

# Differential Equations and MATLAB<sup>TM</sup>!

We have looked at phase planes of different systems in MATLAB<sup>TM</sup>. Now we are concerned with actually solving the systems of differential equations. Once again we will turn to some very nice code writing done by John Polking at Rice University<sup>1</sup>. But just before we do that, we want to discuss various ways of solving odes numerically.

There are many algorithms for computing the solutions to differential equations, for example Euler's method and the Runge-Kutta algorithms are just two of these. It is common to write your own code when solving differential equations. However, software programs such as MATLAB<sup>TM</sup> have their own solvers built in as functions. Or you can use a solver package such as the one we will soon use written by John Polking. If you are interested in writing your own codes, one reference to try is Garcia's Numerical Methods for Physics (I have a copy with me!).

Polking's code "odesolve" uses different MATLAB<sup>TM</sup> solvers in order to compute the solutions to initial value problems. MATLAB<sup>TM</sup> has several such ode solvers as functions. The following table<sup>2</sup> briefly illustrates what the different solvers are and when they should be used.

Solver	Problem Type	Order of Accuracy	When to Use
ode45	Nonstiff	Medium	Most of the time. This should be the first solver you try.
ode23	Nonstiff	Low	If using crude error tolerances or solving moderately stiff problems.
ode113	Nonstiff	Low to High	If using stringent error tolerances or solving a computationally intensive ODE file.
ode15s	Stiff	Low to Medium	If ode45 is slow because the problem is stiff.
ode23s	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
ode23t	Moderately Stiff	Low	If the problem is only moderately stiff and you need a solution without numerical damping.
ode23tb	Stiff	Low	If using crude error tolerances to solve stiff systems.

Attached at the end is a copy of the information under the different algorithms used for each of the different ode solvers<sup>3</sup>.

<sup>1</sup>Polking, web

<sup>2</sup>re-created from MATLAB<sup>TM</sup>'s Help under: MATLAB Functions: ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb

<sup>3</sup>Also taken from MATLAB<sup>TM</sup> Help under the "Algorithms" link from the previous function page

Although we are planning to use Polking’s solver package, notice that the table points out some different things we should have in mind while computing—these include the stiffness of the problem, and the order of accuracy. In other words, not all solution methods are appropriate for all uses. This is something that is easy to forget while approaching modeling problems!

Open your MATLAB<sup>TM</sup> window, making sure you are in the Public Server, cancer\_matlab directory (P:cancer\_matlab) and at the prompt type

odesolve

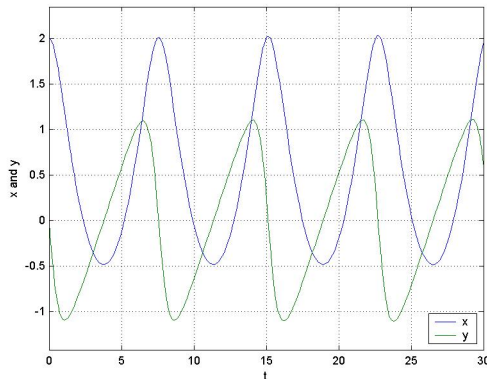
and hit return. A setup window should appear that looks like this:

The differential equations.			The initial conditions.		
x' = y			x = 2		
y' = y <sup>2</sup> · x			y = 0		
Number of equations.	Independent variable.	Solution interval.		t = 0	
2	t	0 <= t <= 30			
Parameters:					
	=		=		=
	=		=		=
Expressions:					
	=		=		=
	=		=		=
Solver: ode45		Output: time plot		all variables vs t.	
Quit		Add to the existing display		Solve in a new window	

Notice that the window opens to solve the system:

$$\begin{aligned} \frac{dx}{dt} &= y, \\ \frac{dy}{dt} &= y^2 - x, \end{aligned}$$

subject to the initial condition  $(x = 2, y = 0)$  at time  $t = 0$ , over the time interval  $0 \leq t \leq 30$ . It is setup to use the **ode45** solver and to output a plot of  $x, y$  versus  $t$ . Click on the button “Solve in a new window.” In a window entitled ODESOLVE Display, the following image should appear:



Take some time to play around with this problem. You can change the type of output (try the 3D plot with the default values of  $x, y$  and  $z$ !) and/or the type of solver used. (Does the solver make a difference in this problem? Why?)

**Back to our other sample problem:** When we investigated phase planes in MATLAB<sup>TM</sup> we visited the system<sup>4</sup>:

$$\begin{aligned}\dot{x} &= -y + ax(x^2 + y^2), \\ \dot{y} &= x + ay(x^2 + y^2).\end{aligned}$$

Now we will do so once again in order to see the solutions. In the ODESOLVE Setup window, enter the equations *with the  $a$  included*. When you use the proper MATLAB<sup>TM</sup> notation, you should have the following in your equations boxes:

The differential equations.	
$x'$	$-y + a*x*(x^2 + y^2)$
$y'$	$x + a*y*(x^2 + y^2)$

---

<sup>4</sup>Strogatz, 1994

DON'T worry about having the  $a$  in the equations. We can change the value of  $a$  in the parameter box. Let's try  $-1$  first. Enter  $a = -1$  in the parameter dialogue box. It should look like this:

The differential equations.		
$x'$	=	$-y + a^2(x^2 + y^2)$
$y'$	=	$x + a^2(x^2 + y^2)$
Number of equations.	Independent variable.	Solution interval.
2	t	0 <= t <= 30
Parameters:	a = -1	

Go ahead and solve the system for the given initial condition as well as the given time interval. Does it make a difference in this case which solver you use? Go ahead and try different values of  $a$  ( $0, 1$  may be appropriate for comparison with the phase plane work you did previously). If you do try  $a = 1$ , you may want to shorten the time interval. Notice that while the phase plane gave us the right idea about the behavior, we are now able to understand more about how the solution changes in time. For example, the decaying behavior when  $a = -1$  is much slower than the growing behavior when  $a = 1$ .

Once again, feel free to try more examples. Be creative! (Or use a book!)

### References

1. MATLAB Help under the Help button at the top of the MATLAB window.
2. Polking, John C. <http://math.rice.edu/~dfield/>
3. Strogatz, S. *Nonlinear Dynamics and Chaos*. Perseus Books Publishing, LLC., 1994.

## ALGORITHMS FOR MATLAB<sup>TM</sup>'s ODE SOLVERS

**ode45** is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a one-step solver - in computing  $y(t_n)$ , it needs only the solution at the immediately preceding time point,  $y(t_{n-1})$ . In general, **ode45** is the best function to apply as a "first try" for most problems.

**ode23** is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than **ode45** at crude tolerances and in the presence of moderate stiffness. Like **ode45**, **ode23** is a one-step solver.

**ode113** is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than **ode45** at stringent tolerances and when the ODE file function is particularly expensive to evaluate. **ode113** is a multistep solver - it normally needs the solutions at several preceding time points to compute the current solution.

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

**ode15s** is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like **ode113**, **ode15s** is a multistep solver. Try **ode15s** when **ode45** fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem.

**ode23s** is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than **ode15s** at crude tolerances. It can solve some kinds of stiff problems for which **ode15s** is not effective.

**ode23t** is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. **ode23t** can solve DAEs.

**ode23tb** is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like **ode23s**, this solver may be more efficient than **ode15s** at crude tolerances.