# Informal Introduction to Differential Equations

Chris Brown

June, November 2010, Dec. 2012, Feb. 2015

## Contents

# 1 Abstract

Originally for CSC160, this non-mathematical treatment of differential equations (DE)

1. assumes only elementary acquaintance with the concepts of derivatives and integrals; treatment is mostly graphical.
2. presents basic DE definitions (most unused)
3. shows how higher-order DEs can be represented as systems of simple (first-order) DEs.
4. presents the idea of numerical solution of DEs.
5. shows how to solve DEs using Matlab.
6. gives examples, along with Matlab code.
7. describes (in the appendix) pointers (indirection) and handles.

# 2 Motivation

*Differential equations* (DE's) express how related phenomena (variables) changing over time affect each other. That is, basically how everything in the world works.

**Physics:** Newton's second law: A body of mass $m$ subjected to force $F$ undergoes acceleration $a$ proportional to $F$ and inversely proportional to $m$: $F = ma$, also written $F = mx''$ or $F = m\ddot{x}$ or $F = m\frac{d^2x}{dt}$.

**Electrical Circuits:** The circuit of Fig. 1 has a resistor, capacitor, and inductor in series, and by appealing to various laws of physics (Ohm's law, Gauss's law, Faraday's law...), we can write a DE for the system: here is the DE for the current (it has an analytic solution).

$$\frac{d^2i(t)}{dt^2} + \frac{R}{L}\frac{di}{dt} + \frac{1}{LC}i(t) = \frac{1}{L}\frac{dv}{dt}$$

**Environmental Engineering:** Microbial growth rates and radioactive decay follow:

$$\frac{dX}{dt} = \mu X,$$

where $\mu$ is a decay rate or a rate of change of cell populations given by the cell-growth rate minus the cell decay rate, or $rX - dX$.

**Population Dynamics:** Predator-Prey equations (Vito Volterra).

Assume that a prey population $x(t)$ is controlled only by the predators, in whose absence it would increase exponentially. The predator population $y(t)$ is controlled only by the prey and with no prey would decrease exponentially. That is, with no interactions

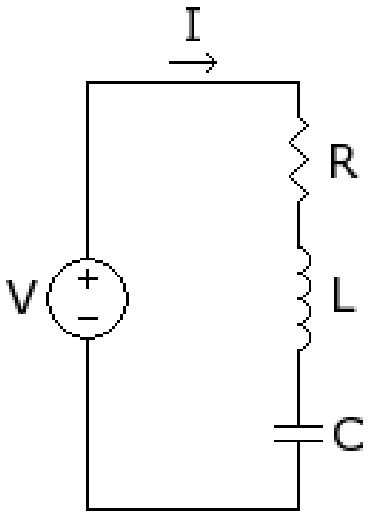



Figure 1: Circuit with voltage source, resistor, inductor, and capacitor in series.

$$x'(t) = ax(t)$$

$$y'(t) = -by(t).$$

Modeling interactions: 1) assume a meeting is unfavorable to prey. 2) measure number of meetings by $x(t)y(t)$, (so number goes up if one population goes up, for example). So we throw in some constants for rates that scale all these factors and get

$$x'(t) = ax(t) - cx(t)y(t)$$

$$y'(t) = -by(t) + dx(t)y(t)$$

These are nonlinear equations (the $x(t)y(t)$ terms,) so are not easy to solve analytically.

**Finite element** methods model large systems (an automobile body, the atmosphere) by a collection of smaller ones (e.g. a grid of plates or volumes) whose local interactions are modeled by differential equations. It can take tens of hours of computer time to model a 40-millisecond car crash, and weather prediction is not an exact science.

So solutions to DEs are important. Several types of DE can be solved by analytical methods, but we don't touch on that here: it's the stuff of a traditional math course in DEs. Thus, even though we don't want to be pure 'app users', we must pass almost entirely over the mathematics of DEs. Our goal here is just to understand and do run-of-the-mine routine problems, using built-in "DE solvers." For future cutting-edge work and trouble-shooting we'll need to get beyond being pure 'user's.

In the coming years you'll be learning how to model natural and man-made systems with differential equations: it's like algebra word problems – we need to express the relevant processes, laws, constraints, and interactions as differential equations. We're also not going

into that basic skill at all here (but the Chemical Engineering option in the assignment broadly illustrates the process.)

Many DEs and systems of DEs cannot be solved analytically. We resort to "numerical methods". *Mathematica* and the symbolic math package in Matlab can do analytical solutions as well as numerical methods. For lots of math problems, including solving nonlinear (systems of) equations (algebraic and differential), we have to do approximation or search or both, yielding approximate answers in some finite time. The field of *Numerical Analysis* studies and tries to improve the accuracy, speed, convergence, etc. of numerical algorithms: it's a deep field.

What we'll see below:

1. some examples of first-order ODEs, higher-order ODEs, and systems of ODEs.
2. implementation of the easiest numerical algorithm, Euler's method.
3. general principles for how increasingly sophisticated numerical solutions work.
4. how to use a particular Matlab ODE solver for first-order initial-value ODEs and systems of ODEs.

What we won't see:

1. How (in terms of the relevant laws of nature) to set up DEs to describe a particular problem.
2. Anything much about known analytical solutions, other common sorts of DEs, anything "mathematical."

# 3 Differentiation and Integration

Here we leave real mathematics aside and go for geometric intuition, so nothing in the sequel is 'really' mathematically true. Trust your calculus professor. What follows is basically just the Fundamental Theorem of the Calculus (FTC) (Fig. 2).

Differentiating a function amounts to computing its slope for any point. The slope shows how fast it is headed up or down. The bottom function has a range of constant positive slope followed by a range of 0 slope. A plot of these slopes is something like the top function, which has a range of constant positive value followed by a range of zero.

Integrating a function amounts to calculating "the area under its curve". Integration describes the area calculation as the sum of a number N of strips whose height is the function's height and whose width gets less as N increases. Then we let N go to infinity, getting a "continuous sum" or integral.

In computing, integration (unless you're in a symbolic math package) always just means "Summation" and usually iteration, and is just another example of an "accumulator algorithm" like computing the sum or product of the first N integers with a `for` loop.

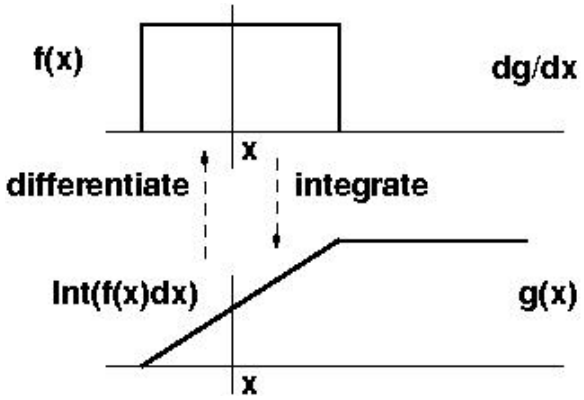



Figure 2: Top: a "boxcar" function, the derivative of the ramp function below. Bottom: a "ramp" function, the integral of the boxcar function above.

So, the top function has two ranges where slope= 0. Imagine pushing a vertical line through the function from left to right and recording all the area it has swept out to a given point x. That is the integral from zero to x, and is given by the bottom function.

Sometimes the integral is called the anti-derivative: they're inverse operations (almost – notice that adding a constant to the bottom function shifts it up or down but does not change its slopes.)

# 4 Solving Differential Equations

Again, this is **NOTHING** like a real mathematical treatment. Trust your engineering and math professors.

If you have an equation of the special form

$$y'(t) = g(t),$$

then the reasoning of the last section applies. Use the FTC and integrate both sides. This process is called 'quadrature', a word first appearing in 1500's in the context of approximating curves areas (especially the circle) with squares. You get the solution: $y(t)$ expressed as the integral of the right hand side. By extension, we often talk about solving DEs as "integrating" them, even if it's an approximate "numerical integration" calculation.

**Examples:** Recall $\frac{d(ax^n)}{dx} = nax^{n-1}$ . Therefore if you are given the DE

$y' = nax^{n-1}$ you could say the solution was $y = ax^n$. But since the derivative of constant $C$ is zero, you could add an arbitrary constant to your answer and still have a correct solution.

Or, you recall $sin(x)' = cos(x)$ (Fig. 3).

So if you see the DE $y' = cos(x)$ you know $y = sin(x) + C$.

You also recall $\frac{de^{ax}}{dx} = ae^{ax}$,

so if $y' = ay$ you know $y = Ce^{ax}$, another family.

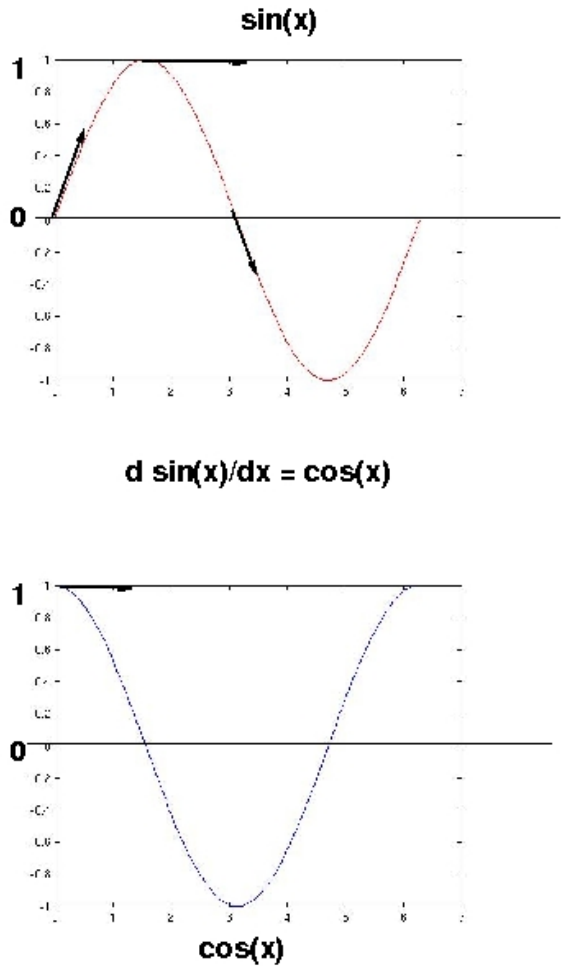



Figure 3: Sine and cosine (of x), with some slopes (d/dx) suggested. The slope (derivative) of the sine is given by the cosine. Is the derivative of the cosine related to the sine?

In general, a DE has a family of solutions: our additive offset is the simplest manifestation of this phenomenon. In general the families are much more interesting. In Fig. 4, the left side shows a made-up function varying by an additive constant, and on the right a sample family from an equation of the form $y' = ay - by^2$.

Sometimes we want the family, sometimes we need a specific solution. *Boundary Values* (BVs) give the value of the $y$ or $y'$ for some $x$s, and enough BVs can determine one particular solution. *Initial Value* problems (IVPs) are common: *Initial Conditions* give the value of $y$ and $y'$s for $x$ (or $t$) = 0 and so describe what happens in the future given this initial state of the system. Two-point BV problems can, for instance, specify starting and ending states. We'll be dealing exclusively with IVPs.

Things go pretty steeply uphill after about this point (That is, page 3 of most DE texts) and by the time you get to solving something as simple as $y' = y + x$ analytically, you'll be asked to remember integration by parts and you'll want to understand the notion of an integrating factor (a DE solving thing). That's why we have courses in DEs. Or numeric solutions, of course...

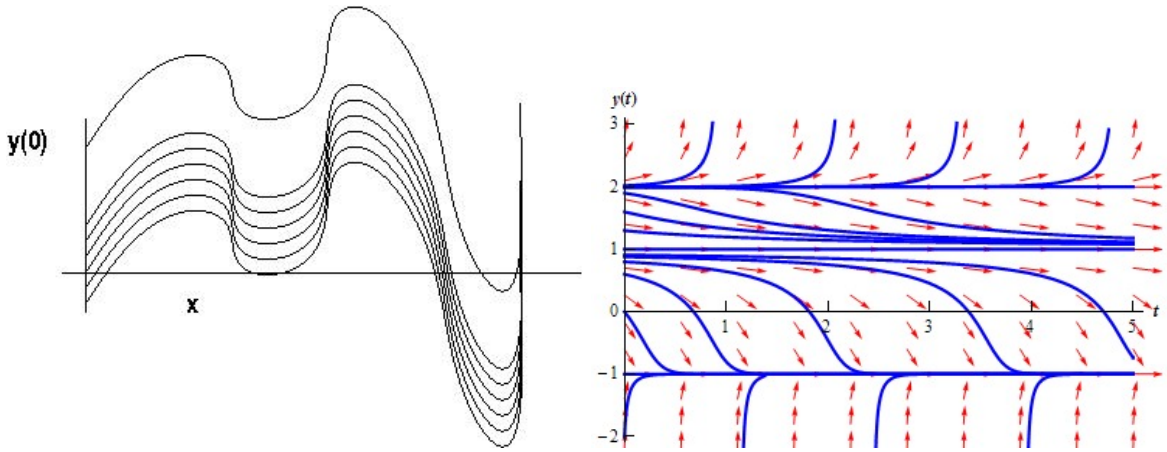



Figure 4: Families of DE solutions.

# 5 A Little Terminology

Not that we'll need all this, but it places what we're doing in some sort of taxonomic context.

**Ordinary and Partial DEs (ODEs and PDEs)**. ODEs have ordinary derivatives only (deal with one-dimensional functions), while PDE's can have partial derivatives. PDEs are harder, and we only deal with ODEs.

Example PDE: the electromagnetic wave equation (related to the famed Maxwell's equations). Those curly d's are partial derivative operators: something to look forward to.

$$\left(\nabla^2 - \mu\epsilon\frac{\partial^2}{\partial t^2}\right)\mathbf{E} = 0$$

$$\left(\nabla^2 - \mu\epsilon\frac{\partial^2}{\partial t^2}\right)\mathbf{B} = 0,$$

where $B$ is the magnetic field, $E$ the electric field, $c = \frac{1}{\sqrt{\mu\epsilon}}$ is the speed of light in the medium, and $\nabla^2$ is the Laplace operator, itself a second-order differential operator representing flux density of gradient flow.

**Linear and nonlinear DEs**. A linear DE in $y$ is a sum of weighted y-derivatives of different orders set equal to zero: e.g.

$$a(x)y' + b(x)y + c(x) = 0.$$

If $c(x)$ is always 0, it's a *homogeneous* equation, *inhomogeneous* otherwise. Now we don't particularly care about this homogeneity issue but it's basic to the general study of DEs: a solution to the homogeneous form of the equation may be added to any solution to the original equation and the result is also a solution to the original equation. For example a homogeneous equation can describe a mechanical or chemical system at equilibrium, and

the $c(x)$ can represent a disturbing influence that affects the system. In Fig. 1, the equation describing the circuit is homogeneous if the voltage is off, inhomogeneous if it is on. The results of a disturbance on the response of the mechanical system of Fig. 6 is shown in Fig. 7.

Linear DEs can have *constant coefficients* or coefficients that are functions of time. Linear ordinary constant-coefficient DEs were first solved by Euler (mid-1700s), who saw that all solutions were of the form $e^{kx}$ (written in some contexts $e^{st}$), where the $k$ and $s$ may be complex. These solutions form a family of decaying, exploding, or sinusoidally-oscillating functions.

**Order of a DE** is highest order of derivative that appears in it. So a first-order equation looks like

$$\frac{dy}{dt} = y' = f(t, y),$$

where $f(t, y)$ is a continuous function. We'll see that a higher-order DE can be expressed as a *system* of first-order DEs.

Preview: we'll deal below with numerical solutions of initial value problems for linear and nonlinear first- and higher- order ODEs and systems of the same.

# 6 Higher-Order ODEs and Systems of First-Order ODES

Here's an example third-order ODE.

$$y''' + g(x)y'' + r(x)y' + s(x)y = t(x)$$

It, and indeed any Nth-order ODE, can be rewritten as an equivalent system of N first-order ODEs. Using matrices is convenient: The equation above is given by the bottom row of this matrix equation, and the (pretty simple!) equality of $\frac{dy}{dx}$ to $y'$ and $\frac{dy'}{dx}$ to $y''$ are asserted in rows 1 and 2 (since we need to give $\frac{d}{dx}$ for the whole vector $[yy'y'']^T$.)

$$\frac{d}{dx}\begin{bmatrix} y \\ y' \\ y'' \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -s(x) & -r(x) & -g(x) \end{bmatrix}\begin{bmatrix} y \\ y' \\ y'' \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ t(x) \end{bmatrix} \tag{1}$$

The matrices for higher-order ODEs will all have the same "Identity Matrix" look in their upper rows. Systems of ODEs with this very special form of matrix are said to be in `phase canonical form`.

You may see the same trick done in a slightly different way, so be prepared. We write first-degree equations in terms of renamed variables, most of which are thus derivative functions in disguise. It's nice when the derivatives already have names we recognize:

$$my'' = g(t) - by' - cy,$$

for instance, could be describing a situation involving variables whose values are acceleration $a = y''$, velocity $v = y'$, and position $y$:

$$ma = g(t) - bv - cy.$$

It isn't very confusing to consider this last equation as a first-order DE in $v'$ (acceleration): $mv' = g(t) - bv - cy$, and we also can write a first-order equation in $y'$ (velocity) from its definition: $v = y'$. These two first-order equations form a system that is equivalent to the original second-order one.

Here's a bigger, more general example:

$$y''' + g(x)y'' + r(x)y' + s(x)y = t(x)$$

Solving this for $y$ has to be equivalent to solving these three equations for $y, u, v, w$:

$$\begin{array}{rcl} u(x) & = & y'(x) \\ u'(x) & = v(x) = & y''(x) \\ u''(x) & = w(x) = & y'''(x) = t(x) - s(x)y - g(x)v(x) - r(x)u(x) \end{array}$$

These equations are all in the form $z' = f(x, z)$: here $z$ is a "state vector": $z = (v, u, y) = (y'', y', y)$. $f(x, z)$ is necessarily a vector, too. So for the current example, $f(x, z)$ is $Az$, where A is the matrix in Eq. (1). This form is one of the two formulations of the DE that basic ODE solvers accept (and the only one we'll consider).

Going back to the first (matrix) formulation, it's easy to imagine filling in the matrix elements arbitrarily – not having just that shifted identity-like matrix in the first order-1 rows. That allows more general, still-linear systems.

Finally, we can have systems of nonlinear equations. For instance, here is a form of the *Lorenz Attractor* (see Wikipedia on "the Butterfly Effect" and "Lorenz Attractor"). $x, y, z$ are all functions of time that depend on each other in nonlinear ways (note the $xz$ and $xy$ multiplications):

$$\begin{array}{rcl} x' & = & 10(y - x) \\ y' & = & -xz + \rho x - y \\ z' & = & xy - 8z/3 \end{array}$$

The values of the constants 10 and 8/3 go back to Lorenz's original work in 1960's: other values don't change the qualitative nature of the solutions. $\rho = 28$ is interesting because small variations around it cause major variations in the (x,y,z) trajectory of the point. Thus it is a chaotic region: you can test this by setting $\rho = 28$ and starting out the integration with slightly different initial conditions (really slightly) for $x(0), y(0)$, or $z(0)$.

In any event, what we're going to need for anyone's ODE solver Matlab, Mathematica, whoever... is those equations of the form

$$y' = f(t, y)$$

with $y'$ and $f(t, y)$ both vectors. Here we're differentiating with respect to $t$.

We supply a few vectors (initial conditions, a range of times, answer vectors, etc. and the derivative function (in Matlab, in the form of a function that computes $y'$ for each variable $y$ in the system of equations). The ODE solver 'does the rest' (sometimes a big lie, but that's its goal and our wish).

# 7 Numerical Integration: The Idea

How can we solve an equation $y' = f(t, y)$ that is not susceptible to analytical solution? That is, how can we get (or approximate) $y(t)$?

Suppose (as we always shall in this tutorial) that we know $y(0)$, $y$'s `initial value`. The equation for $y'$ is sitting there telling us explicitly how much $y$ changes for a small change in $t$. That's what we pay it for. So if we wanted to compute $y$ for a small $t$ (call it $dt$) close to 0, and we know $y(0)$, why not say the answer was $y(dt) = y(0) + y'dt$?

Well, there's no good reason why not, and in fact that's a pretty good idea (Fig 5 A).

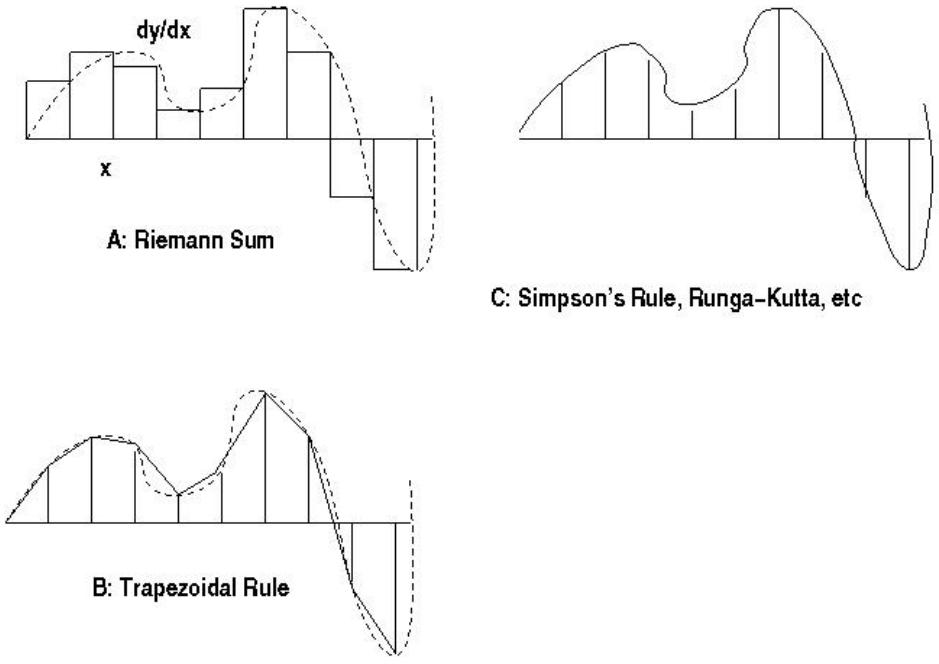



Figure 5: A: Euler quadrature: add up the rectangles. B: Trapezoidal rule: add up the trapezoids for a better fit: C: Higher-order methods: use more points to get a better estimate of function and its derivatives for better fit.

Our idea is in fact known as the Euler method for "integrating an ODE". It simply repeatedly adds the area of small strips whose height is the height of the one before, plus the change in $t$ times $y'$ at that $t$.

So given we have a value $y(0)$, the differential equation $y' = f(t, y)$, and a step size $h$, we can increment the time $t$ by $h$-steps and think of $y$ and $f$ as time-varying functions (or

vectors whose indices represent different times). Here, time moves along in discrete steps so can be thought of as an index if we want to think of $t, y, f$ as vectors. We can set $t(0)$ to 0 and

$$\begin{aligned} t(n+1) &= t(n) + h \\ y(n+1) &= y(n) + h * f(t(n), y(n)). \\ &\ldots \end{aligned}$$

The first equation says that time is stepping along from one instant to the next by the fixed interval $h$, and the second says that the new value for $y$ is approximated by the last value of $y$ plus time interval $h$ times the value of the derivative $y' = f(t, y)$.

This process converges to the exact solution as $h$ goes to zero (Cauchy 1824).

Now consider the matrix equation (1) in Section 6. We can sort of visualize how that system might be integrated. First, note that to get started, we know several things. We know the initial value $x_0$ where we want to start the integration so we know $t(x)$. We know the initial value of the state vector (lower derivatives of $y$): $(y, y', y'')$. But this means we know both the rightmost vectors in eq. (1). That means we can immediately use the third row's equation to solve for $y'''$. For the first "timestep" (or "x-step") $\Delta x$, we use $y'''$ and the Euler method to solve for $y''$; then $y'', \Delta x$ and the Euler method solve for $y'$; then $y', \Delta x$ and the Euler method solve for $y$. At this point we have the new state $(y, y', y'')$ at time $x_0 + \Delta x$, and are ready to go around again. We can almost see the `for-loop` operating over that matrix equation. *Luckily we don't have to see it, much less write it! Matlab does it all "for free".*

That's coming up. For now, let's get back to numerical integration. We can get a better approximation by taking the average (interpolate between) two neighboring $f(t, y)$ values (this method is called the *Trapezoidal Rule*; Fig. 5 B).

$$\begin{aligned} t(n+1) &= t(n) + h \\ z(n+1) &= y(n) + hf(t(n), y(n)) \\ y(n+1) &= y(n) + (h/2)[f(t(n), y(n)) + f(t(n+1)z(n+1))] \end{aligned}$$

Further, we can use Simpson's rule (2nd semester calculus concept) to fit three neighboring points with a parabola to get an even better solution (Fig. 5 C).

**Example:** A very useful and common system (often it's a good approximation to more complicated ones) is the second-order equation describing the electrical circuit in Section 1, which also describes the "mass, damper, spring" mechanical system of Fig. 6: it is a weight suspended by a spring, with a dashpot or damper (source of friction) in parallel with the spring.

Something is also pushing and pulling (driving) the weight as a function of time $g(t)$ (for instance, gravity). The equation is

$$My'' + Cy' + Ky = g(t),$$

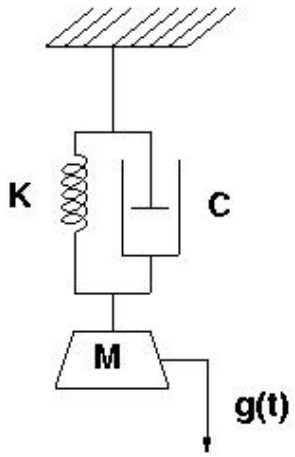



Figure 6: A Mass-spring-damper system.

which says that acceleration will be inversely proportional to mass (F = ma) Friction in this example is proportional to velocity (not all friction is: e.g. "sticktion" is the frictional resistance offered by something at rest to getting it started moving). The spring's force is proportional to its displacement from equilibrium. In an electrical analogy, the spring works like a capacitor and the damper like a resistor.

Ideally, without friction or any driving function, if you displaced the weight from equilibrium and let it go, or gave it an initial velocity, you'd expect it to keep bouncing forever and you might suspect (or remember from high-school physics) that its position (and velocity) through time would vary sinusoidally. With a damper the amplitude of the oscillations would die out. The astute reader will remember from Section 5 that the solutions for constant coefficient systems like ours are exponentials, and so realize that both the sinusoidal and the exponential-decay versions of exponential functions appear in this solution (Fig. 7), which was produced by the Euler integration code below.

**Important Note:** It is always best, or even always necessary, to be able to check your solution against a known one. We can do that for constant coefficient systems, and for several others. However, we wouldn't need numerical integration if everything was solvable in closed form. Then it is important to find simplifications of your system (maybe special initial conditions or constants that can be set to convenient values) that yield cases you can solve, or whose answers you can predict more or less. Getting 'the right answer' for such simple cases using your more-general solution method can make you (and your reader) more confident that your general solution might actually work.

```
function y = eulmsb(m,k,s,y0,yp0)
%  euler numerical integration of mx'' +kx' +sx = g(t)
% with initial conditions y(0) and y'(0)

N = 500 % steps and answers
```

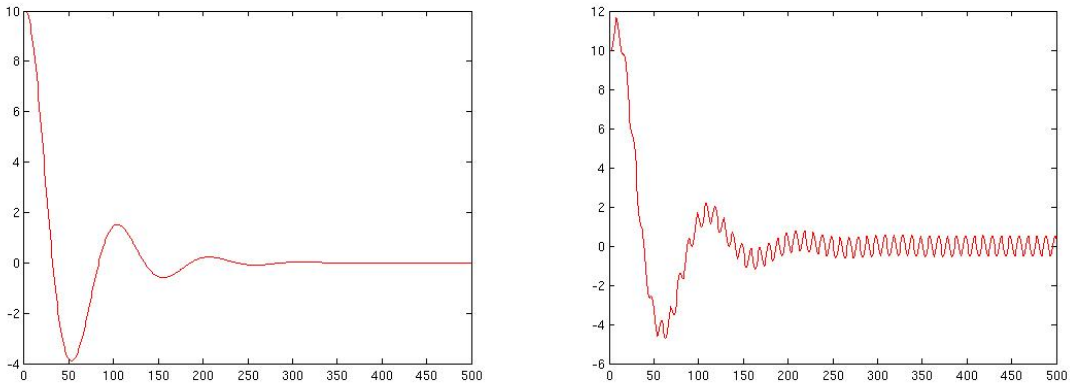


Figure 7: Response (position of the mass) of the Mass-spring-damper system with *g(t)* constant and with *g(t)* a small high-frequency sinusoidal driving function.

```
y1 = zeros(1,N);  % y(t) answers to be computed
y2 = zeros(1,N);  % intermediate  y'(t) answers computed
InitVal = 0;  % between 0
FinalVal = 100; % and here

Range = FinalVal-InitVal;
h = Range/N;   %step size
x = 0.0;
y1(1) = y0;   % initial condition for y
y2(1) = yp0;  % initial condition for y'
y2a = yp0;    % IC and intermed. ans. for y' not saving y'(t)
for i = 2:N
   x = x+h;
%    y1(i) = y1(i-1)+ h*y2(i-1); %keep y'(t)
%    y2(i) = y2(i-1)+ h*(y1(i-1)*(-s/m) +y2(i-1)*(-k/m)) +g(i-1);

    y1(i) = y1(i-1)+ h*y2a;    %only keep latest y'(t)
    y2a = y2a+ h*(y1(i-1)*(-s/m) +y2a *(-k/m)) +g(i-1);

end
plot(y1, 'r');
y = y1;
end

function driven = g(k)  % Produces right-hand graph Fig.7
driven=  sin((k/10)*(2*pi));
% driven = 0;
end
```

This code may be worth studying. It shows how the second-order equation is broken up

into two first order ones, how they are separately integrated and how they depend on each other. If we don't care about the history of velocities we don't need to keep each answer $y'(t)$, just the latest one so we can compute the current `y2a` (hence the commented out statements in the for loop).

*Runga-Kutta* ODE integration methods generalize the interpolation idea: you take small trial sub-steps across the integration step, use the value of $y$ there to adjust the derivative for the next sub-step. A very common choice is rk4, which uses four sub-steps. MatLab has many ode-solvers, including an improved rk4 that is not quite an rk5, called rk45. Likewise an rk23. For now we'll just pick one (rk23 in fact) but there are so many choices because different sorts of actual systems lead to systems of DEs that need different numerical solution approaches – it's a big and sophisticated field. For example, "stiff" systems have solution components that vary with different time scales (rather like the driven system of Fig. 7), so one needs to adapt a smaller step size near the quickly changing areas. `http://www.scholarpedia.org/article/Stiff_systems` shows (along with quite an article) a schematic depiction of variable step-size integration of a step function, where the step size increases when nothing is going on but decreases when things get interesting: it beats the straight Euler approach (here, same answer, fewer steps).

For us as engineers DEs are not normally a subject of study in themselves (as they can be for mathematicians). However it is best to know mathematically what is going on so you can troubleshoot, interpret surprising results or develop new algorithms. If you can't fix your car you won't get too far from civilization. If you can solve a problem with unmodified off-the-shelf technology, so can anyone else: so the work is probably not interesting or new.

# 8 Numerical Integration In Practice: ODEs in Matlab

The basics above are to show DEs aren't magic and to give you a hint that knowing more mathematics will help you in several ways, e.g. to understand your answers, to customize software tools, and to use them in ever more powerful ways.

Matlab has powerful numerical integration programs (type `>> help ode23`, for example.) They automate the sort of algorithm shown in Section 7 in the `eulmsb()` example. They give a terse, elegant interface that may seem mysterious because it's so smart. In every case of ODE solution with Matlab, you only need to do two simple things:

1. Write a function that computes and returns the derivatives of the system's *state vector* (see below). This function is often just a Matlab-ification of the differential equations defining the problem, so it is basically a translation job and should be considered easy. This function always has the same prototype (type and number of input and output arguments). You get to choose its name and write its body, but you never call it(!).

2. Write a few lines to define a span of time over which you want solutions, the initial conditions of your problem, and invoke your chosen ODE solver (e.g. `ode23()`, `ode45()`). This part should be considered very easy.

One helpful way to think about physical systems is that they have a *state* that contains all the information of interest about the system. The state *evolves* according to physical laws, control inputs, etc. E.g. on the highway we often think in terms of the position and velocity of our car and other cars, whereas their color or number of occupants isn't of interest. If $(x, x')$ is the vector of position and velocity of the car, it's the state vector for that particular abstract simpleminded way to think about the car. Position and velocity will evolve as a result of physical laws (e.g. air and tyre friction) and can be influenced by control inputs (accelerator, brake, steering). The laws and controls that make the state evolve actually dictate the *derivatives* of the state. This little bit of terminology may be helpful in thinking about our examples below.

## 8.1 Ballistics in a Vacuum

In a vacuum, a falling body accelerates due the force of gravity. On earth, the acceleration due to gravity is called $g = 9.8 meters/sec^2$

Let's plot the position (height) and velocity of a ball (in a vacuum) acted on by gravity. So the state vector of the ball will contain its height and velocity. Furthermore, we can start the ball at whatever height and velocity we want. This initial state is, by definition, the *initial conditions* (ICs) of the problem, which we must specify in order for it to have a unique solution. The state evolves according to the law of gravity, which is simply, assuming $y$ measures height, $y'' = -g$.

If we put this situation into into our favorite matrix form, we see (with $x$ a different name for the time variable in this context),

$$\frac{d}{dx}\begin{bmatrix} y \\ y' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} y \\ y' \end{bmatrix} + \begin{bmatrix} 0 \\ -9.8 \end{bmatrix} \tag{2}$$

From this, we can write down the derivatives in terms of the state vector components $(y, y')$ and the physical laws, or we can skip the whole matrix business and go right to the derivatives: either way we get

$$\begin{aligned} \frac{dy}{dx} &= y' \\ \frac{d^2y}{dx^2} &= y'' = -9.8 \end{aligned}$$

Watch closely. We're ready now to do Step 1 mentioned back in Section 8: write the *derivatives function* for our ODE solver. Given the scalar value of time and the state vector, this function computes the derivatives of the state vector (the left-hand side of our matrix equation above).

*The derivatives function always returns a column vector and must always have two arguments for time and the state vector*. The above is true even if, for instance time is not involved in the derivatives (as it is not in this ballistics problem). The output must be a *column* (not a row) vector.

So our derivative function in Matlab must look something like this:

```
function dy = ball_der(x,y)  % output vector, input scalar and vector
dy = zeros(2,1); % column vector output
dy(1) = y(2);    %dy/dx = y'
dy(2) = -9.8;  % y'' = -9.8
end
```

We've followed the rules: input parameters are a scalar and vector `x,y`, output `dy` is a column vector of the derivatives of input `y`. (It seems wasteful to create a new `dy` matrix at every call: could make it persistent or global.)

Here the "time" variable (called `x`) was not used. In fact, we'll specify a time (or $x$) range of values to the ODE solver, so that is where the time comes from. We don't use time explicitly because nothing physical (here, the gravitational constant) depends on time.

Also, we were lucky: our variable denoting height is called $y$ and our state vector is also `y`. But suppose our state vector has values with names $[x, x', y, y', z, z']$? (as it will in the ballistics assignment). If we call this vector *Y*, then we need to remember that `Y(1)` $= x$, `Y(2)` $= x'$, `Y(3)` $= y$, etc. Keeping the names straight can be confusing. Also note that we have our choice of how to arrange the state variables into a vector: Another logical choice here is $[x, y, z, x', y', z']$, in which case `Y(1)` $= x$, `Y(2)` $= y$, `Y(3)` $= z$, etc.

As mentioned in the first subsection, we don't get to call our cute little derivative function. We hand it off to the ODE solver, which calls it whenever it needs the derivatives to take the next step in the integration. To hand it off, we need a *handle* to the derivative function (see Appendix A for a little disquisition on pointers and handles in general). Some things in Matlab have handles: graphic objects and functions for example. Among other things, the handles allow us to pass a function into another function as an argument. You can create a handle for just about any pre-existing function, built-in or defined by you (but not infix functions like * or +).

For example, for the absolute value function and our derivative we'd write `@abs` and `@ball_der`. We can give handles names, as in `MyAbsHdl = @abs`. We pass the handle of our derivative function to the ODE solver.

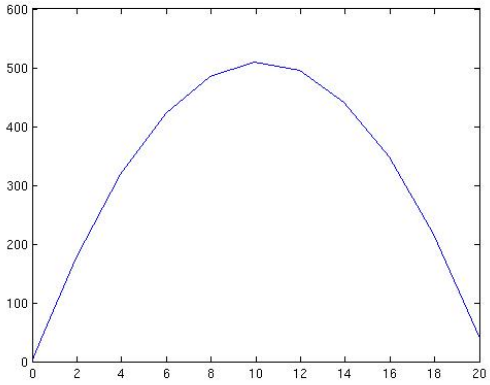
We also need a vector specification for the time interval to be covered by the integration (a couple of ways to specify this) and a vector of initial conditions. The result comes back as two arrays, `Y` and `T`: our solution is `Y(T)`. That is, `T` gives the times $t$ at which solutions are found, and `Y` gives the state vector $y(t)$ for every time in `T`.

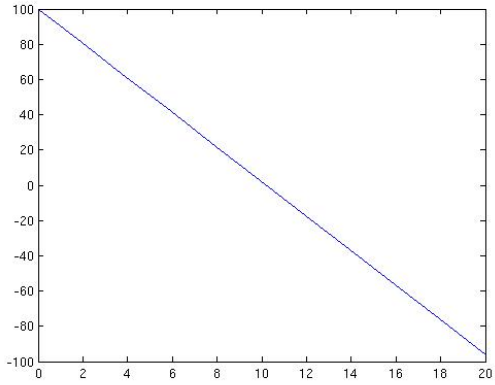
To specify the time interval or time-span, you can chose the form `[start, finish]` in which case it seems the ODE solvers choose a minimal number of steps that gives a rather coarse approximation to the answers, or you can specify an explicit set of times like `linspace(start, finish, no_steps)` which will give `no-steps` equally-spaced time steps in the range `[start finish]`. The initial conditions (values for *(y, y')*) are another vector (row or column). Unsurprisingly, the initial condition vector is the same size as the state vector, since we need an initial value for every value in the state vector.

Let's see what happens for a timespan of 20 seconds, and let's set initial conditions to start the height ($y$) at 0 and toss the ball upwards with a velocity ($y'$) of 100 (meters/sec). Here's a script:

```
ballhandle = @ball_der; %give the deriv-fn handle a name
tspan = [0 20];  % [start finish] (seconds)
y0 = [0 100];    % [ y y'] initial conditions
[T Y] = ode23(ballhandle, tspan, y0); %boom
plot(T,Y(:,1)) % T is the column vector of times from ode23
           % and the first Y column is y (height)
plot(T,Y(:,2)) % second col. of Y is y' (velocity)
```

and we see the height go up and down parabola-wise and the velocity linearly decreasing from 100 to -100.



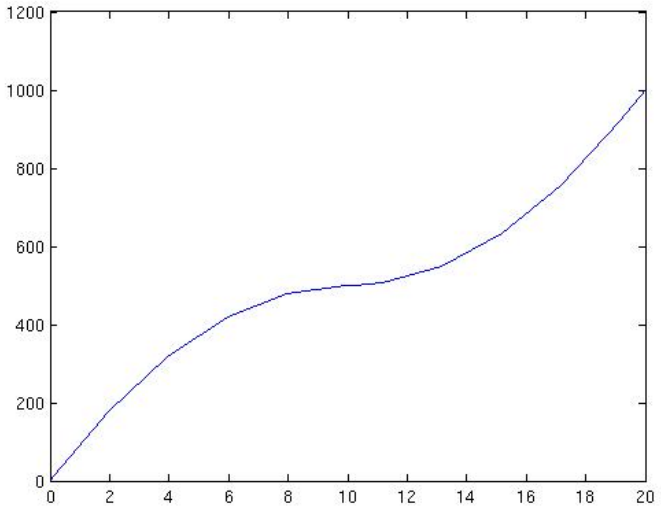

Notice how much is being done for us. We supply (but never really use) the derivative function, we start the system in the state given by initial conditions, and let it run for our time-span. We specify all these things but the ODE solver uses them, we (as programmers) do not.

Adding a Time Dependence: Simulations are great fun, since we get to play around with the rules. Suppose the force of gravity were 10, and it reversed (became anti-gravity) 10 seconds into the ball's flight? This calls for a different derivative function:

```
function dy = gball(t,y)  % t a scaler, y vector, dy col vector
if t<10
    g = -10;
else
    g = 10;
end   % gravity of 10 (not 9.8)  flips at 10 secs

dy = zeros(2,1); % col vec
dy(1) = y(2);  % y'
dy(2) = g;  % y''. now g computed above, no longer const.
end
```

Here the plot of height against time is:




## 8.2 Lorenz Attractor

Let's do the Lorenz attractor (Section 6). The equations are:

$$\begin{aligned} x' &= 10(x-w) \\ w' &= -xz + 28x - w \\ z' &= xw - 8z/3 \end{aligned}$$

These equations define a path in $(x, w, z)$ space *parameterized by time*. That is, we think of the solution tracing out a point in space as time goes by. At each instant the system's state is simply $(x, w, z)$ and the point moves (evolves) according to the law given by these equations, specifying the change in state as a function of the current state. (Sorry about using $w$ for $y$ here: by all means think of the usual Euclidean XYZ coordinates. MatLab has stolen `y, Y` in its documentation and examples, and I want to avoid that ambiguity.)

We need to write a derivative function that computes `dy = lorenz_der(t,y)` with `dy` a *column vector*. For Lorenz our state is the $(x, w, z)$ values of our point and so we need three derivatives $(x', w', z')$, and so `dy` will be a 3-by-1 vector.

For input to our derivative function `lorenz_dir()` we need a scalar "time" and a state vector called $y$. That input vector can be a row or column vector, your choice. For Lorenz, let's say `y(1) =x, y(2) = w, y(3) = z`. Thus our function looks like this, mapping directly over from the original equations.

```
function dy = lorenz_der(t, y) % col vec output, scalar, vec input
%Derivatives for Lorenz Attractor.
%t a scalar, y a vector of y's (i.e. x,w,z) of point)
%dys returned as a column vector
```

```
% y(1)... are the x,w,z
dy = zeros(3,1);
dy(1) = 10*(y(1) - y(2)); % from the equations, (q. v.)
dy(2) =  - y(1)*y(3) + 28*y(1) - y(2);
dy(3) = y(1)*y(2)-(8/3)*y(3);
end
```

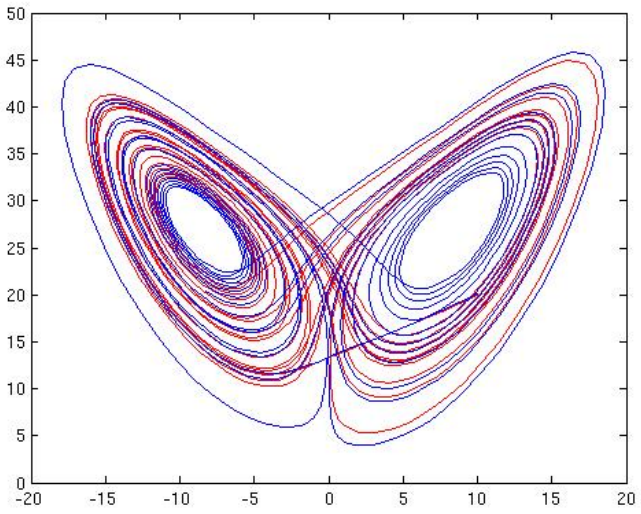
The above fits the format required – notice that there's no explicit dependence on $t$. What that means is, just as in the previous example, the next state depends on the current state and $\Delta t$, so if the ODE solver picks (or we specify) a big step there could be big changes, and we'd expect small evolution from small steps. The law is independent of any explicit time.

(In this example, we could easily write a matrix formulation as we did in the previous example, but here it doesn't seem to add any clarity since the "law" given by the Lorenz equations is so close to what our derivative function must do.)

My driver script and output look like this:

```
%  Run Lorenz
lorenzhandle = @lorenz_der;
y0 = [10, -10, 20];  %  initial conditions for (x,w,z)
timespan = linspace(0, 20, 1500); %specify 1500 instants
[T,Y] = ode23 (lorenzhandle,timespan, y0); % *boom*
plot(Y(:,1),Y(:,3), 'r');
hold on
y0 = [10, -10, 20.1];  %  initial conditions close to previous
[T,Y] = ode23 (lorenzhandle,timespan, y0);
plot(Y(:,1),Y(:,3), 'b');  %BUT big difference in the two solns
```

and we see



Here, plotting x,w, or z against time is pretty boring `plot(T, Y(:,1))` for instance ....each coordinate just wiggles up and down through time somehow: try it. The pretty "butterfly plots" are from plotting the variables against each other (here, x against z). That's the same as looking at the path of the point in 3-D from one of the XYZ directions (here, Y). The red and blue plots arise from two different initial conditionss for z: 20 and 20.1, respectively.

There is an option for the ODE solvers to return a structure that has the same `T,Y` information in it, but can also be used with the function `deval` to evaluate the solution (and its derivative) at *any* point, not just those created in the original solution.

## 8.3 Real-World Example

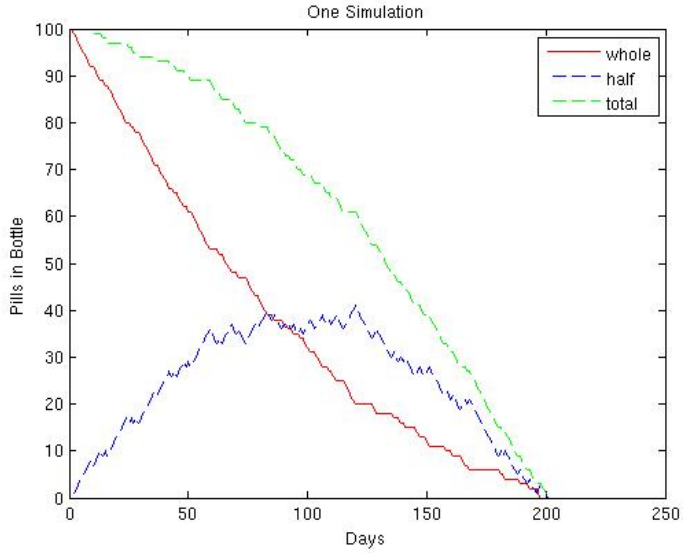
A professor is prescribed half a Lisinopril tablet a day. Every day he randomly shakes out a pill from the bottle. If it's a whole tablet he bites off half and spits the other half back. If it's half a tablet he swallows it. How does the population of total pills, half-tablets, and whole-tablets vary through time?

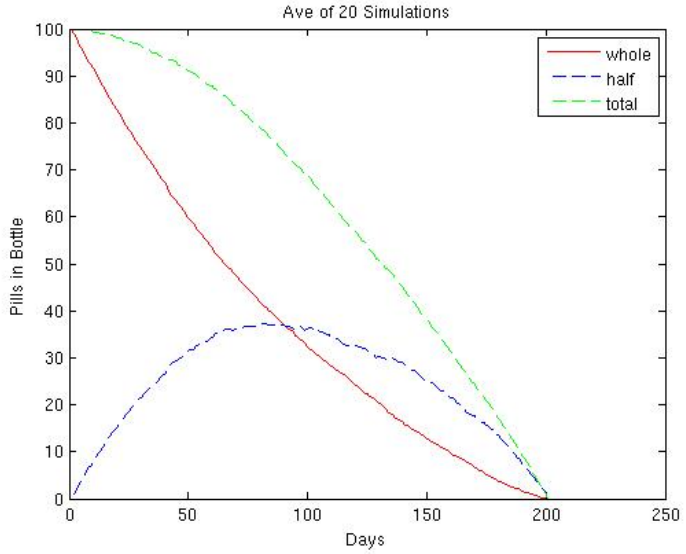
We expect the number of whole tablets to fall, maybe something like exponentially, but meanwhile the number of half-tablets should rise and then fall. The total number of pills starting at N should go to zero exactly on day 2N.

One could just simulate the process with a little code (which also reveals the recurrence equations at work). That is,

```
for i = 1:N
  PW = W(i)/(W(i)+H(i));
  if rand < PW
      W(i+1)=W(i)-1;
      H(i+1) = H(i)+1;
  else
      W(i+1) = W(i);
      H(i+1) = H(i) -1;
  end
end
```

Here are results for a single simulation and the average of 20 simulations.




The pill-consumption and generation process is something like a chemical equilibrium reaction, with the amount of reactants and rate of reactions depending continuously on things like concentrations of reactants. If we think not about actual pill numbers but *expected* pill numbers the problem is really easy.

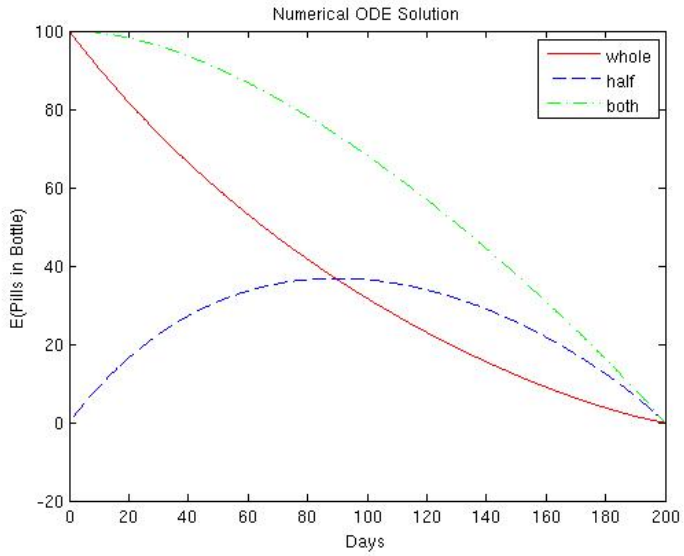
It's obvious that the probability of picking a whole tablet is proportional to the (expected) concentration of whole tablets. With $W$, $H$, and $T$ being the expected numbers of whole, half, and total pills, the probability of getting a whole tablet is $p(Whole) = W/T = W/(W + H)$. To shake out a whole tablet is to lose it, due to the biting process. Thus at every step, $dW = -p(Whole)$ is the change of the expected number of whole pills every day. But the biting process yields a half tablet that is added to the half-tablet population, so we also have an equal effect of the opposite sign there: $dH = p(Whole)$.

The tablet shaken out is half or whole, so the probability of a half-tablet is $(1-p(Whole))$. When that happens, the half-tablet is consumed: $dH = -(1 - p(Whole))$ Adding the two half-tablet effects says that every day $dH = 2p(Whole) - 1$.

That's our system: formulas for $dW$ and $dH$ that depend on $p(Whole)$, a function of $W$ and $H$. The derivative function looks (with `y(1) = W, y(2) = H`) like

```
function dy = liso_der(t,y)
dy = zeros(2,1);
probW = y(1)/(y(1)+y(2));   % p(Whole) = W/(W+H)
dy(1) = -probW;             % dW = -p(Whole)
dy(2) = 2*probW -1;         % dH = 2p(Whole)-1
end
```

A simple script or just typing into the top level can set the initial conditions (say `W = y(1) = 100, H = y(2) = 0`) and a time range from 0 to 200 (days). Calling good old `ode23()` yields




## 8.4 Made-up Example

This example is a little like the Ballistics assignment in some ways, but it's really just made up to look a little more complicated than previous two examples.

Suppose we have these two equations for the little-known two-dimensional 'Phoobar' process:

$$\begin{aligned} y'' + [g(x)y' + r(x)y]z &= v(x) \\ z' + s(x)y'z &= u(x) \end{aligned}$$

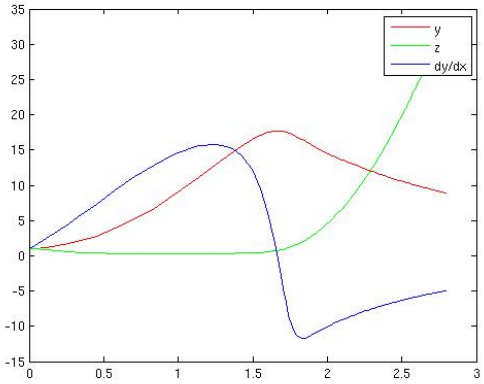
First, remember that DEs are traditionally functions of time ($t$) or some other independent variable symbol, often $x$, as here. That means in our derivative function, the $x$ in the

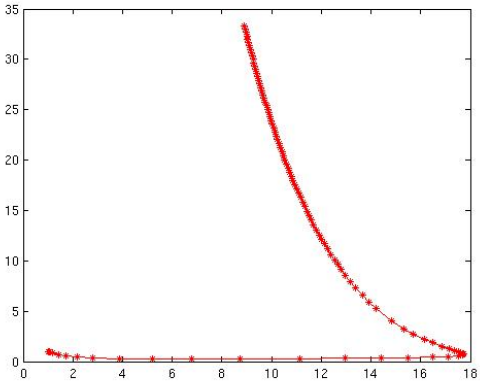
equations above is the `t` that Matab expects in its `(t,y)` arguments of the derivative function. Also, our state vector involves the quantities $(y, y', z)$ (in some order), with both $y$ and $z$ being functions of $x$.

Thus we could be daring and call the input state vector `state` instead of `y`, and make the correspondences `state(1) = y; state(2) = z; state(3) = y';` to compute the derivative `dstate` vector:

```
function dstate = phoobar_der(t,state) % output col vec, input scalar, vec
dstate = zeros(3,1); % output must be col vec
dstate(1) = state(3); % y'
dstate(2) = u(t) - s(t)*state(3)*state(2); % z'
dstate(3) = v(t) - (g(t)*state(3) +r(t)*state(1))*state(2); % y''
end
```

Here we need to use one `dstate` component ($z'$) to define another ($y''$), so it has to be computed first. We need lots of functions defined in M-files (or as subfunctions might be nicer): `u.m, s.m, v.m, g.m, r.m`. With a random bunch of definitions $u(t) = 2t, s(t) = 1/(t+1), v(t) = 13, r(t) = t\log(t+2), g(t) = t^2$ and with the timespan and ICs given in the call `[T,Y] = ode23(@phoobar_der, [0,10],[1,1,1]);`
we see how the 'Phoobar' state vector components evolve (left) and a "phase diagram" plotting $y$ versus $z$ shows its path through the $y, z$ phase space (right):



# 9 Afterword and Acknowledgment

With this super-introductory paper, what you can find on-line and in books, and the copious Matlab documentation and examples available from Mathworks, you've got all you need.

Thanks to Roger Gans for significant technical corrections and editorial improvements.

# 10 Sources and References

Euler, L. *Institutiones calculi integralis*, 1768-1770. (1909 Edition available free online)

Cauchy, A.L *OEuvres completes d'Augustin Cauchy*, publiees sous la direction scientifique de l'Academie des sciences et sous les auspices de m. le ministre de l'instruction publique, 1882.

Volterra, V. "Variazioni e fluttuazioni del numero d'individui in specie animali conviventi", *Memorie della R. Accademia Nazionale dei Lincei,* anno CCCCXXIII, II (1926) pp. 1-110.

Lorenz, E. N. (1963). "Deterministic nonperiodic flow", *J. Atmos. Sci. 20:* 1963, pp. 130141.

Wikipedia *Runga-Kutta Methods, List of Runga-Kutta Methods, The Day After Tomorrow, Lorenz Attractor, Butterfly Effect, ODE solution, etc etc.*.

Matlab Documentation: Mathworks, Inc. For instance, this page has a nice rundown of the mathematics and several examples:
`http://www.mathworks.com/support/tech-notes/1500/1510.html`¡br¿ Calls and Characteristics of ODE solvers can be found from command level by something like `>> help ode23`.

This van der Pol page is nice:
`http://www.mathworks.de/access/helpdesk/help/techdoc/math/f1-662913.html#brfharp-7`

*The Day After Tomorrow*: Film by Richard Emmerich. 2004, The perils of a chaotic system where changes can occur dramatically quickly.

*Numerical Recipes in C*: Press, W.H.; Flannery, B.P.; Teukolsky, S.A.; Vetterlng, W.T. Cambridge U. Press, 1988. Also available online I think.

*Ordinary Differential Equations (4th Ed.)*: Birkhoff, G. and Rota, G-C. Wiley, 1989.

*Differential Equations, a Dynamical Systems Approach Part I*: Hubbard, J.H and West, B.H. Springer-Verlag, 1990.

*Matlab*: S. Attaway, Elsvier, 2009.

# A Variables, Pointers, and Handles

We know how variables act in Matlab: they are the names of particular "things" in the program, like complex numbers, strings, integers, arrays... Most programming languages have similar sorts of variables.

A variable is implemented as a little data structure containing (at least) the name associated with its value, and also an address for that value in the computer: where the value's bit representation lives.

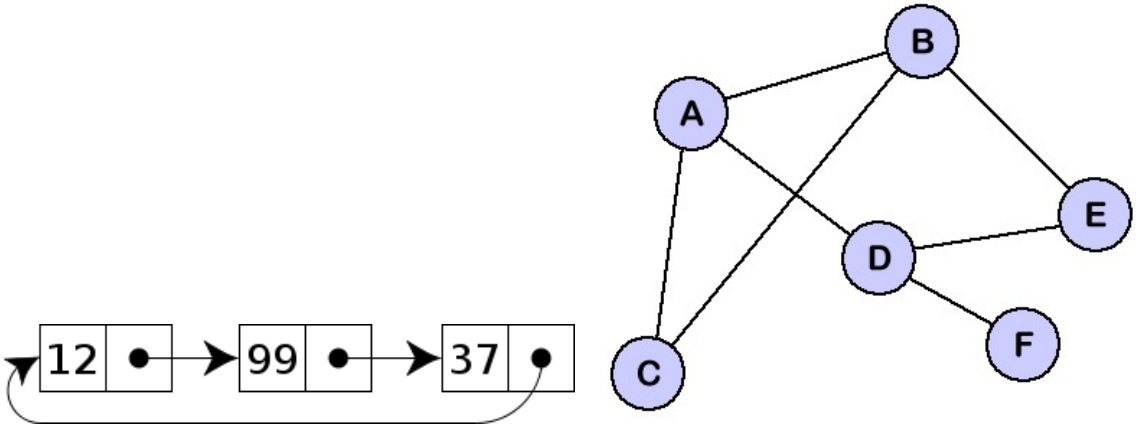
Now addresses are useful in themselves, and in most languages are a datatype of their own, often called a pointer or a handle. In the language C, the address of a variable is obtained with the & operator, so the address of `count`, is `&count`. We can treat this address like any other value (it's really an integer): `count_ptr = &count`.

To *follow a pointer* or *de-reference a pointer*, is to evaluate it and thus retrieve the thing it's pointing at (whose address it is). De-referencing is done with *, so we if we say

```
...
count = 10;
count_ptr = &count;
x1 = *count_ptr;
*count_ptr = 25;
x2 = *count_ptr;
printf("%d %d %d", x1, x2, count);
```

We'll get output 10 25 25. There are lots of consequences: if there is more than one pointer to a variable and one of them is used to change its value, following any of them finds the changed value. If the last pointer to a thing is moved to point away from it, the thing cannot be accessed and it may be *garbage collected*.

Pointers are often used to construct lists, trees, and graphs, which are among the *data structures* studied in the eponymous CS courses.




Usually *structures*, which Matlab has (called 'structure arrays'), are accessed by pointers to them. They have named fields, which contain values of arbitrary types, including pointers to other structures. One can make an arbitrarily connected set of structs, or related pieces of data of possibly different types. A typical struct or structure array might look like

```
Employee Struct
  Name-field:  ``Fred Bloggs''
  Addr-field: <pointer to an Address Struct>
  Salary-field: 102,345.02
  Suspect-Friends-field: <pointer to linked list of Employee Structs>
```

In practice, in CSC160 you might want to use Matlab structure arrays, but we won't address them in class.

In some programming languages, functions are "things" just like real numbers, and can be created, passed into functions, returned from functions, etc. For instance, in the Scheme language, the function `lambda` creates an anonymous function. Let's use it to create a nameless function that itself takes a function as a first argument (say + or * or `mean`) and applies it to values of the next two arguments:

`(lambda (f x y) (f x y))`.
The call to `lambda` creates an anonymous function: evaluating this statement yields a function with no name as result. The first `(f x y)` is simply the list of arguments and the second `(f x y)` applies function f (the first argument) to the values of the next two formal parameters x and y. The whole statement evaluates to a function object that can be given a name or applied directly to an argument list, as in `((lambda (f x y) (f x y)) (- 5 3))`, which evaluates to 2.

Back to Matlab. Matlab has anonymous functions (Attaway 9.1) and function pointers, called function handles (Attaway 9.1, 9.2). Some examples:

```
abs(-4) % call abs function, returns 4
@abs    % a fn. handle.  returns @abs
my_abs = @abs % now I have name for handle
my_abs(-5)    % returns 5 (? maybe shouldn't work?)
@abs(-5)      % syntax (? semantics surely? ) error
[T,Y] = ode23(@abs,timespan,y0) % OK
[T,Y] = ode23(my_abs,timespan,y0) % OK, same thing
% this next implements the lambda expression above
function apply (funh,x,y)
funh(x,y)
end
%so
apply(@max,5,4)   % returns 5
% but....(!)
@+                %  syntax error (??-- OK in Lisp)
```

The the question marks show points of possible confusion for anybody with an ounce of sense. They clearly have technical answers someone like Michael Scott could answer, which stem from the implementation. Don't expect to transfer either the good (flexibility) or bad (unpredictability) behavior of Matlab to real, compiled computer languages. For example here is a simple function declaration and call in C for a function `apply_binop`:

```
%declaration
int apply_binop(int (*fn_ptr) (int, int), int a, int )
{return (*fn_ptr)(a, b));    // call using function pointer }
%call
 apply_binop(&max, 2, 5);
```

Note the inflexible and explicit declaration of `apply_binop` (it returns an int and takes exactly three arguments). The types of all the arguments are similarly fixed here, including `fn_ptr`, which the declaration says is a function pointer: *literally: if de-referenced (*) it yields a function* that expects two integer arguments and produces one integer result. Sending in a function like sqrt would be a bug. Note then that that pointer *is* de-referenced when it is used, yielding the function to apply to 2 and 5. The call sends in the pointer to max (yielded by the &) as it should. Now *that's* unambiguous and "Verry Korrekt".

But not neat. In fact, the klumsiness has led the C standards committee to allow the (confusing ONLY if you think about it, Matlab-like) syntax

```
return fn_ptr(a, b);    // OK call (mis)using function pointer }
```

Moral: handles and pointers can be *very* useful in many programming contexts. You'll see the concept in other languages. We can't avoid a very simple and obvious use of function handles when using ODE solvers in Matlab.