevents it from being approximated. The symbolic math functions can then, for example, simplify `9*sym('1/3')` as exactly 3.

Symbolic math functions include:

- `diff(SYMFUN,SYMVAR,...)`, to differentiate a symbolic function with respect to one or more of its symbolic variables.
- `int(SYMFUN,SYMVAR,[START END])`, to try to integrate a symbolic function with respect to one of its symbolic variables from a starting value to an end value.
- `int(SYMFUN,SYMVAR)`, to try to find an antiderivative of a symbolic function with respect to one of its symbolic variables.
- `solve(SYMEQ,SYMVAR)`, to try to solve a symbolic equation for one of its symbolic variables. Specify the symbolic equation as `LHS == RHS` where `LHS` and `RHS` are symbolic expressions; note the doubled equals sign. You can also specify more than one equation in more than one unknown.
- `taylor(SYMFUN,VAR,VAR0,'Order',DEGREE)`, to write the Taylor series of a symbolic function. `VAR0` is the base point; if `",VAR0"` is left away, the value of `VAR0` is zero (so a McLaurin series). The value of `DEGREE` is the degree of the term *no longer shown.* If `",'Order',DEGREE"` is left away, the value of `DEGREE` is 6.
- `subs(SYMEXPR,SYMVAR,SYMVAL)`, to substitute `SYMVAL` for `SYMVAR` in symbolic expression `SYMEXPR` and clean up.
- `subs(SYMEXPR,{SYMVAR1 SYMVAR2 ...},{SYMVAL1 SYMVAL2 ...})`, to perform multiple substitutions all at once.

- `factor(SYMEXPR)`, to factor a symbolic expression.
- `factor(SYMINT)`, to factor a symbolic integer into its prime factors.
- `partfrac(SYMRAT,SYMVAR)`, to take apart a symbolic ratio into its partial fractions with respect to symbolic variable SYMVAR.
- `expand(SYMEXPR)`, to fully write out a symbolic expression.
- `prod(SYMEXPRVEC)`, to multiply out a vector of symbolic expressions. (Note that `prod` is not actually a symbolic function, but it can be used on the vectors of symbolic expressions produced by `factor` in Matlab.)
- `children(SYMEXPRVEC)`, to take apart a symbolic expression into a vector of subexpressions. Depending on conditions, the subexpressions can be left and right hand sides, terms, factors, etcetera. So you may need to apply `children` recursively to get the pieces you want.
- `simplify(SYMEXPR)`, to try to simplify a symbolic expression.
- `collect(SYMEXPR,SYMVAR)`, to collect the coefficients of the powers of a symbolic variable in a symbolic expression.
- `pretty(SYMEXPR)`, to show a symbolic expression in, maybe, a more readable form.
- `double(SYMVAL)`, to convert an exact symbolic value into a simple Matlab number with about 16 significant digits.
- `vpa(SYMVAL,DIGITS)`, to evaluate a symbolic value to at least `DIGITS` significant digits. If `",DIGITS"` is left away the value of `DIGITS` is 32.
- `HANDLE=matlabFunction(SYMFUN)`, to convert an exact symbolic function into a normal Matlab anonymous function (which approximates numbers to about 16 significant digits). `HANDLE` is the name for a handle to the anonymous function. It can be used much like a name for the anonymous function too.

To get help on a symbolic function FUNCTION, in Matlab you may need to specify it as

```
help sym/FUNCTION
```

to prevent getting help on a non-symbolic function FUNCTION. In Octave, that is

```
help @sym/FUNCTION
```

Key areas of the online book

Before the lecture, in the online book do:

- 21.1 Symbolic variables: skip the stuff after PA 21.1.2.
- 21.3 Formula manipulation and simplification: all.
- 21.6 Series: all.
- 21.8 Algebraic equations: all.
- 21.9 Calculus: all.

INTRODUCTION

Normally Matlab generates *numbers*, not *formulae*. If you ask Matlab to find a root where a function is zero, maybe using `fzero`, it gives you a number. If you ask Matlab to integrate a function from a lower limit to an upper limit using `integral` or `quad`, it gives you a number. Etcetera.

But sometimes you want a formula instead of a number. For example, you might want the derivative or antiderivative of a function. Either one is a formula, not a number. Also, sometimes you want to see an *exact* number, and almost all numbers in Matlab have round-off errors.

What you need for such purposes is a *symbolic mathematics* program. Such a program is available inside Matlab as the "Symbolic Math Toolbox". Normally MathWorks charges separately for this package. However, it is included in the "Student Edition", as well as on the COE computers. Octave uses the separate, free, "SymPy" package.

In this section we will illustrate how normal Matlab computations and symbolic ones differ. We will look at a simple example function, a quadratic in fact:

$$q = -4x^2 + 3x + 12$$

Playing with a quadratic numerically

Let's first review some of the normal stuff we can do with the example quadratic q. To do so, we will first define the quadratic as a normal numerical function. In particular, we will define `qNum` as a handle to an anonymous numerical version of the quadratic:

```
% the example quadratic as a normal Matlab function
qNum = @(x) -4*x.^2+3*x+12
```

```
qNum =
@(x) -4 * x .^ 2 + 3 * x + 12
```

We can numerically integrate this function between, say, 0 and 2:

```
% integrate qNum from 0 to 2 using "quad"
qNumInt02=quad(qNum,0,2)
```

```
qNumInt02 =   19.333
```

Note that we got a floating point number for the integral. This number is pretty accurate, but it is *not* exact. Sometimes you want the exact value.

Note also that there is no way to directly find the antiderivative of `qNum` as a simple cubic.

We can certainly find one of the two roots using `fzero`, by giving `fzero` an appropriate starting value:

```
% find a root near x = 1
qNumRoot=fzero(qNum,1)
```

```
qNumRoot = -1.3972
```

Again we got an approximate floating point number, rather than an exact answer. And without using our knowledge of quadratics, (or careful plotting), this does not tell us that there is another root somewhere.

Finally, there is no way to directly find the derivative of `qNum` as a linear function. And you may recall from the lesson on interpolation that even finding a bunch of *values* of the derivative of a numerically given function can be very tricky.

Playing with a quadratic symbolically

So let's now do the above things using symbolic variables!

The first thing to remember is: Be sure to inform Matlab with the `syms` or `sym` command when variables and/or numbers are intended to be symbolic. Normal variables are names of storage locations with a number in it. But a symbolic variable does not store a number; at all times it can stand for *any* number. So a normal variable `x` is very different from a symbolic variable `x`, and Matlab must know which of the two `x` is.

```
% tell Matlab that, from now on, x is a symbolic variable
syms x
```

If `x` is a symbolic variable, then expressions involving `x` are symbolic too. So we can define the symbolic quadratic q as follows:

```
% the example quadratic as a symbolic function
qSym=-4*x^2+3*x+12
```

```
qSym = (sym) -4*x**2 + 3*x + 12
```

We can integrate this function symbolically with respect to `x` between, say, 0 and 2 using the symbolic `int` function.

```
% integrate qSym with respect to x from 0 to 2
qSymInt02=int(qSym,x,[0  2])
```

```
qSymInt02 = (sym) 58/3
```

Note that the value 19.333... that we got earlier using `quad` on `qNum` is equal to 58/3 to about 16 significant digits. But you can easily check using calculus that the 58/3 value is *exact*.

Using symbolic math we can also find an antiderivative; just use `int` but do not specify any limits of integration now:

```
% find the symbolical antiderivative
qSymInt=int(qSym,x)
```

```
qSymInt = (sym) -4*x**3/3 + 3*x**2/2 + 12*x
```

In Matlab you may want to clean up the result using the `expand` function; in Octave it is already expanded:

```
% expand the result as needed and desired
expand(qSymInt)
```

```
ans = (sym) -4*x**3/3 + 3*x**2/2 + 12*x
```

You can readily check by differentiation that the above is the correct antiderivative of $-4x^2 + 3x + 12$ that is zero when x is zero.

Note that if you leave the `x` away from `int(qSym,x)`, it will still work, and we might not even subtract much credit, as there is no other variable that Matlab can integrate the expression with respect to. If that was ax^2 instead of $-4x^2$, well there goes your credit, as that can be integrated with respect to a instead of x.

We can find the derivative using the `diff` function:

```
% find the symbolical derivative
qSymDiff=diff(qSym,x)
```

```
qSymDiff = (sym) -8*x + 3
```

You can readily check that this is the correct derivative of $-4x^2+3x+12$. (Yes, the `x` must be specified for full credit.)

Both roots of the quadratic can be found exactly using the `solve` function (note the doubled equals sign):

```
% Find the roots exactly
qSymRoots=solve(qSym == 0,x)
```

```
qSymRoots = (sym) Matrix([[3/8 + sqrt(201)/8], [-sqrt
    (201)/8 + 3/8]])  (2x1 matrix)
```

Note that we got *both exact* roots as a symbolic vector.

You might try using `pretty` for getting a formatted look at the roots:

```
% format the roots
pretty(qSymRoots)
```

```
  [         _____ ]
  [ 3     \/ 201  ]
  [ - + ---------- ]
  [ 8         8    ]
  [                ]
  [     _____      ]
  [   \/ 201     3 ]
  [- ---------- + -]
  [       8      8 ]
```

How about the root -1.3972 that we got earlier using `fzero`? Is it one of the two roots above to 16 digits accuracy, and if so, which one? To answer that, you can use the `double` function to simplify the symbolic roots to normal Matlab numbers with 16 significant digits.

```
% the root found using fzero on qNum
qNumRoot

% the symbolic roots converted to normal Matlab numbers
qSymRootsNum=double(qSymRoots)

% compare qNumRoot to the second symbolic root
qNumRootError=qNumRoot-qSymRootsNum(2)
```

```
qNumRoot = -1.3972
qSymRootsNum =
```

```
   2.1472
  -1.3972
qNumRootError = 0
```

Apparently, `fzero` did find the numerical root very accurately.

Next recall from your math classes that if you know the roots of a quadratic, you can factor it. In particular, if we call the two roots of q as obtained above x_1 and x_2, then $q = -4(x - x_1)(x - x_2)$.

However, we should also be able to factor the quadratic *directly* using the `factor` function.

```
% try to factor the quadratic
qSymFactors=factor(qSym)
```

```
qSymFactors = (sym) -4*x**2 + 3*x + 12
```

Oops, that did not work! Try looking for help on function `factor`. Now you need to be careful: if you just use `"help factor"`, you will get help on the *wrong* function `factor` (one that only factors integers). To get help on the desired *symbolic* function `factor`, you must precede it with `"sym/"` in Matlab, or `@sym/` in Octave:

```
% get help on symbolic function factor (disabled here)
%help sym/factor % Matlab version
%help @sym/factor % Octave version
```

Anyway, if you read the help given in Matlab, you will see that the problem is that the roots of our quadratic are irrational. To get Matlab to factor the quadratic, you need to add `",'FactorMode','full'"` to the arguments:

```
% factor a quadratic with irrational roots
qSymFactors=factor(qSym,'FactorMode','full')
```

```
qSymFactors = (sym) -4*x**2 + 3*x + 12
```

This does not (yet) work in Octave, at least not in my relatively old version of SymPy.

Matlab gives the factors as a vector. To see the normal factored form use the `prod` function.

```
% in Matlab, form the normal factored form using prod
%qSymFactors=prod(qSymFactors)
%pretty(qSymFactors)
```

If the quadratic has rational roots, like

$$q_{\text{rat}} = -4x^2 + 3x + 27$$

(note 27 instead of 12), there is no problem:

```
% a quadratic with rational roots
qRatSym=-4*x^2+3*x+27

% factor it
qRatSymFactors=factor(qRatSym)
% in Matlab, form the normal factored form using prod
%prod(qRatSymFactors)
```

```
qRatSym = (sym) -4*x**2 + 3*x + 27
qRatSymFactors = (sym) -(x - 3)*(4*x + 9)
```

SIMPLIFYING ANSWERS

In classes like Analysis in Mechanical Engineering, you are required to simplify your answers. Symbolic math to the rescue!

Watch it, however. If you try to simplify, say, a numeric ratio like 629/969 as

```
simplify(629/969)
```

then Matlab sees "629/969", evaluates that as 0.6491... to 16 significant digits and gives that to the symbolic `simplify` function. Of course `simplify` cannot make any sense out of 0.6491.... In fact, it will refuse to cooperate.

What you need to do is tell Matlab that the entire "629/969" is to be treated as a symbolic expression, to be given to `simplify` "as is". You can do that with the `sym` function:

```
% simplify 629/969 (divides out the common factor 17)
ratioSimplified=simplify(sym('629/969'))
```

```
ratioSimplified = (sym) 37/57
```

VERY HIGH ACCURACY

Function `vpa`, ("variable precision arithmetic"), will give you numbers to arbitrarily high accuracy. As an example, let's try to get the result $\pi^2/6$ of the infinite sum that we talked about in the previous lesson to 50 significant digits.

Watch it again. There is a difference between what Matlab makes of `pi^2/6` and `sym('pi')^2/6`:

```
% pi^2/6
sumNum=pi^2/6

% sym('pi')^2/6
sumSym=sym('pi')^2/6
```

```
sumNum =  1.6449
sumSym = (sym)  pi**2/6
```

So there is a wrong way and a right way to get the sum to 50 significant digits:

```
% Matlab gives vpa a number equal to pi^2/6 to 16 digits:
wrongSum=vpa(pi^2/6,50)

% Matlab gives vpa the symbolic string pi^2/6:
rightSum=vpa(sym('pi')^2/6,50)
```

```
wrongSum = (sym)
   1.6449340668482264060656916626612655818462371826172
rightSum = (sym)
   1.6449340668482264364724151666460251892189499012068
```

In the first case, Matlab evaluated $\pi^2/6$ in its normal way, losing all significant digits beyond the 16th, and gave that number to function `vpa`. Of course there was no way for `vpa` to correctly reconstruct the lost 17th to 50th significant digits. So all shown digits beyond the 16th (or 17th, to be picky) are wrong. (I assume they are correct for the inaccurate number `vpa` received from Matlab.)

In the second case, Matlab gave `vpa` the symbolic string `'pi^2/6'` and `vpa` had no problem evaluating that correctly to 50 digits accuracy.

SOLVING EQUATIONS

The Symbolic Toolbox can solve quite a lot of equations exactly.

If it cannot, it will drop back to a numerical solution.

Solving quadratic equations

Suppose you no longer remembered the solution to the quadratic equation

$$ax^2 + bx + c = 0$$

The symbolic toolbox can give it to you. First define the quadratic symbolically:

```
% tell Matlab that the variables are symbolic
syms a b c x
```

```
% define the generic quadratic polynomial
qGen=a*x^2+b*x+c
```

```
qGen = (sym) a*x**2 + b*x + c
```

Next use `solve` to find the two roots where the quadratic is zero:

```
% find the symbolic solution using solve
qGenRoots=solve(qGen == 0,x)
```

```
qGenRoots = (sym) Matrix([[(-b + sqrt(-4*a*c + b**2))
    /(2*a)], [-(b + sqrt(-4*a*c + b**2))/(2*a)]])  (2x1
    matrix)
```

If you leave `== 0` away in the `solve` command, it will still work, but your credit will plunge for writing confusing code. A bare `qGen` is a function, not an equation.

For another look at the roots, let's reformat them using pretty:

```
% show the roots with pretty
pretty(qGenRoots)
```

```
  [          _______________  ]
  [         /             2   ]
  [ -b + \/   -4*a*c + b     ]
  [ ------------------------ ]
  [            2*a           ]
  [                          ]
  [ /        _______________\ ]
  [ |       /             2 | ]
  [-\b + \/   -4*a*c + b   / ]
  [--------------------------]
  [            2*a           ]
```

Now this should agree with the roots we got in the first section for the special case that a was -4, b was 3, and c was 12. To see whether this is really true, we can use the `subs` function to substitute these values for a, b, and c:

```
% see whether we get back our previous solution
qSymRootsTest=subs(qGenRoots,{a b c},{-4 3 12})
qSymRoots
```

```
qSymRootsTest = (sym) Matrix([[-sqrt(201)/8 + 3/8], [3/8
    + sqrt(201)/8]])  (2x1 matrix)
qSymRoots = (sym) Matrix([[3/8 + sqrt(201)/8], [-sqrt
    (201)/8 + 3/8]])  (2x1 matrix)
```

You can see they are the same roots, just reordered.

If you want to evaluate roots of a quadratic a lot, using symbolic logic would be a slow process for Matlab. In that case, you are better off writing a normal Matlab numerical function for them; one that uses normal numbers with 16 significant digits instead of symbolic logic.

But you do not have to write that function yourself from scratch. Matlab can convert the symbolic solution to a normal function for you. Just use function `matlabFunction`:

```
% create a normal Matlab function for the roots
qGenRootsFun=matlabFunction(qGenRoots)
```

```
qGenRootsFun =
@(a, b, c) [(-b + sqrt(-4 .* a .* c + b .^ 2)) ./ (2 .*
    a); -(b + sqrt(-4 .* a .* c + b .^ 2)) ./ (2 .* a)]
```

Looks OK. We can test it on the values for a, b and c from the first section and see whether we get the same roots:

```
% test it for our earlier example
qGenRootsFunTest=qGenRootsFun(-4,3,12)
```

```
qGenRootsFunTest =
  -1.3972
   2.1472
```

That are indeed the two roots that we got in the first section to about 16 significant digits.

Solving other equations

Let's look at some other equations that the Symbolic Toolbox manages to solve.

One famous result of complex variable theory is that $e^{i\pi}$ equals -1, where i equals $\sqrt{-1}$. Let's see whether the Symbolic Toolbox knows that:

```
% make sure x is still a symbol
syms x

% solve the equation e^x = -1 for x
solve(exp(x) == -1,x)
```

```
ans = (sym) I*pi
```

So the answer is yes. The Toolbox knows it. (However, the correct solution to the equation above is $i\pi$ *plus any integer multiple of* $2i\pi$.)

To make things more interesting, let's have a second variable y in addition to x.

A simple quadratic equation in those two variables is

$$axy - bx - 1 = 0$$

where a and b are constants.

```
% make sure a, b, and y are symbols too
syms a b y

% create the symbolic quadratic function
qxySym=a*x*y - b*x - 1
```

```
qxySym = (sym) a*x*y - b*x - 1
```

If we consider y a given quantity, we can solve the equation for x:

```
% solve for x
xSol=solve(a*x*y - b*x - 1 == 0, x)
```

```
xSol = (sym) 1/(a*y - b)
```

You can easily check this solution.

If instead we consider x a given quantity, we can solve the equation for y:

```
% solve for y instead
ySol=solve(a*x*y - b*x - 1 == 0, y)
```

```
ySol = (sym) (b*x + 1)/(a*x)
```

You can also easily check that solution.

Sometimes rewriting the equation can help you understand it better. For example, the `collect` function can collect the coefficients of the powers of an unknown. Unfortunately, Octave does not (yet) have this function.

```
% collect the coefficients of powers of x
%qxySymMod=collect(a*x*y-b*x-1,x)
```

Note that normally a single equation can only be solved for a single unknown. Our quadratic equation can be solved either for x or for y but not for both. To solve for two unknowns, you need two equations.

However, `solve` can solve some systems of equations as well. For example, it can solve linear systems of equations:

```
% solve two linear equations in two unknowns
[xSol,ySol] = solve(x+y == 7, x-y == 1,x,y)
```

```
xSol = (sym) 4
ySol = (sym) 3
```

You can check this solution by simply plugging it in.

When the equations are nonlinear though (like those involving nonlinear functions), analytical solution tends to get much more difficult.

If `solve` fails, Matlab will fall back to trying to solve the equations numerically. Of course, that is a tricky business; you better check the results. Octave simply produces an error if analytic solution fails.

```
% try to solve two non-linear equations (fails in Octave)
%[xSol,ySol] = solve(x^2+cos(y) == 7, cosh(x)-y == 1,x,y)
```

PARTIAL FRACTIONS

In analyzing the dynamics of controlled systems, you often encounter ratios of big polynomials. Then you want to take these ratios apart in simpler fractions. The reason is that these simpler fractions tell you many important things. For example, they tell you whether, if the system is disturbed, it will return to its normal position, versus, say, crash. And, if it does return to its normal position, they will also tell you how fast the system will return to normal.

In Calculus, partial fractions are also used to integrate ratios of big polynomials.

Symbolic function `partfrac` can give you the partial fractions of a ratio of polynomials. First, let's create an example ratio:

```
% make sure x is still symbolic
syms x

% the example symbolic ratio
ratSym=(2*x^2-3*x+1)/(x^3+2*x^2-9*x-18)
```

```
ratSym = (sym) (2*x**2 - 3*x + 1)/(x**3 + 2*x**2 - 9*x -
    18)
```

```
% show it with pretty
pretty(ratSym)
```

```
        2
     2*x  - 3*x + 1
-----------------------
  3        2
 x  + 2*x  - 9*x - 18
```

Now take this apart into partial fractions using `partfrac`:

```
% find the partial fractions
ratPartFrac=partfrac(ratSym,x)
```

```
ratPartFrac = (sym) 14/(3*(x + 3)) - 3/(x + 2) + 1/(3*(x
    - 3))
```

```
% show it with pretty
pretty(ratPartFrac)
```

```
     14          3          1
---------- - ----- + ----------
3*(x + 3)    x + 2    3*(x - 3)
```

Each of the three fractions above is a partial fraction.

(Yes, the `x` in the `partfrac` command must be specified.)

If you want to have some clue where the partial fractions come from, look at the factorization of the original ratio, and especially at the factors of the denominator:

```
% find the factorization
ratSymFactors=factor(ratSym)
% in Matlab, form the normal factored form using prod
%ratSymFactors=prod(ratSymFactors)
```

```
ratSymFactors = (sym) (x - 1)*(2*x - 1)/((x - 3)*(x + 2)
    *(x + 3))
```

```
% show it with pretty
pretty(ratSymFactors)
```

```
     (x - 1)*(2*x - 1)
-------------------------
(x - 3)*(x + 2)*(x + 3)
```

Now you can see that the partial fractions have factors of the denominator in their denominators.

FUNCTION MANIPULATIONS

Below are some more example manipulations involving functions.

For example, let's try to find the antiderivative of

$$\frac{1}{x\sqrt{ax^2+bx+c}}$$

I know the antiderivative can be found, for one because it is in my table book (Schaum's). Let's see how well Octave fares.

```
% make sure x, etc, are still symbolic
syms x a b c

% try integrating the example function
intHard1=int(1/(x*sqrt(a*x^2+b*x+c)),x)

% try again
intHard1=int(sym('1/(x*sqrt(a*x^2+b*x+c))'),x)
```

```
intHard1 = (sym) Integral(1/(x*sqrt(a*x**2 + b*x + c)),
   x)
intHard1 = (sym) Integral(1/(x*sqrt(a*x**2 + b*x + c)),
   x)
```

Useless. But the solution is in a basic table book! Well, Octave is free (and so is Sympy).

As another attempt, let's try to find the antiderivative of

$$\frac{1}{x(ax^2+bx+c)^{3/2}}$$

The integral is again in my table book.

```
% try integrating the second example function
intHard2=int(1/(x*(a*x^2+b*x+c)^(3/2)),x)

% try again
intHard2=int(sym('1/(x*(a*x^2+b*x+c)^(3/2))'),x)
```

```
warning: Using rat() heuristics for double-precision
   input (is this what you wanted?)
intHard2 = (sym) Integral(1/(x*(a*x**2 + b*x + c)**(3/2)
   ), x)
intHard2 = (sym) Integral(1/(x*(a*x**2 + b*x + c)**(3/2)
   ), x)
```

Oh, well.

Sometimes function `int` is useful to figure out the name of an antiderivative. For example, let's figure out what the antiderivative of e^{-x^2} is called:

```
% the antiderivative of exp(-x^2)
intExp_Minus_xSqr=int(exp(-x^2),x)
```

```
intExp_Minus_xSqr = (sym) sqrt(pi)*erf(x)/2
```

```
% Show with pretty
pretty(intExp_Minus_xSqr)
```

```
  _____
\/ pi *erf(x)
--------------
      2
```

Using `help erf` will tell you that function erf is called the "error function."

How about the antiderivative of $\sin(x)/x$?

```
% the antiderivative of sin(x)/x
intSinx_Over_x=int(sin(x)/x,x)
```

```
intSinx_Over_x = (sym) Si(x)
```

Function Si is called the "sine integral". Matlab uses the name `sinint`.

Suppose we want the Taylor series of an antiderivative. One way is to write the Taylor series of the original function and then integrate that. Below we demonstrate that for function Si:

```
% write the Taylor series of sin(x)/x
sinx_Over_xTaylor=taylor(sin(x)/x,x)

% lets have a few more terms than that
sinx_Over_xTaylor=taylor(sin(x)/x,x,'Order',10)

% integrate the Taylor series to get that of Si
sinintTaylor=int(sinx_Over_xTaylor)
```

```
sinx_Over_xTaylor = (sym) x**4/120 - x**2/6 + 1
sinx_Over_xTaylor = (sym) x**8/362880 - x**6/5040 + x
    **4/120 - x**2/6 + 1
sinintTaylor = (sym) x**9/3265920 - x**7/35280 + x
    **5/600 - x**3/18 + x
```

```
% show it with pretty
pretty(sinintTaylor)
```

```
      9          7       5     3
     x          x       x     x
 ------- - ----- + --- - -- + x
 3265920    35280    600    18
```

We can use that to figure out the logic of how to get each term from the previous one in the Taylor series of Si.

First put the individual terms in the Taylor series in an array called `terms`. Function `children` can do that:

```
% get the individual terms
terms=children(sinintTaylor)
```

```
terms = (sym) Matrix([[x, -x**3/18, -x**7/35280, x
   **5/600, x**9/3265920]])  (1x5 matrix)
```

However, you would probably prefer to have the terms indexed by the power of x, call it n, rather than by the term number. Call the renumbered terms $t_1, t_3, t_5 \ldots$. They are

```
% the renumbered terms
t1=terms(1)
t3=terms(2)
t5=terms(4)
t7=terms(3)
t9=terms(5)
```

```
t1 = (sym) x
t3 = (sym) -x**3/18
t5 = (sym) x**5/600
t7 = (sym) -x**7/35280
t9 = (sym) x**9/3265920
```

Now examine the ratio of successive terms t_n/t_{n-2}:

For n=3, that is `t3/t1`:

```
% the ratio t3/t1
ratio31=simplify(t3/t1)
factors18=factor(18)
```

```
ratio31 = (sym) -x**2/18
factors18 =
   2    3    3
```

Note that 2*3*3 equals $(n-1)n^2$.

For n=5:

```
% examine t5/t3
ratio53=simplify(t5/t3)
factors100=factor(100)
```

```
ratio53 = (sym) -3*x**2/100
factors100 =
   2    2    5    5
```

Note that 2*2*5*5 equals 4*5*5, again being $(n-1)n^2$. Also note that the 3 in the numerator equals $n-2$, and that that was 1 when n was 3.

For n=7:

```
% examine t7/t5
ratio75=simplify(t7/t5)
factors294=factor(294)
```

```
ratio75 = (sym) -5*x**2/294
factors294 =
   2    3    7    7
```

For n=9:

```
% examine t9/t7
ratio97=simplify(t9/t7)
factors648=factor(648)
```

```
ratio97 = (sym) -7*x**2/648
factors648 =
   2    2    2    3    3    3    3
```

In every case,

$$\frac{t_n}{t_{n-2}} = -\frac{(n-2)x^2}{(n-1)n^2}$$

So this seems to be generally true for all terms beyond the first.

Matlab has a function `funtool` to play with functions. This has not (yet) been implemented in Octave, at least not the older version I have. In Matlab, have fun!

```
% run funtool (can only run interactively, not in Octave)
%funtool
```