MATLAB / Simulink Lab Course

Symbolic Math Toolbox

UPDATED TO

R2019A

Lehrstuhl für
Flugsystemdynamik

Objectives & Preparation “Symbolic Math Toolbox"

- Which MathWorks products are covered?
  - ⇨ Symbolic Math Toolbox

- What skills are learnt?
  - ⇨ Usage of symbolic objects within MATLAB
  - ⇨ Symbolic calculus including differentiation, integration and vector analysis
  - ⇨ Solving symbolic algebraic equations and equation systems
  - ⇨ Solving symbolic differential equations and symbolic Laplace transform

- How to prepare for the session?
  - ⇨ MathWorks Tutorials:
    - https://de.mathworks.com/help/symbolic/getting-started-with-symbolic-math-toolbox.html?s_tid=CRUX_lftnav
    - https://de.mathworks.com/help/symbolic/create-symbolic-numbers-variables-and-expressions.html
    - https://de.mathworks.com/help/symbolic/create-symbolic-functions.html

Outline

1. Introduction
2. Symbolic Objects
   - Variables, Numbers, Expressions & Functions
   - Assumptions & Simplification
   - Variable Precision Arithmetic
3. Calculus
   - Differentiation & Integration
   - Vector Analysis
   - Series & Limits
   - Transforms
4. Equation Solving
   - Algebraic equations
   - Ordinary differential equations (ODEs)
5. Graphics
6. Code Generation
7. List of Useful Commands

UPDATED TO
R2019A

1. Introduction

Introduction

SYMBOLIC vs. NUMERIC

SYMBOLIC

- “you write it down”
- The symbol π stands for an infinite number
- $y = \sin^2 x + \cos^2 x$
can be simplified to
$y = 1$
- Computation of indefinite integrals
- **Results are always exact**

NUMERIC

- “you (or a computer) compute it”
- A finite number must be used to approximate π
- $y = \sin^2 x + \cos^2 x$
can be computed for different values of x. The result is always close to 1, depending on the numerical precision.
- Computation of definite integrals
- **Results are always approximate**

Introduction

Software for symbolic computations

MATLAB Symbolic Math Toolbox

Scope of this course

MuPAD

A computer algebra system originally developed at the University of Paderborn.

(Part of the Symbolic Math Toolbox since 2008.)

Interface

Mathematica

https://upload.wikimedia.org/wikipedia/de/b/bf/MathematicaSpikeyVersion8.png

Interface

Maple

Maple™

https://www.maplesoft.com/images2015/resources/Maple/Maple_2015_logo.jpg

Introduction – Toolbox Features

Symbolic Math Toolbox features:

- Analytically **manipulate and solve** symbolic math expressions.

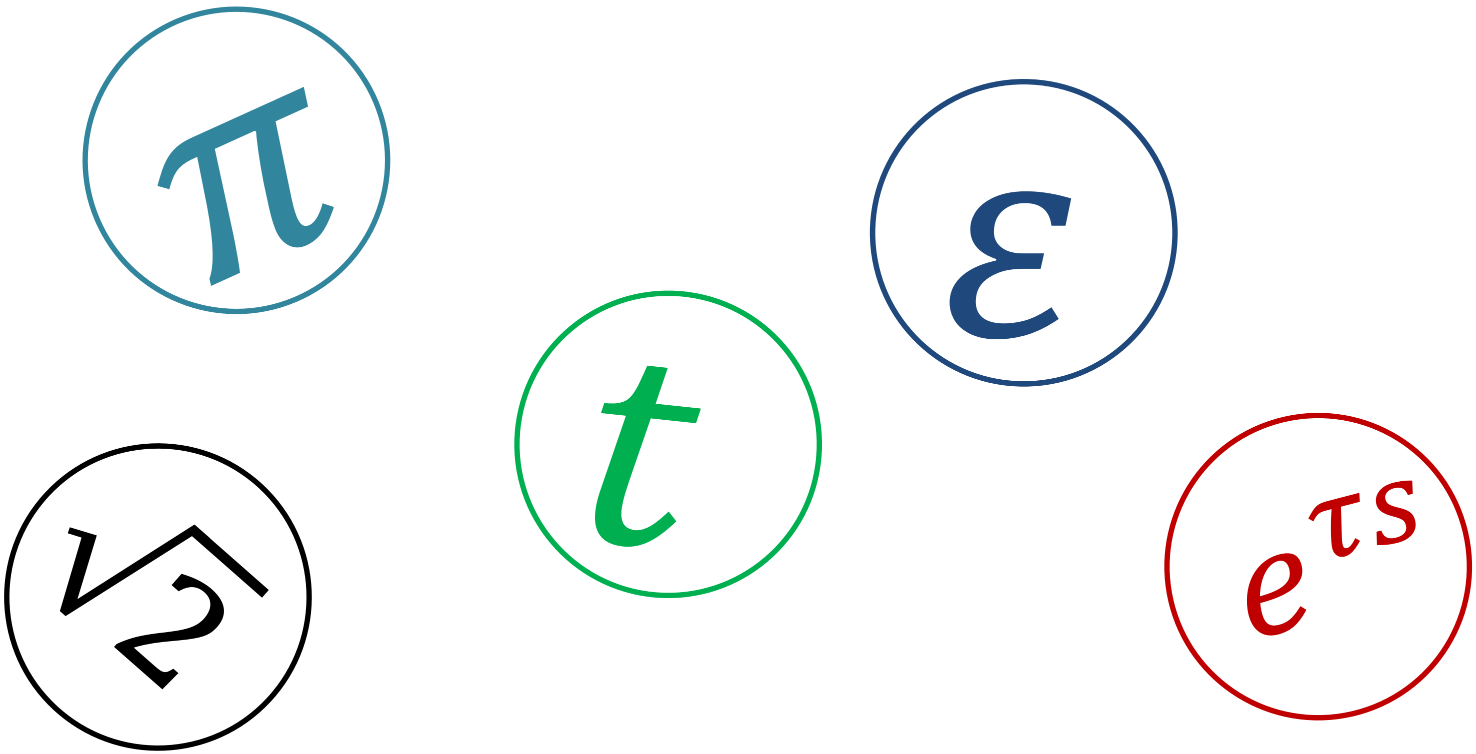
Simplification	Differentiation	Series	Transforms
Equation Solving	Integration	Limits	Vector Analysis

- Perform **variable-precision arithmetic (VPA)**.

Exact computations	VPA	Double-precision floating-point arithmetic
$1/3$	0.33 or 0.33333	$3FD5\ 5555\ 5555\ 5555_{16}$

- **Generate code** from symbolic expressions for MATLAB, Simulink, Simscape, C, Fortran, MathML, and TeX
- The Symbolic Math Toolbox includes the **MuPAD** language, which is optimized for handling and operating on symbolic math expressions.
  - Combinatorics, number theory, and other mathematical areas
  - MuPAD Notebook for performing and **documenting** symbolic calculations

2. Symbolic Objects



Symbolic Objects – Overview

The following four types of symbolic objects are distinguished.

Symbolic Variables `rho, s` MATLAB data type: sym	**Symbolic Numbers** `sqrt(2), 1/3` MATLAB data type: sym
Symbolic Expressions `distance = rho*exp(s)` MATLAB data type: sym	**Symbolic Functions** `distance(rho,s) = rho*exp(s)` MATLAB data type: symfun

Symbolic Objects – Symbolic Variables

- There are two ways to create symbolic variables:

```
x = sym('x');
y = sym('y');
```

or

```
syms x y
```

- MATLAB language vector and matrix notation extends to symbolic variables:

```
A = x.^((0:2)'*(0:2));
```

```
A =
[1,   1,   1]
[1,   x, x^2]
[1, x^2, x^4]
```

Note: in the MATLAB command window, symbolic results are not indented!

Symbolic Objects – Symbolic Variables

- There are two ways to create symbolic variables:

```
x = sym('x');
y = sym('y');
```

syms x

→ x → 'x'

or

```
syms x y
```

z = sym(2)

- MATLAB language vector and matrix notation extends to symbolic variables:

z =
2

```
A = x.^((0:2)'*(0:2));
```

```
A =
[1,   1,   1]
[1,   x, x^2]
[1, x^2, x^4]
```

z = 2 z

z =
→ 2

Note: in the MATLAB command window, symbolic results are not indented!

Symbolic Objects – Symbolic Matrices

- To create matrices of symbolic variables, either create the matrix elements and assemble them, or use the functionality of the `sym` command.

```
syms a b c d
A = [a,b; c,d];
B = sym('b', 2);
C = sym('val%d%d', [2,3]);
```

Note: only when using the first method, the individual matrix elements are created in the workspace.

val_i,j

```
A =
[a, b]
[c, d]
B =
[b1_1, b1_2]
[b2_1, b2_2]
C =
[val11, val12, val13]
[val21, val22, val23]
```

Example for operations with symbolic matrices:

```
A = sym('a',2);
det(A)
```

```
ans =
a1_1*a2_2 - a1_2*a2_1
```

Symbolic Objects – Assumptions

- It is possible to define assumptions about symbolic variables in several ways:

```
I = sym('I','integer');
P = sym('P','positive');
Q = sym('Q','rational');
syms R
assume(R,'real');
```

- To add further assumptions after the first one:

```
assumeAlso(I ~= 5);
```

- To check what assumptions are currently being made:

```
assumptions
```

```
[0 < P, in(Q, 'rational'), in(R, 'real'), I ~= 5, in(I, 'integer')]
```

- To remove an assumption:

```
assume(R, 'clear')
```

Symbolic Objects – Symbolic Numbers

- Symbolic numbers are created in a similar way:
  There are no floating point approximations – **symbolic numbers are exact!**

```
sum_of_angles_of_a_triangle = sym('pi');
one_third = sym('1/3');
two_fifths = sym('2/5');
one_third + two_fifths
```

```
ans =
11/15
```

- To convert symbolic numbers to a numeric representation:

```
s = sym(str2sym('sqrt(2)'))
s_numeric = double(s)
```

Note: The function **str2sym** has been introduced in R2017b for non-number entries.

```
s =
2^(1/2)
s_numeric =
     1.4142
```

Note the indentation! It indicates that a numeric representation is shown.

Symbolic Objects – Symbolic Numbers

- Symbolic numbers are created in a similar way:
  There are no floating point approximations – **symbolic numbers are exact!**

```
sum_of_angles_of_a_triangle = sym('pi');
one_third = sym('1/3');
two_fifths = sym('2/5');
one_third + two_fifths
```

```
ans =
11/15
```

s = ~~sym('~~sqrt(2)')

- To convert symbolic numbers to a numeric representation:

```
s = sym(str2sym('sqrt(2)'))
s_numeric = double(s)
```

```
s =
2^(1/2)
s_numeric =
    1.4142
```

Note: The function str2sym has been introduced in R2017b for non-number entries.

Note the indentation! It indicates that a numeric representation is shown.

Symbolic Objects – Numeric to Symbolic Conversion

- It is also possible to convert numeric values to symbolic numbers. For highest accuracy, be sure to use `sym` subexpressions instead of using `sym` on an entire expression:

```
s1 = 1/sym(234567)
s2 = sym(1/234567)
s3 = sym('1/234567')
```

Note: The string represents a number and can be entered without str2sym.

```
s1 =
1/234567
s2 =
5033067825897979/1180591620717411303424
s3 =
1/234567
```

This ratio is not exactly 1/234567, but it can be represented exactly by a double-precision floating point number.

- When using `sym` on an entire expression, the expression is converted to a floating point number first and this floating point number then is converted to a symbolic number.

Symbolic Objects – Numeric to Symbolic Conversion

- The conversion technique can be chosen by specifying a second function argument. It can be either `'r'` (default), `'f'`, `'e'`, or `'d'`.

```
s_rational = sym(0.1,'r')
```

```
s_rational =
1/10
```

- `'r'` stands for rational. 分数
  Floating point numbers obtained by evaluating expressions of the form
  - p/q,
  - $p \cdot \pi/q$,
  - $\sqrt{p}$
  - 2^q and
  - 10^q

  for modest sized integers p and q are converted to the corresponding symbolic form.

Symbolic Objects – Numeric to Symbolic Conversion

```
s_float = sym(0.1,'f')
s_eps   = sym(0.1,'e')
```

```
s_float =
3602879701896397/36028797018963968
s_eps =
eps/40 + 1/10
```

- `'f'` stands for floating point. The floating point value is captured exactly.
- `'e'` stands for estimate error. The `'r'` form is supplemented by a term involving the variable `eps`, which estimates the difference between the theoretical rational expression and its actual floating point value.

Note: `eps` is the distance from 1.0 to the next largest double-precision number.
`eps = 2^-52`

Symbolic Objects – Numeric to Symbolic Conversion

```
s_decimal = sym(0.1,'d')
```

```
s_decimal =
0.10000000000000000555111512312578
```

- `'d'` stands for decimal.
  This 32 digit result does not end in a string of zeros, but is an accurate decimal representation of the floating point number nearest to 0.1.

- The number of significant decimal digits can be changed using the `digits` command.
  Note the different result in the numeric to symbolic conversion!

```
digits(15)
s_decimal = sym(0.1,'d')
```

```
s_decimal =
0.1
```

Symbolic Objects – Numeric to Symbolic Conversion

- The following example illustrates possible consequences of non-exact floating point approximations. The signum or sign function is defined as follows:

$$\mathrm{sgn}(x) = \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x < 0 \end{cases}$$

Thus, $\mathrm{sgn}(\sin(\pi)) = 0$

```
numeric_incorrect  = sign(sin(pi))
symbolic_incorrect = sign(sym(sin(pi)))
symbolic_correct   = sign(sin(sym(pi)))
```

```
numeric_incorrect=
     1
```

num: very close to zero, but not exactly zero

```
symbolic_incorrect =
1
symbolic_correct =
0
```

In this case, `sin(pi)` is computed with floating point accuracy before being converted to symbolic. **The floating point approximation is not exactly zero!** This is the cause for the incorrect result.

Symbolic Objects – Symbolic Expressions

- Symbolic variables can be combined in symbolic expressions:

```
syms A omega t
d = A*sin(omega*t);
e = d^2 + (A*cos(omega*t))^2
```

```
e =
A^2*cos(omega*t)^2 + A^2*sin(omega*t)^2
```

- In symbolic expressions, each symbolic variable can be substituted by a numeric value or by another symbolic variable.

```
f = subs(e, [omega,t], [A,2])
```

```
f =
A^2*cos(2*A)^2 + A^2*sin(2*A)^2
```

- It is quite obvious that the expression for `f` can be simplified. The next slide shows how…

Symbolic Objects – Simplification

- Here is an example for simplification:

```
syms A omega t
e = (A^2*cos(omega*t)^2 + A^2*sin(omega*t)^2)^(1/2);
simplify(e)
```

```
ans =
(A^2)^(1/2)
```

Note: simplified expressions are always mathematically equivalent to initial expressions! Therefore, the result is not A.

- The toolbox can simplify expressions and functions with
  - polynomials,
  - trigonometric,
  - logarithmic, and
  - other functions (e.g., Gamma or Bessel function)

Note: assumptions can be helpful in this case. If A is assumed to be positive, the result of `simplify(e)` is indeed A.

- `simplify(f,'Steps',n)` with a positive integer `n > 0` can increase simplification of a complex expression `f`

Symbolic Objects – Symbolic Functions

- Symbolic functions are created by specifying the function variables and the function itself:

```
syms A omega t
f(A,omega,t) = A*sin(omega*t);
```

- A generic function and its variables can be created simply by:

```
syms g(x,y)
```

- Symbolic functions can be evaluated as follows, without using the command `subs`

```
f(3, 0.1*t, t)
```

```
ans =
3*sin(t^2/10)
```

- To find all symbolic variables in expressions, functions and matrices, use

```
symvar(f)
```

```
ans =
[A, omega, t]
```

Symbolic Objects – Variable Precision Arithmetic

- Symbolic numbers can be approximated with varying accuracy. Default accuracy, defined in significant decimal digits, can be retrieved or set with the `digits` command.

```
old_digits_setting = digits;
digits(3);
```

- The approximation itself is done using the `vpa` command:

```
v1 = vpa('1/2')
v2 = vpa('1/3000')
v3 = vpa(str2sym('sqrt(2)'), 20)
```

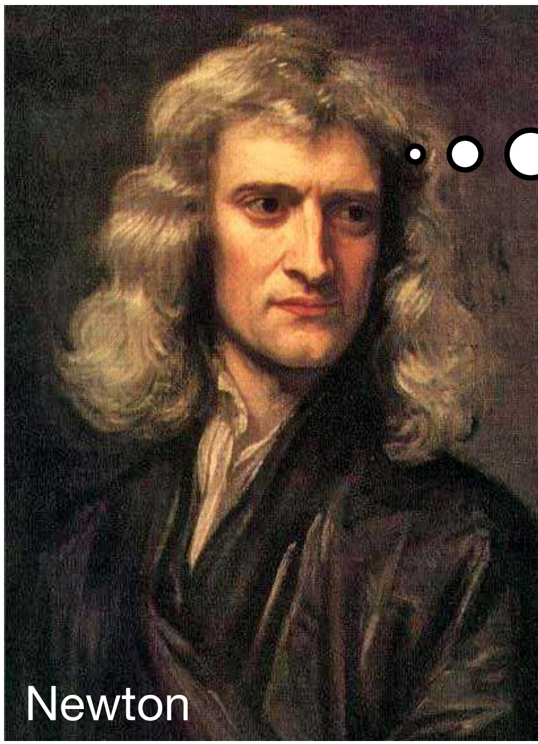
Note: the second argument of the vpa command substitutes the `digits` accuracy.

```
v1 =
0.5
v2 =
3.33e-4
v3 =
1.4142135623730950488
```

Note: accuracy cannot be read from individual numbers. (Accuracy is 3 here as well.)

VPA lets you trade off accuracy and code performance!

3. Calculus



$$\frac{dy}{dx}$$

$$\int y \, dx$$

https://upload.wikimedia.org/wikipedia/commons/3/39/GodfreyKneller-IsaacNewton-1689.jpg

https://upload.wikimedia.org/wikipedia/commons/8/8d/Christoph_Bernhard_Francke_-_Bildnis_des_Philosophen_Leibniz_%28ca._1695%29.jpg

Calculus – Differentiation

- Symbolic expressions or functions can be differentiated as follows:

```
syms A omega t x y
f = A*sin(omega*t);
g(x,y) = x^2 + x*y^2;
df_dt = diff(f,t)
ddf_dtdt = diff(f,t,2)
ddg_dxdy = diff(g,x,y)
```

```
df_dt =
A*omega*cos(omega*t)

ddf_dtdt =
-A*omega^2*sin(omega*t)

ddg_dxdy(x,y) =
2*y
```

Note how the function / expression character is conserved.

- Avoid not specifying the variable to be differentiated by! The `diff(f)` command would then assume the default variable, given by `symvar(f,1)`. Whatever this will be…

Calculus – Integration

- Similarly, you can perform indefinite and definite integrations (same functions f, g as above)

```
F = int(f, t)
Gdef = int(g, x, -3, 1)
```

```
F =
-(A*cos(omega*t))/omega
Gdef(y) =
28/3 - 4*y^2
```

Note that the integration constant is missing!

- By default, the integration function also considers special cases:

```
int(x^t, x)
int(x^t, x, 'IgnoreSpecialCases', true)
```

```
ans=
piecewise([t == -1, log(x)], [t ~= -1, x^(t+1)/(t+1)])
ans =
x^(t+1)/(t+1)
```

Calculus – Vector Analysis

- There are various functions for vector analysis, including `gradient`

```
syms x y z
m = x*y*z;
n = gradient(m, [x,y,z])
```

```
n =
y*z
x*z
x*y
```

- ...and `divergence`

```
p = [x, 2*y^2, 3*z^3];
q = divergence(p, [x,y,z])
```

```
Q =
9*z^2 + 4*y + 1
```

Other functions for vector analysis:
`curl`
`hessian`
`jacobian`
`laplacian`
`potential`
`vectorPotential`

Calculus – Series and Limits

- Examples of operations on series include symbolic summations

```
syms x k
s = symsum(x^k, k, 0, inf)
```

Remember that the geometric series is only finite for $|x| < 1$

```
s =
piecewise([1 <= x, Inf], [abs(x) < 1, -1/(x-1)])
```

Other functions for series and limits:
`cumprod`
`cumsum`
`pade`
`rsums`
`symprod`
`limit`

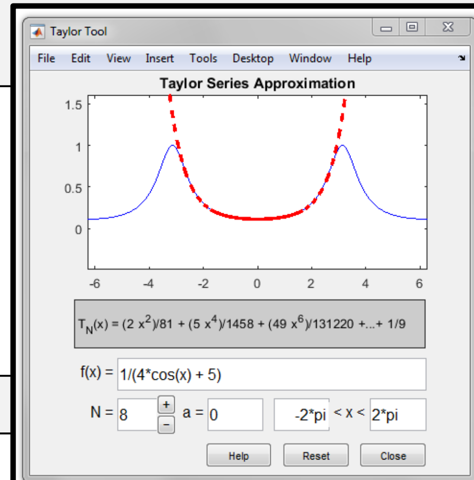
- …and taylor series

```
f = 1/(5 + 4*cos(x));
T = taylor(f, 'Order', 8)
```

```
T =
(49*x^6)/131220 + (5*x^4)/1458 + (2*x^2)/81 + 1/9
```

- Taylor series can also be graphically manipulated:

```
taylortool(f)
```



Calculus – Transforms

- You can also perform Fourier, Laplace and Z-Transforms

```
syms x s
f = 1/sqrt(x);
Lf = laplace(f, x, s)
```

```
Lf =
pi^(1/2)/s^(1/2)
```

- ...and the inverse of these transforms

```
iLf = ilaplace(Lf, s, x)
```

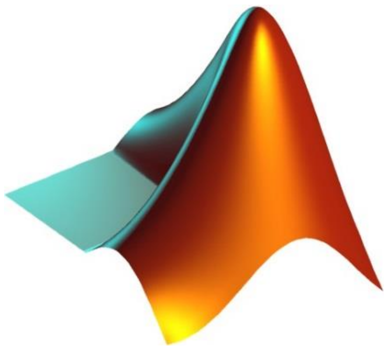
```
iLf =
1/x^(1/2)
```

Other functions for transforms:
`fourier`
`ifourier`
`ztrans`
`iztrans`

- These transforms can be used to solve differential equations.
  A more efficient way to do solve equations is shown in the following section.

4. Equation Solving

$$
\begin{aligned}
1+1 &= \sin^2(u)+\cos^2(u)+\ln(e)\\
&= \sin^2(u)+\cos^2(u)+\ln\left(\lim_{c\to\infty}\left(1+\frac{1}{c}\right)^c\right)\\
&= \sum_{n=0}^{\infty}\frac{\cosh(s)\cdot\sqrt{1-\tanh^2(s)}}{2^n}\\
&= \sum_{n=0}^{\infty}\frac{1}{2^n}\\
&= ???
\end{aligned}
$$



Linear and Nonlinear Equations and Systems

Introduction

SYMBOLIC SOLVER vs. NUMERIC SOLVER

`solve`	`vpasolve`
Returns exact solutions, which can then be approximated using `vpa`	Returns approximate solutions, whose precision can be controlled using `digits`
Returns a general form of the solution, providing insight into the solution.	Returns all/the first numeric solution(s) of polynomial/nonpolynomial equations.
Search ranges can be specified using inequalities.	Search ranges and starting points can be specified.
Can return parameterized solutions	Does not return parameterized solutions.
Equations may comprise parameters and inequalities	Runs faster than the symbolic solver.

Equation Solving – Algebraic Equations

- An equation can be defined and solved as follows:

```
syms a b c x
eqn = a*x^2 + b*x + c == 0;
sol_x = solve(eqn, x)
```

Advice: make assumptions on equation variables whenever possible. Thereby, only relevant solutions will appear!

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

- With the second argument of the `solve` function, specify the variable to solve for! Otherwise, `solve(eqn)` would solve for the default variable, given by `symvar(eqn,1)`.

- Here, the same equation is solved for a different variable:

```
sol_b = solve(eqn, b)
```

```
sol_b =
-(a*x^2 + c)/x
```

Equation Solving – Algebraic Equations

- The `solve` function does not return all solutions by default:

```
syms x
sol_x = solve(cos(x) == -sin(x), x)
```

```
sol_x =
-pi/4
```

- To return all solutions along with the parameters in the solution and the conditions on the solution, set the `ReturnConditions` option to `true`.

```
[sol_x, parameters, conditions] = solve(cos(x) == -sin(x), x, 'ReturnConditions', true)
```

```
sol_x =
pi*k – pi/4
parameters =
k
conditions =
in(k, 'integer')
```

Equation Solving – Algebraic Equations

- Moreover, it is possible to use the parameters and conditions to find solutions under additional conditions. For instance, to find values of $x \in (0, 2\pi)$

```
assume(conditions)
sol_k = solve(0 < sol_x, sol_x < 2*pi, parameters)
```

It is possible to hand over multiple equations and even multiple variables!

```
sol_k =
1
2
```

- To find values of x corresponding to these values of k, use `subs` to substitute for k in `sol_x`.

```
x_values = subs(sol_x, sol_k)
```

```
x_values =
(3*pi)/4
(7*pi)/4
```

Equation Solving – Systems of Algebraic Equations

- When solving for multiple variables at a time, `solve` returns a structure of solutions.

```
syms u v a
S = solve(u^2 - v^2 == a^2, u + v == 1, a^2 - 2*a == 3, [a, u, v])
```

```
S =
      a: [2x1 sym]
      u: [2x1 sym]
      v: [2x1 sym]
```

Specifying the variables is optional when the number of equations equals the number of variables.

The system of equations in a more readable form:

$$u^2 - v^2 = a^2$$
$$u + v = 1$$
$$a^2 - 2a = 3$$

- The system of equations has 2 solutions. The second solution can be visualized as follows.

```
s2 = [S.a(2), S.u(2), S.v(2)]
```

```
s2 =
[3, 5, -4]
```

Equation Solving – Systems of Linear Algebraic Equations

- Systems of **linear** algebraic equations can also be solved using matrix division.

```
syms u v x y
equations = [x + 2*y == u, 4*x + 5*y == v];
[A,b] = equationsToMatrix(equations, x, y)
S = A\b
```

```
A =
[1, 2]
[4, 5]
b =
 u
 v
S =
 (2*v)/3 - (5*u)/3
     (4*u)/3 - v/3
```

Remember: this expression solves $A \cdot S = b$ for S.

An alternative would be to use the MATLAB function `linsolve`:

```
S = linsolve(A, b)
```

Equation Solving – Numerical Solving: Polynomial Roots

- Equations can be solved numerically using `vpasolve`. This is especially useful to find polynomial roots.

```
syms f(x)
f(x) = x^2 - 2;
sol_x = vpasolve(f)
```

`vpasolve(f)` is equivalent to `vpasolve(f == 0)`

```
sol_x =
-1.4142135623730950488016887242097
 1.4142135623730950488016887242097
```

- Again, `digits` can be used to adjust the desired precision.

Equation Solving – Numerical Solving: Nonpolynomial Equations

- Nonpolynomial equations can be solved numerically as well. Consider the function $f(x) = e^x + e^{-x} - 10$, with a minimum $f(0) = -8$ and two roots.

```
syms f(x)
f(x) = exp(x) + exp(-x) - 10;
sol_x = vpasolve(f)
```

```
sol_x =
2.2924316695611776878007873113480
```

- Note that only one root has been found! To find the second root, provide an initial guess:

```
sol_x = vpasolve(f, -2)
```

```
sol_x =
-2.2924316695611776878007873113480
```

- It is also possible to make a random initial guess

```
sol_x = vpasolve(f, 'random', true)
```

The initial guess can also be a range. However, `vpasolve(f, [-3, 3])` only finds -2.29. To avoid this, combine the range with a random initial guess.

Ordinary Differential Equations (ODEs) and Systems of ODEs

常微分方程

Equation Solving – Ordinary Differential Equations

- ODEs can be solved using `dsolve`

```
syms x(t)
x(t) = dsolve(diff(x,t) == -5*x)
```

$$\frac{dx(t)}{dt} = -5 \cdot x(t)$$
$$\Rightarrow x(t) = C \cdot e^{-5t}$$

```
x(t) =
C5*exp(-5*t)
```

- Note that `dsolve` works with integration constants. They can be directly eliminated by specifying initial conditions:

```
syms x(t)
x(t) = dsolve(diff(x,t) == -5*x, x(0) == -3)
```

```
x(t) =
-3*exp(-5*t)
```

Equation Solving – Ordinary Differential Equations

- Higher order ODEs require multiple initial conditions. To be able to specify these, create additional symbolic functions like Du in this case.

```
syms u(b)
Du   = diff(u,b);
u(b) = dsolve(diff(u, b, b) == cos(2*b) - u, u(0) == 1, Du(0) == 0)
```

```
u(b) =
1 - (8*sin(x/2)^4)/3
```

- Nonlinear ODEs can have multiple solutions, even if initial conditions are specified.

```
syms x(t)
x(t) = dsolve((diff(x,t) + x)^2 == 1, x(0) == 0)
```

```
x(t) =
exp(-t) - 1
1 - exp(-t)
```

Equation Solving – Systems of Ordinary Differential Equations

- Systems of differential equations can be solved much like systems of algebraic equations.

```
syms f(t) g(t)
S = dsolve(diff(f,t) == 3*f + 4*g, diff(g,t) == -4*f + 3*g)
```

```
S =
     g: [1x1 sym]
     f: [1x1 sym]
```

- It is also possible to retrieve f and g directly:

```
[f(t), g(t)] = dsolve(diff(f,t) == 3*f + 4*g, diff(g,t) == -4*f + 3*g)
```

```
f(t) =
C2*cos(4*t)*exp(3*t) + C1*sin(4*t)*exp(3*t)
g(t) =
C1*cos(4*t)*exp(3*t) - C2*sin(4*t)*exp(3*t)
```

Equation Solving – Systems of Ordinary Differential Equations

- Systems of differential equations can also be solved in matrix form:

```
syms x1(t) x2(t)
A = [1 2; -1 1];
B = [1; t];
x = [x1; x2];
[x1, x2] = dsolve(diff(x) == A*x + B)
```

$$\dot{x} = Ax + B$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 \\ t \end{bmatrix}$$

```
x1 =

2^(1/2)*exp(t)*cos(2^(1/2)*t)*(C3 + (exp(-t)*(4*sin(2^(1/2)*t) + 2^(1/2)*cos(2^(1/2)*t)
+ 6*t*sin(2^(1/2)*t) + 6*2^(1/2)*t*cos(2^(1/2)*t)))/18) +
2^(1/2)*exp(t)*sin(2^(1/2)*t)*(C2 - (exp(-t)*(4*cos(2^(1/2)*t) - 2^(1/2)*sin(2^(1/2)*t)
+ 6*t*cos(2^(1/2)*t) - 6*2^(1/2)*t*sin(2^(1/2)*t)))/18)


x2 =

exp(t)*cos(2^(1/2)*t)*(C2 - (exp(-t)*(4*cos(2^(1/2)*t) - 2^(1/2)*sin(2^(1/2)*t) +
6*t*cos(2^(1/2)*t) - 6*2^(1/2)*t*sin(2^(1/2)*t)))/18) - exp(t)*sin(2^(1/2)*t)*(C3 +
(exp(-t)*(4*sin(2^(1/2)*t) + 2^(1/2)*cos(2^(1/2)*t) + 6*t*sin(2^(1/2)*t) +
6*2^(1/2)*t*cos(2^(1/2)*t)))/18)
```

5. Symbolic Plotting Functions

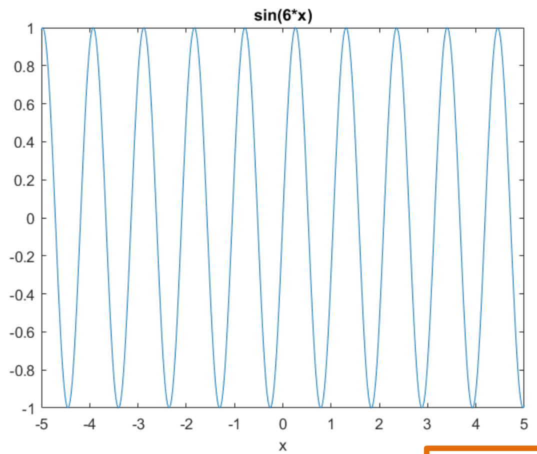
Symbolic Plotting Functions

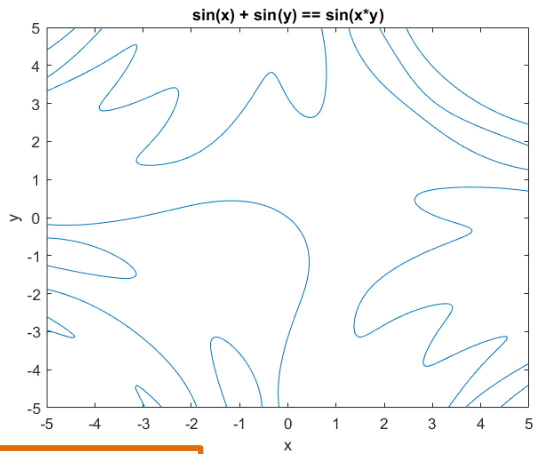
- Apart from the standard MATLAB plotting functions, there are also symbolic plotting functions. Here are very simple example (corresponding figure on the bottom left):

```
syms x;
fplot(sin(6*x));
```

- With `fimplicit`, it is possible to combine two variables and plot an implicit equation:

```
syms x y;
fimplicit(sin(x) + sin(y) == sin(x*y));
```



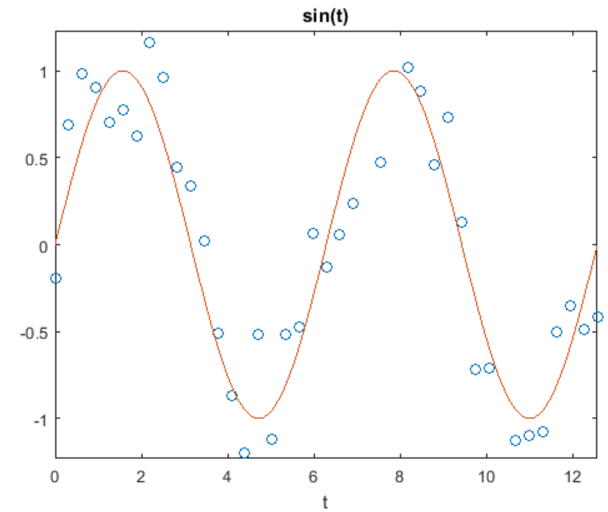


Title and labels have been added separately.

Symbolic Plotting Functions

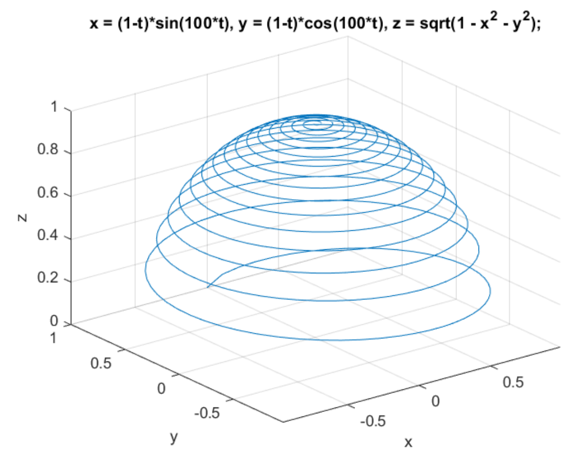
- Numeric and symbolic plots can be combined

```
x = 0:pi/10:4*pi;
y = sin(x) + (-1).^randi(10, 1, 41).*rand(1, 41)./2;
syms t;
figure; plot(x,y,'o'); hold on;
fplot(sin(t),[0, 4*pi]);
```



- 3-D symbolic plots are possible as well

```
syms t;
x = (1-t)*sin(100*t);
y = (1-t)*cos(100*t);
z = sqrt(1 – x^2 – y^2);
fplot3(x, y, z, [0 1]);
```

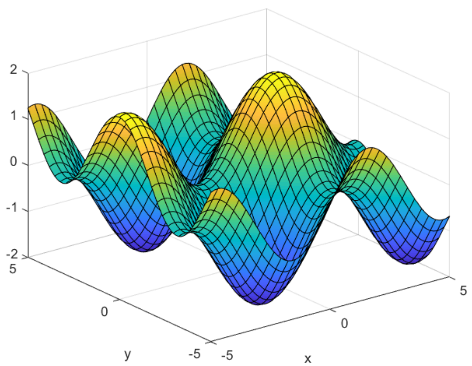


Title and labels have been added separately.

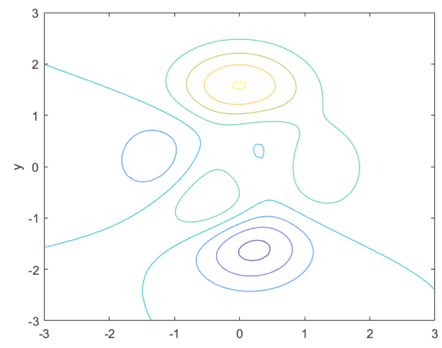
Symbolic Plotting Functions

- Examples for other symbolic plots:

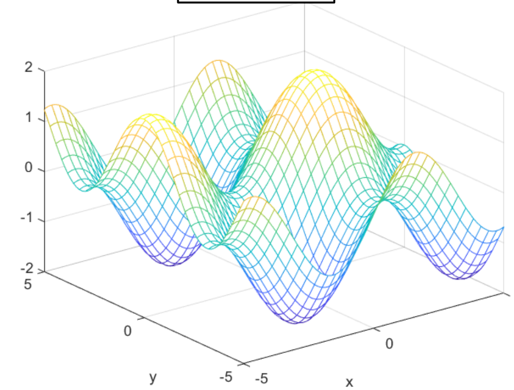
`fsurf`



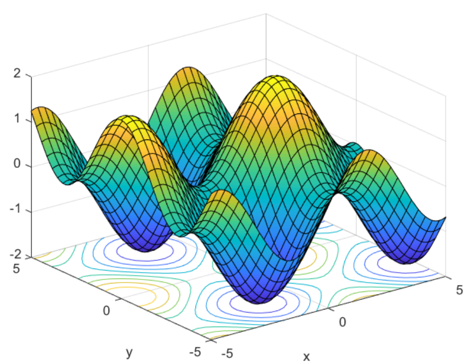
`fcontour`



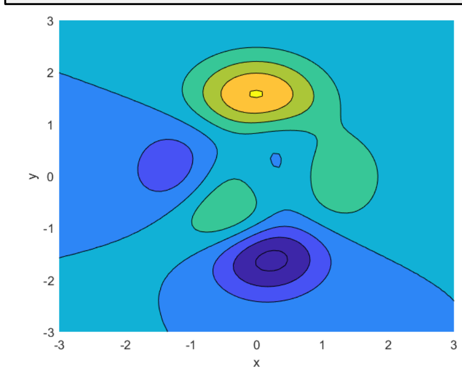
`fmesh`



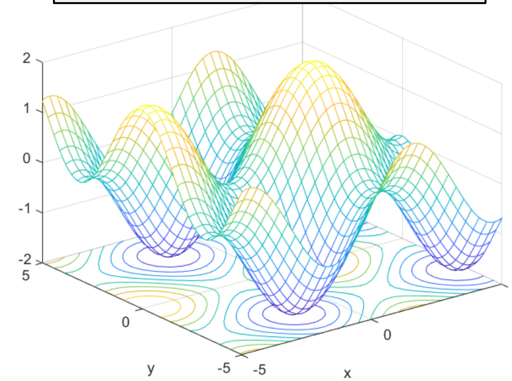
`fsurf`
`with 'ShowContours', 'on'`



`fcontour`
`with 'Fill', 'on','LineColor','k'`



`fmesh`
`with 'ShowContours', 'on'`



6. Code Generation

Code Generation – MATLAB Functions

- Symbolic expressions and functions can be converted into code (MATLAB, Simulink, Simscape, C, Fortran, MathML, and TeX).
  Here an example for the creation of a MATLAB function:

```
syms x y;
z = 30*x^4/(x*y^2 + 10) - x^3*(y^2 + 1)^2;
matlabFunction(z,'file','MExample.m');
```

- The file created has the following content:

```
function z = MExample(x,y)
%MEXAMPLE
%    Z = MEXAMPLE(X,Y)

%    This function was generated by the Symbolic Math Toolbox version 8.3.
%    01-Oct-2019 15:29:18

t2 = y.^2;
z = (x.^4.*3.0e+1)./(t2.*x+1.0e+1)-x.^3.*(t2+1.0).^2;
```

By default, optimized code is generated. (This intermediate variable makes the code more efficient.)

Code Generation – Simulink Blocks, C-Code

- Similarly, blocks can be created in a Simulink model. First, the model needs to be opened. Then, a block (type: MATLAB function) named `pythagoras` can be created as follows:

```
syms a b;
c = sqrt(a^2 + b^2);
matlabFunctionBlock('simulink_system/pythagoras',c);
```

- It is also possible to generate optimized C-Code:

```
syms x y;
z = 30*x^4/(x*y^2 + 10) - x^3*(y^2 + 1)^2;
ccode(z,'file','CExample');
```

- The output file looks like this:

```
t2 = y*y;
t0 = ((x*x*x*x)*3.0E+1)/(t2*x+1.0E+1)-(x*x*x)*pow(t2+1.0,2.0);
```

7. List of Commands

List of Commands

Command	Explanation	Slide #
sym	Create symbolic object	9
syms	Create symbolic object	9
assume	Make/clear an assumption	11
assumeAlso	Add assumptions	11
assumptions	Show assumptions	11
subs	Substitute	18
simplify	Simplify expression/function	19
symvar	Show symbolic variables	20
digits	Show/set signif. decim. digits	16, 21
vpa	Variable Precision Arithmetic	21
diff	Differentiate	23
int	Integrate	24
gradient	Compute gradient	25
divergence	Compute divergence	25
curl	Curl of vector field	-

Command	Explanation	Slide #
hessian	Hessian matrix	-
jacobian	Jacobian matrix	-
laplacian	Laplacian of scalar function	-
potential	Potential of vector field	-
vectorPotential	Vector potential of vec. field	-
symsum	Sum of series	26
symprod	Product of series	-
cumsum	Cumulative sum	-
cumprod	Cumulative product	-
taylor	Taylor series expansion	26
taylortool	Taylor series calculator	26
pade	Padé approximant	-
rsums	Riemann sums (interactive)	-
limit	Compute limit	-
sympref	Symbolic preferences	-

List of Commands

Command	Explanation	Slide #
`laplace`	Laplace transform	27
`ilaplace`	Inverse Laplace transform	27
`fourier`	Fourier transform	-
`ifourier`	Inverse Fourier transform	-
`ztrans`	Z-transform	-
`iztrans`	Inverse Z-transform	-
`solve`	Symbolic solver	31
`vpasolve`	Numeric solver	36
`equationsToMatrix`	Convert set of linear equations to matrix form	35
`dsolve`	ODE solver	39

Command	Explanation	Slide #
`fplot`	Symbolic 2D plot	45
`fplot3`	Symbolic 3D plot	46
`ezpolar`	Symbolic polar plot	-
`fsurf`	Symbolic surface plot	47
`fmesh`	Symbolic mesh plot	47
`fcontour`	Symbolic contour plot	47
`fimplicit`	Implicit function 2D plot	45
`fimplicit3`	Implicit function 3D plot	-
`matlabFunction`	Generate MATLAB function	49
`matlabFunctionBlock`	Generate MATLAB function block for Simulink	50
`ccode`	Generate C-Code	50