

Oktoberfest!

5C.1.1 ODE: Outrageous Destinations in Europe (Optional)

ODE is short for *Outrageous Destinations in Europe*. Okay, that's not true. It is an initialization for *Ordinary Differential Equations*, but if I put that in the title, along with the word "Optional", you might not have opened this module.

Although the material is optional, I'd like to give you a short pitch on why you might want to spend a half hour reading this and the next few sections.

First, ODEs are among the tools of the trade in social science, engineering, medical epidemiology -- you name it. They are used by people who do the research, make the discoveries and publish the results that the rest of the world reads about in articles with titles like "*New Study Shows Foothill Programmers Live Longer, Make More Money and are Happier.*"

Secondly, ODEs are not that hard. Yes, you have to first have a course in Calculus to study them, and yes, they really do require a semester to master, but you can get the general idea, right here, right now, with no obligation in six short -- and don't forget *optional* -- module sections. This is one of those times you can read about something fascinating in class, stress-free, no worries.

Finally, if you are one of those students who has already had a couple years of calculus and maybe a beginning physics course, you might take the material here and run with it immediately. You can do one of the intermediate options in an up-coming assignment and feel the power of numerical solutions to ODEs.

Because this is optional, I will be speaking to students who have had a year of calculus. I will assume you (they) know what functions, derivatives, integrals and infinite series are. Even if these terms are foreign to you, try reading this anyway, just to see what the conversation is like. We will be talking about *climate change*, *astrophysics*, *epidemics* (like H1N1) and *chaos*, and we will see how a computer program solves ODEs in these fields *numerically*.

5C.1.2 A Visual Interpretation of ODEs.

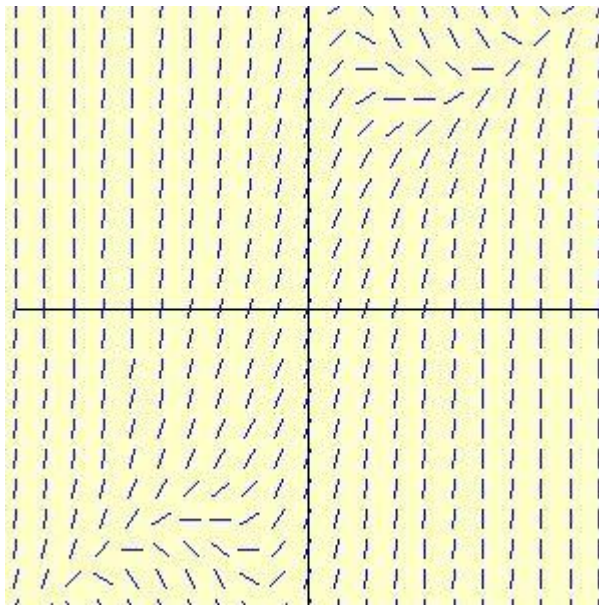
As you know (remember I'm talking to calculus people, but you are welcome to eavesdrop if you are not in this group) the *derivative of a function at a point* is often interpreted as the *slope of the*



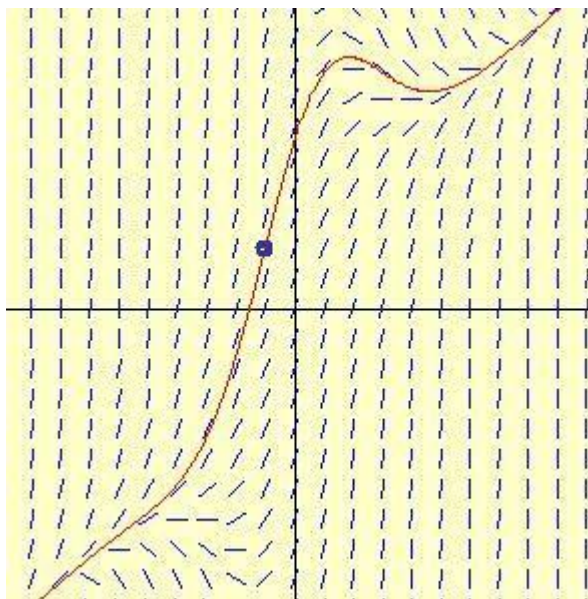
tangent line to the graph of the function at that point. In your first few weeks in beginning calculus, you started out with a function like $f(x) = x^3 - x + 17$ and computed its *derivative* (which is, in this example, $f'(x) = 3x^2 - 1$). So you start with one function, $f(x)$, and the *derivative* gives you a new function, dy/dx or $f'(x)$, that represents the *rate of change* of the original function. Geometrically, however it also tells us about the *slopes of the tangent lines* of the graph of the original function.

Ordinary Differential Equations represent a technique for *reversing* the process. You start out with the *slopes* which, collectively, represent the differential equation, and you want to come up with the original function.

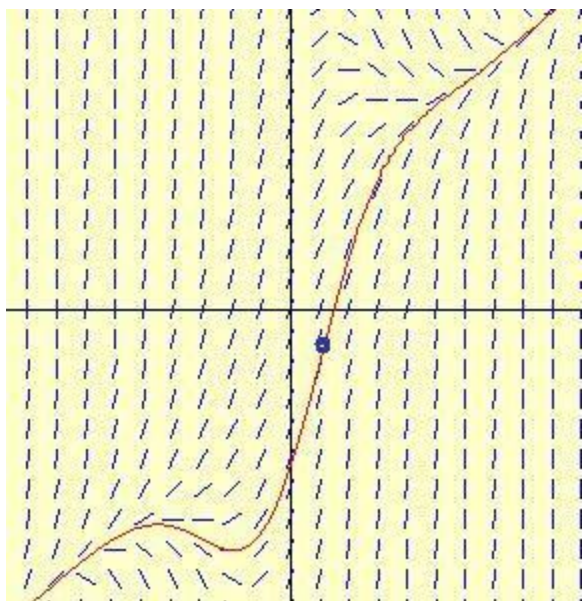
Let's say we have a picture of the x-y plane with a bunch of slopes (this is a pictorial representation of some ODE): short straight lines that indicate what the slope of the unknown function should be at each point in the plane - no functions yet, just a bunch of slopes. This is called a *slope field*, or if you want to impress your kids, a ***vector field***:



Now, if we start at any point in this ***vector field***, we can probably sketch, by hand, an approximate curve whose tangent slopes agree with these short segments. Here is one example, where I picked the point $(-1, 3)$ and drew up and down, going with the slopes.



If you imagine the **vector field** as wind directions (we'd need to add arrows for that, but, okay) then the solution curve represents the path that a dead flea would take if you released it at the point $(-1, 3)$. One direction from the initial point represents where the flea *will* go, the other direction where the flea *has been*. Of course, I could have placed the dead flea somewhere else, and this would result in a different path. Below, I dropped the fallen warrior at about location $(1.2, -1.2)$:



The two curves that I sketched above are the *original functions* that we wanted to find. We had the slopes, but were looking for -- and seemed to have found -- two functions that satisfy the ODE. Because the actual function depends on both the **vector field** (the slope diagram) as well as the **initial condition** (where we drop the flea), we see that there are many solutions to the same differential equation. If you also specify an **initial condition** -- a first location for the flea -

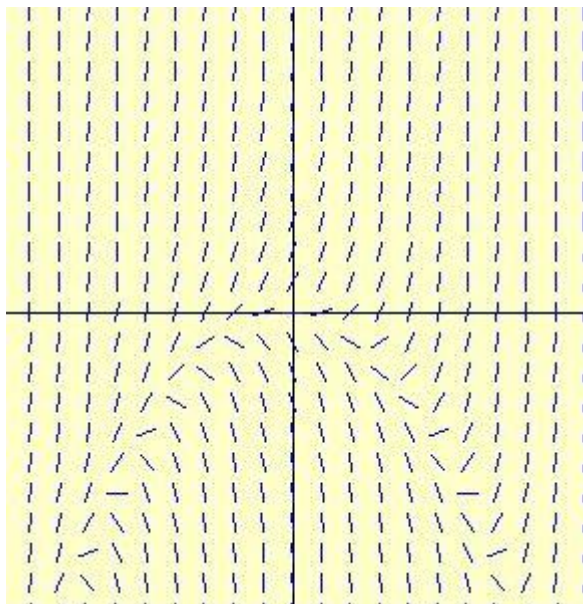
- that usually pins it down to one unique solution curve. Consequently, we often refer to an ODE with initial conditions as an *initial value problem*.

Before we complete this section, let's associate a real ODE with a geometric *vector field*. Here's the ODE (without any initial condition):

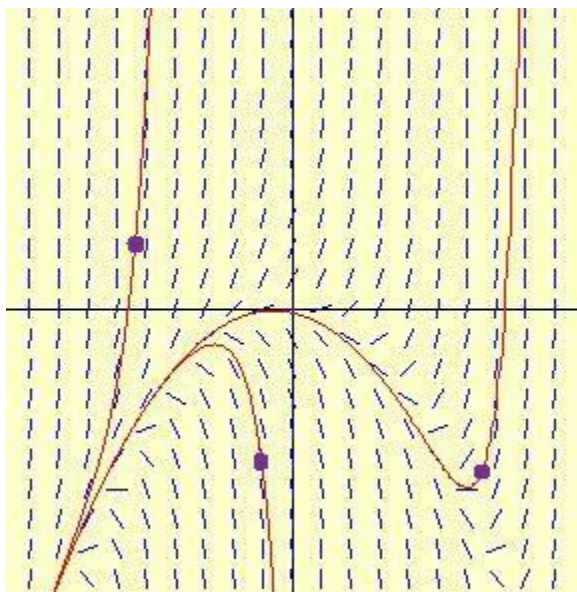
$$f'(x) = x^2 + 3y$$

or, using the common alternate notation where y is the dependent variable that represents $f(x)$:

$$\frac{dy}{dx} = x^2 + 3y$$



And here are three different *solutions* that arise from three different *initial conditions* (the dot on each curve represents the initial condition that spawned the solution):



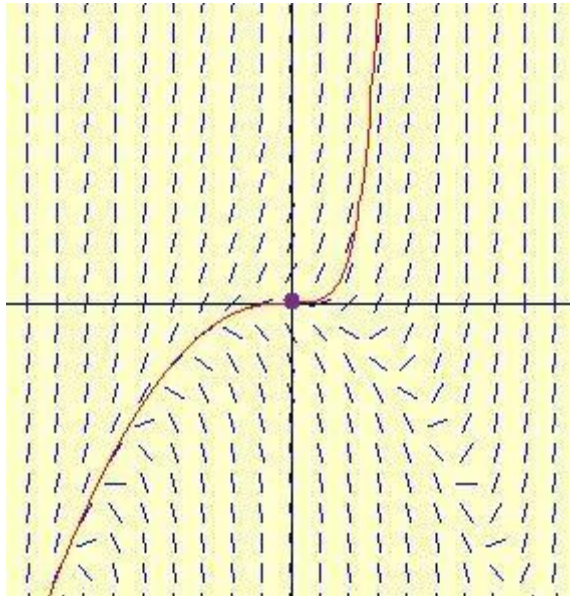
5C.1.3 Analytical Solutions

Why do we need computers, anyway? Isn't the *integral* the opposite of the *derivative*? Sometimes. *Integrals* usually apply when the slopes are only functions of the independent variable, x , but as you see from above, the derivative, y' , is a function of x and y . Also, even if this were just a simple integration problem, sometimes we can't compute the *integral* analytically. An *analytical solution* is one that can be computed precisely using pencil and paper. Most interesting real world problems in science and engineering don't have analytical solutions, and that's why we programmers get to have fun.

Actually, in the above ODE, $y' = x^2 + 3y$, one *can* find an analytical solution. We would learn, for example, in the first week of our first *differential equations course* how to solve it. If we started with that ODE and the initial condition $\mathbf{0} = f(\mathbf{0})$ (which says we want the curve to go through the (x,y) point $(\mathbf{0},\mathbf{0})$), we would, in a few minutes of pencil pushing, calculate the exact solution. It is:

$$y = \frac{1}{27} (2e^{3x} - 9x^2 - 6x - 2)$$

and here's the graph:



This next paragraph is only for real math types - you can skip it if you are just an observer:

With these things, it helps to look at the graph and see if it matches the slope field (it does) and also see if the graph matches the *solution formula*. Well, when x gets very negative, e^x becomes unimportant and the *quadratic* part of the formula dominates, so we expect an upside-down parabola and, indeed, that's what we see. On the other hand, when x becomes very positive, e^x dominates so we expect an exponential curve, as we also observe.

Time to get a little more precise. We will drift a bit from the visual interpretation, although it still applies, and focus now on the precise definition of an ODE. Using the x - y notation (rather than the $x, f(x)$ notation), we can describe an ODE using the following algebraic symbolism:

$$y' = G(x, y)$$

This assumes that we can get y' alone on the left hand side, which we will assume we can do, and this is the case in all of our up-coming examples. Don't confuse the function $G()$ with our solutions $y = f(x)$ of the ODE. $G()$ is the ODE, that is, it is the description of the relationship between the derivative y' and the x and y variables. So, in our previous example, the ODE was

$$y' = x^2 + 3y$$

If you are keeping score, this would make our function $G()$:

$$G(x, y) = x^2 + 3y$$

This differential equation has the following properties:

1. It is **linear** in the **dependent** -- or function -- variable y . (We don't care too much about the **independent** variable, x). This is called a **linear differential equation**.
2. It is **first order**: there are no higher derivatives beyond y' in the equation (no y'' , y''' , or $y^{(6)}$ for example. (The ⁽⁶⁾ in the exponent here does not mean power because it is in parentheses; it means the **6th derivative**, i.e. $y^{(6)}$.)

Therefore, it is a **linear first order ODE**.

Linear first order ODEs are usually pretty easy to solve by hand, and even if you can't, you can at least get the solution in the form of an integral which can be computed numerically by computer. This does not require any new differential equation theory -- it is just an application of basic calculus.

Let's look at a **non-linear ODE**:

$$y' = \frac{y^2 - x}{y^2 + 1}$$

Now the dependent variable, y , appears with an exponent, **2** so we are no longer linear. In fact, the following ODE, without an exponent on y , is also **non-linear**:

$$y' = \frac{y - x}{y + 1}$$

That's because y appears in the denominator. These are two **non-linear first order ODEs** and are not so easy to solve. So if the function $G(x,y)$ which defines the ODE is **non-linear** in the y variable, we are in non-linear "land", and this land requires carrying heavy artillery. That artillery consists of a little knowledge of a field called **numerical analysis**, a computer, and a programmer (that's *you*) to fire the shots.

5C.1.4 Systems of Non-Linear ODEs

We are going to learn to solve the above **non-linear first order ODEs** numerically later in this module. But we want to stop here to take some time to see how such ODEs come up naturally in science. We'll study specific problems in two areas: climate change and the interior structure of stars like the sun. Then we'll come back and do some programming for an idealized epidemic outbreak like H1N1.

One thing you will see in all these examples is that, unlike the single non-linear ODE:

$$y' = G(x, y)$$

we will have more than one unknown function. In climate change we have functions like **temperature**(t) or **wind_speed**(t) which are both functions of time, t , and which are related to one another. In order to know what is going to happen in the future, we need to solve for both of these functions simultaneously - you can't find one without the other. Inside a star, we have functions like **pressure**(r), **mass**(r), or **luminosity**(r), which are functions of the radius, r , at which we are measuring the pressure, mass or luminosity, respectively. Again, they have to be solved simultaneously. In such cases we need just as many ODEs as we have unknown functions. Let's say $X(t)$, $Y(t)$ and $Z(t)$ are three unknown functions of time, t . Let's also imagine that they are found to satisfy the following **system** of ODEs:

$$X' = X + Y - t$$

$$Y' = YZ$$

$$Z' = \frac{X}{Y - 15}$$

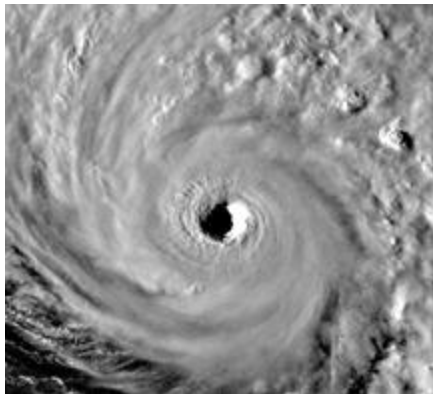
These three equations consist of a **system** of ODEs which are related - note how the Y appears in the ODE for X and the X and Y both appear in the ODE for Z . Even more importantly, the expressions on the right side of these equations involve products of the unknown functions, YZ and $X/(Y-15)$ (a *division* is nothing but a *product*). This makes our system **non-linear**. This means we will need to use **numerical analysis** and a computer to solve it.

Don't let the concept of a **system** of ODEs (many functions) throw you. Going from one ODE to a **system** is actually quite simple - you will learn to do this in your course in differential equations, and you can probably even figure it out if you understand the theory of a single ODE. So if you learn to solve a **single non-linear ODE**, you will be able to extend this to a **system of non-linear** ODEs.



Climate Change

5C.2.1 Modeling Climate Change



There is a statement which attempts to characterize the complexity and chaotic behavior of weather forecasting that goes something like this:

A butterfly flapping its wings in Beijing could cause a hurricane off the Florida coast.

The statement is meant to express a phenomenon known as "**chaos**", a mathematical theory developed in the past half century. (Related terms are *bifurcation theory* and *catastrophe theory*). Chaos applies not only to weather, but to many complex systems. Recent economic turmoil has produced interest in how **chaos** might describe financial markets better than classical, statistical models (do a search on *Taleb and Mandelbrot* about this subject, for example).

The above mentioned *butterfly* actually started life as a *seagull* in 1963 in a paper by Edward N. Lorenz who made the following comment:

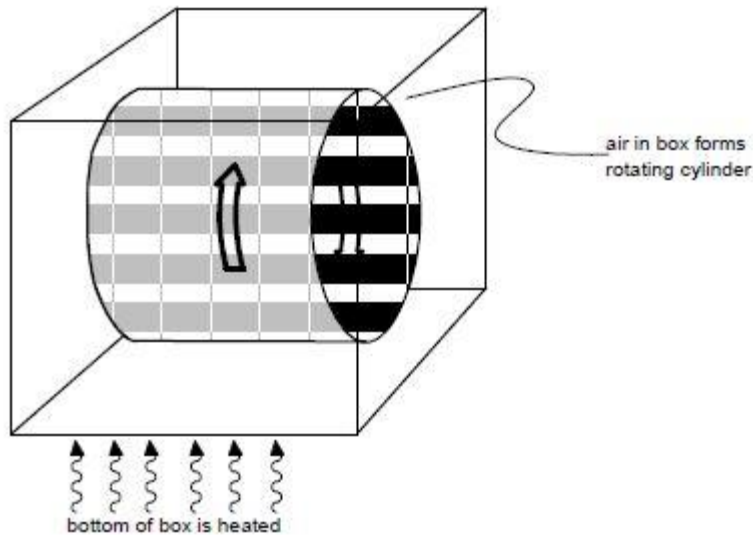
One meteorologist remarked that if the theory were correct, one flap of the sea gull's wings would be enough to alter the course of the weather forever.

The theory to which he was referring was a **climate model** which Lorenz, himself (pictured below), introduced earlier that year, and this is the topic of our discussion.

The study of climate change is a highly complex and mathematical field. Understanding the studies, and more importantly, contributing to new ones, is something that some of my students have indicated to be a goal of theirs. What we'll do here is to catch a glimpse into one approach to modeling climate change -- an approach that must be studied by anyone who goes into the field of climate modeling. It is called the **Lorenz Model**. We are interested because it results in a system of non-linear ODEs which require a numerical (computer), solution that we develop in the following sections.



When modeling any system, we start by making what appears to be absurdly simple assumptions. Lorenz, aware of the well known greenhouse effect, wanted to study how the difference in temperatures and wind speeds affected climate, long term. If one made an extremely small change in initial conditions, how much effect would this have over time. Furthermore, in the true spirit of mathematical modeling, he started with as simple a model as he could imagine. He ignored all aspects other than that of convection of air currents (the rising and falling of warm and cool air in a continuous cycle). Furthermore, he restricted his study to a closed system by placing the air in a large shoe-box shaped cell called a “Rayleigh-Benard” cell.



He picked initial conditions for the air speed and temperatures at the bottom and top of the box, and let the air circulate using the laws of physics. The model viewed the air as a rotating cylinder and consisted of understanding three aspects of the system, all functions of time, t :

1. $x(t)$ - the rate of rotation of the cylinder of air at any given time, t (loosely, the wind-speed)
2. $y(t)$ - the difference in air temperature between the two opposite sides at any given time, t (loosely, the difference between the rising warm air and falling cold air)
3. $z(t)$ - The top-to-bottom difference in air temperatures as a deviation from a straight line at any given time, t (this is hard to describe without some advanced math, but it is the function that relates to warming or cooling trends)

These three functions are described by a simplified *ordinary differential equation*, or ODE. I cannot derive this particular **ODE** - this is a matter for an upper-division course in physics, math or meteorology. What matters here is that when attacking this simplified model for climate, we end up with an ODE:

$$\begin{aligned}\frac{dx}{dt} &= \delta(y - x) \\ \frac{dy}{dt} &= rx - y - xz \\ \frac{dz}{dt} &= xy - bz\end{aligned}$$

There are so many things to say here, I can barely contain myself.

- First, the fact that this model seems unrealistically simple is no reason to dismiss it. The most complex and successful models of nature are built upon simple models. If you don't understand the simple ones, you can't do the harder ones. Furthermore, as is often the case, if you *can* do the simple model, the techniques for the more complex one are not that much harder - in some sense, 90% of the battle is understanding the simple model.
- What do we expect from these equations? We hope to solve for $x(t)$, $y(t)$ and $z(t)$ to give us changing values for temperature differences and wind speeds over time.
- And what does this have to do with *Chaos* and the *Butterfly Effect*? If we assume one starting value for x , $x(0)$, (the initial wind speed) and change it by one part in, say, 100 million, then what we find is that we get a completely different future. This is where the flapping-wing metaphor got started. And it was not something you could smooth over, either. No matter how close two initial wind velocities got, holding everything else constant, the solutions to these equations always become wildly divergent. This is the hallmark of *chaos*. Things are not continuous even at the most accurate level of granularity.
- While all the excitement in this case was about the initial wind speed, $x(t)$, being chaotic, in the context of the 21st century, the focus has been on the *temperature difference* between the upper and lower atmosphere, tracked here by the function $z(t)$. One can study whether tiny changes in $z(0)$ lead to chaotic, unpredictable behavior in the future.
- What are the parameters δ , r and b ? They describe properties of the box that are constant throughout the simulation (viscosity ratios, box boundary temperatures and box dimensions) and are not important to our present discussion.
- This differential equation is our old friend the *first order, non-linear ODE*. Recall, *first order* means only x' (dx/dt) appears, not x'' or x''' . *Non-linear* means that the unknown functions, x , y and z , are messily combined by multiplication due to the xy and xz terms. (The other products are not between two functions, so they don't contribute to the non-linearity).

And now come the items that bring us back to the subject of this course:

- These equations cannot be solved *analytically* (by computing an exact solution using pencil, paper and a Ph.D. from Cal Tech). They must be solved using a computer program written, preferably, at Foothill College, and the technique for solving them is the method we will learn later in this lesson.
- Sometimes you can't use pre-packaged applications to solve equations of interest. It is often the case that the lead researcher asks his or her Ph.D. students to write the programs from scratch because they want certain customizations that cannot be handled by the off-the-shelf programs.

We won't really solve the Lorenz equations, by the way. We will solve the problem for a single non-linear first order ODE. The main point is that we have a non-linear ODE which requires a numerical solution, and you have all the knowledge to solve it. (By "solve it" I mean get an approximation that is as accurate as your CPU and personal schedule can withstand.)

Astrophysics

5C.3.1 Modeling Solar Evolution

A great deal of research has gone into producing mathematical models of the sun. By *models*, we mean equations that tell us how the sun will evolve from its birth to its death, and what its interior conditions are at any point in its interior. For example, how is the mass of the sun distributed when it's young? When it gets old? How does temperature change as we move from the surface to the center?

The way we test a solar model is to start the model at day 1 -- the birth -- and let it simulate the changes for 4.5 billion years (the sun's current age). If the conditions that the model predicts match our current observations, then we have reason to believe that our model might be a good one.



The creation of solar models is a big industry within astrophysics for at least three reasons:

1. If we can get a model to accurately predict the behavior of the sun, we know what it will do in the future.
2. The model we use for the sun helps us create models for other stars leading to an understanding of the evolution of the billions upon billions of stars in our Milky Way and other galaxies.
3. The solar models are based on certain assumptions that we must guess. Examples are (1) the initial amount of helium (*helium abundance*) in the sun, (2) the length (or height) that a convective blob of stuff rises buoyantly within the sun before it combines with the everything around it (*mixing-length parameter*), and (3) the initial abundance of heavy elements. If we can guess values for these parameters that result in an accurate model, then we have evidence that the guesses might be correct.

As you might expect, the creation and justification of a good solar model takes a lot of advanced physics: thermodynamics, quantum mechanics, special and general relativity, and of course a little math. What I will do here is outline the basic equations of an over-simplified solar model, just so we can see how they lead to a system of ***non-linear ODEs***, exactly the kind of equations that we are learning to solve in our programs.

5C.3.2 The Quantities of Interest in the Sun

We are interested in knowing about how the following quantities change (1) over time and (2) from location-to-location within the sun. However, to make the model a bit more manageable, let's forget about the time variable, and assume that the sun is in a *steady-state*. If we can get a feel for how that problem is set up, we'll be satisfied that we can extend it to the more complex problem. Implicit in my discussion will be several other simplifying assumptions that I won't bother to state, except for one: we assume radial symmetry, that is, the direction from the center of the sun does not matter - for anything. If we are 129 km from the center in one direction, conditions are the exactly the same as they are 129 km from the center in some other direction.

The independent variable we will use here is r , the **radius**, and we consider the quantities of interest as functions of r . For example, **temperature** varies as a function of **radius** inside the sun (the center is hotter than the surface), and this function for **temperature** will be called $T(r)$. (To reduce the notation below, I will write this as T_r which shows T 's dependence on r more concisely than does the notation $T(r)$).

Here are the various functions, or quantities, that we will put in our model:

1. M_r , the **mass** of only that portion of the sun from the center out to the radius r (which you should think of as *less than* the full radius R of the sun). r could be anything from 0 to R , so M_r will vary from 0 to M_R , the total mass of the sun, as r goes from 0 to R .
2. T_r , the **temperature** at radius r inside the sun
3. P_r , the hydrostatic (think gas) **pressure** at radius r
4. L_r , the **luminosity**, or energy per unit area per second radiated at radius r inside the sun due to the central nuclear reactions.
5. ρ_r , the matter **density** at radius r inside the sun.

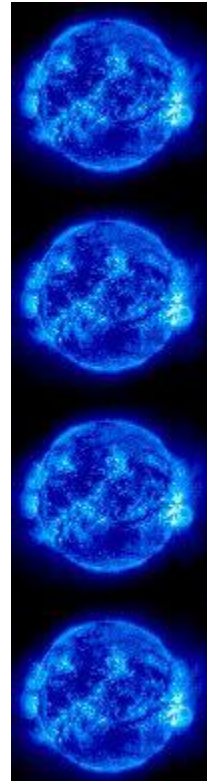
If we can determine these five functions, then we will be content that we know basically what's going on in the sun at any point from the center to the surface.

5C.3.3 The Differential Equations

1st Solar Structure Equation: Conservation of Mass

$$\frac{dM_r}{dr} = 4\pi r^2 \rho$$

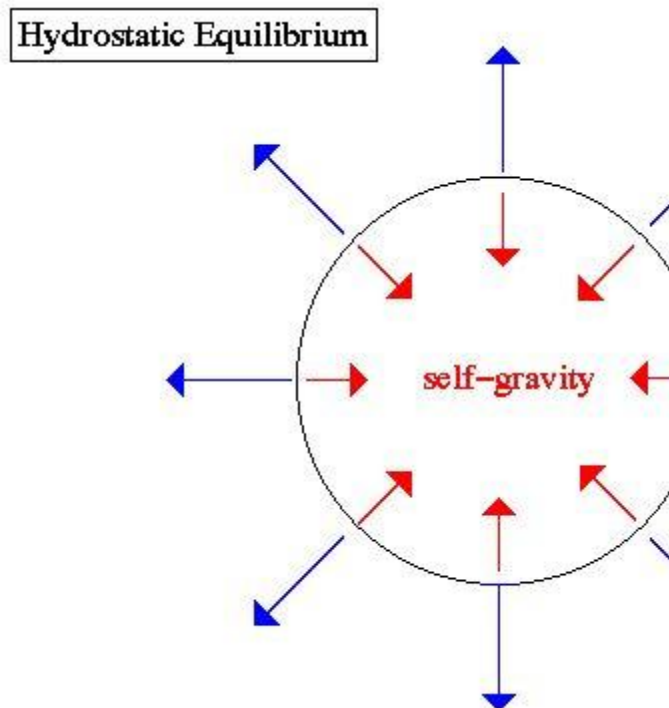
The fact that the mass does not disappear or turn into energy in most of the sun results in this simple (the only simple!) equation, which arises by dividing the **mass** in a thin shell at radius r by the **volume** of that radius to produce the **density** of that shell.



We'll pause a moment to notice that we have the derivative of one of our functions, dM_r/dr , on the left and one of our *other* functions, ρ_r , on the right. This is typical of a first order ODE. In this example, there is no multiplication with another function so it is *not* a **non-linear ODE** ... yet! But we have more equations to go, and if even one of them is non-linear, then the whole shebang is. When that happens, it will all but guarantee that we must use a computer to solve it.

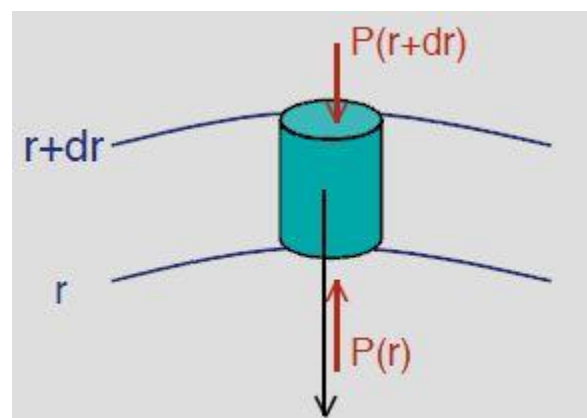
2nd Solar Structure Equation: Hydrostatic Equilibrium

$$\frac{dP_r}{dr} = -\rho_r \frac{GM_r}{r^2}$$



This embodies the most commonly expressed notion about the sun's internal workings, namely that the outward gas pressure from all internal reactions in the sun are balancing the inward forces due to gravity. If you look at this equation the right way, you'll "see" it: the left side of the equation is the net outward force due to the pressure differential in a tiny cylinder at radius r of sun-stuff. The right side is Newton's universal law of gravitation applied to that same tiny cylinder.

Again, we stop to note that the right side contains



the product of two unknown functions, making this immediately a ***non-linear ODE***. Ouch. We now know we'll need a numerical, not an analytical, solution. Hello C++ or Java.

3rd Solar Structure Equation: Energy Transport by Radiative Equilibrium

This is a very difficult equation to explain, so we just have to wait until we take an upper division physics course at Stanford or UC Berkeley before we can really understand it. See you there. Loosely speaking, this equation expresses how energy travels through the sun's layers by the mechanism of radiation.

$$\frac{dT_r}{dr} = -\frac{3}{4ac} \frac{\kappa \rho_r}{T_r^3} \frac{L_r}{4\pi r^2}$$

There are several parameters in this equation (aside from three of our main functions that we already defined). They are:

- ***a*** - the *radiation constant* from statistical mechanics (related to the Stefan-Boltzmann constant)
- ***c*** - the speed of light
- ***κ*** - the *opacity* of the stellar material (how easy is it to see down there?)

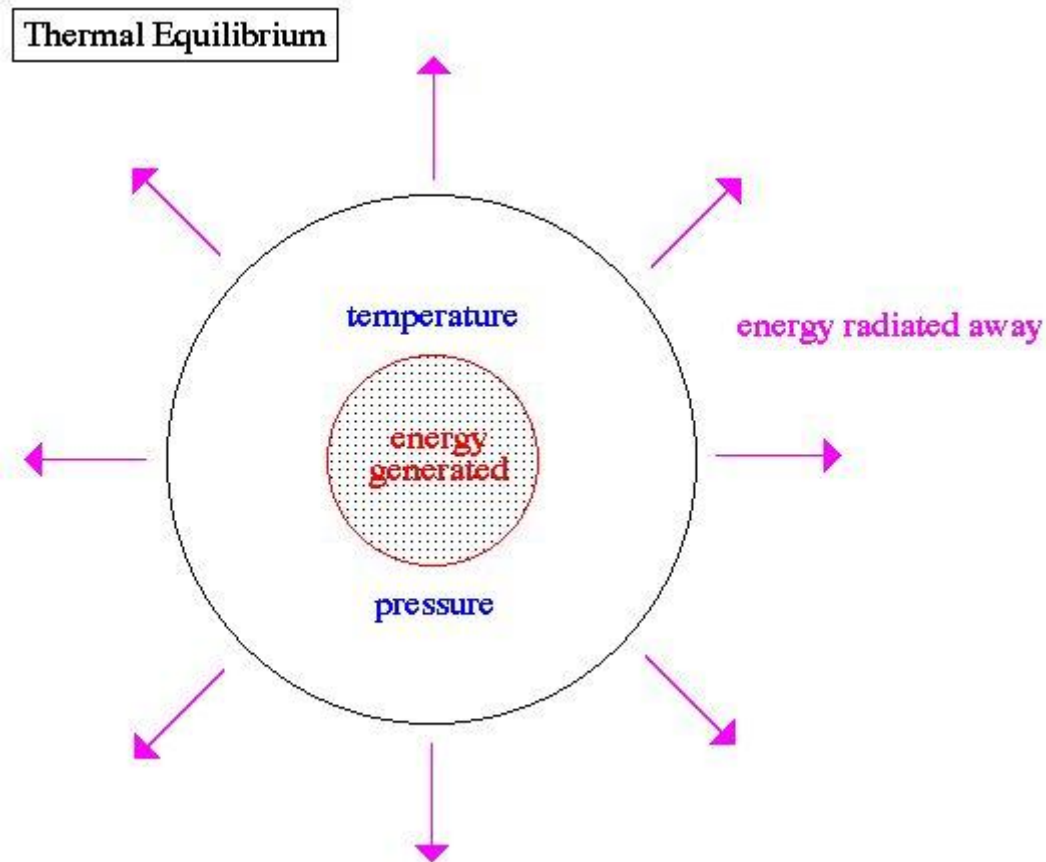
I said that we would be making several simplifying assumptions. One of them is that energy is transported only by *radiation* and not by another very important mechanism, *convection*. There is a similar equation for convective energy transport that I will not present here. In practice, radiation dominates at certain depths of a star and convection at other depths. In a numerical solution, the programmer would decide what depth, i.e., what ***r***, we were considering and use the proper equation -- radiative or convective -- for that layer.

As usual, we see that the right hand side is a product and quotient of three of our unknown functions, further contributing to its non-linearity.

4th Solar Structure Equation: Luminosity Relation

Here is a relation that equates the luminosity (energy) radiated outward from radius, ***r***, to the *energy production rate*, ***ε***, that feeds that radius from below. ***ε***, itself, is a function of the kind and number of nuclear reactions in the core of the sun.

$$\frac{dL_r}{dr} = 4\pi r^2 \rho_r \epsilon$$



By the way -- if we wanted to change the model to handle time in addition to radius (remember my simplifying assumption? I threw away time), then this is the equation that would get modified. The luminosity is dependent on a changing energy production rate *over time* in reality. Here also, is a place where astrophysicists try out different assumptions: ϵ is a function of more basic assumptions (unknown value -- guesses) that can be floated as hypotheses to see if they lead to a good model.

5th Solar Structure Equation: Equation of State

We are actually out of differential relations, but we need one more equation because there are five unknown functions and only four differential equations. A *system* of ODEs needs as many equations as unknown functions. We remedy this by using an ordinary *equation of state* which allows us to make a substitution and remove, say, **P** or **T** from the above. It is the well know $PV = nRT$ of thermodynamics in a sophisticated guise:

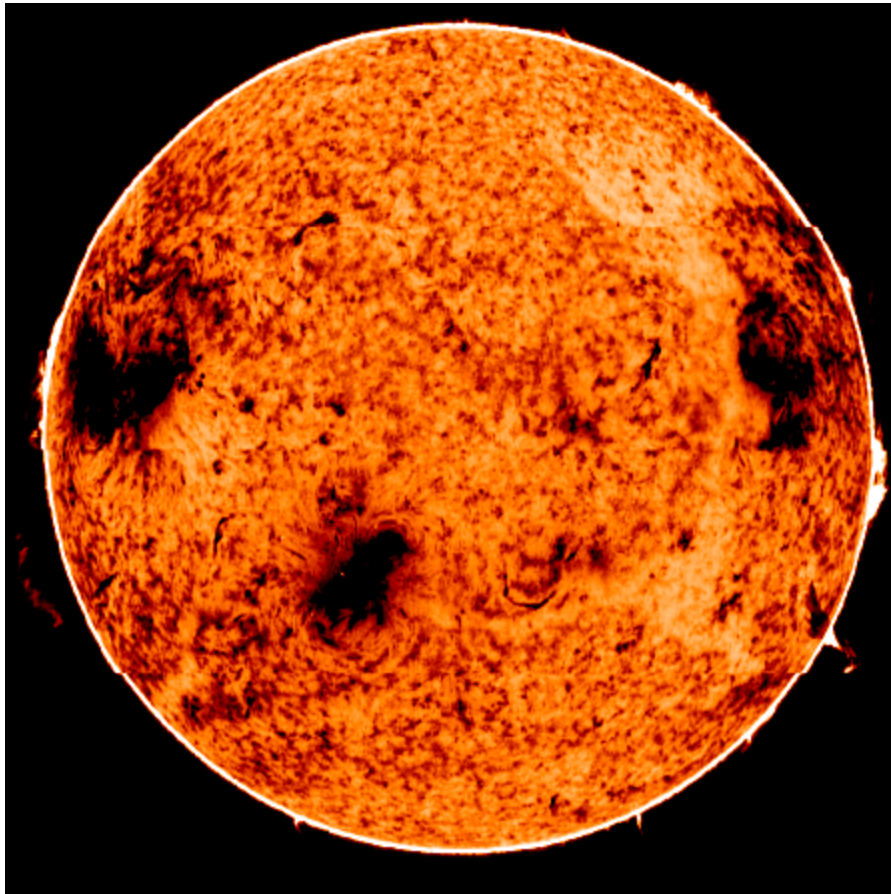
$$P_r = \frac{k}{\mu H} \rho_r T_r$$

- k is *Boltzmann's constant*
- μ is the *mean molecular weight* of the various elements in the Sun (which is where the helium abundance assumption comes in).
- H is the *mass of a proton*

5C.3.4 Conclusion

Well, there you have it. If you look up, you'll see five (actually four) simultaneous non-linear ordinary differential equations in five (actually four) unknown functions. We use a computer and a numerical technique to solve them, just as dozens of physics Ph.D. students have before us.

Remember, we will really learn to solve only a single non-linear ODE numerically, since the technique is easily extended to multiple, simultaneous equations, and we are just interested in the process.



Euler and Runge-Kutta Methods

5C.4.1 Numerical Analysis

When we cannot solve the ODE analytically, we must use the theory from a branch of mathematics called *numerical analysis*. In this theory, we learn to approximate the solution curves. For example, when we considered the *Initial Value Problem (IVP)*:

$$y' = x^2 + 3y, \text{ with initial value } y(0) = 0$$

We were able to find an exact (analytical) solution:

$$y = \frac{1}{27} (2e^{3x} - 9x^2 - 6x - 2)$$

But we can't always do that. Instead, we sometimes have to be satisfied with a **table of x-y values** that we believe *approximates* the solution to the IVP. This approximate solution is what we call the *numerical approximation to the true curve*. If the **x-y** pairs of the numerical solution are close enough together, and we plot them (and connect the dots), we will have what appears to be a smooth curve. This, then, is what we mean by a "numerical solution" to an IVP. Let's demonstrate the idea on a problem that we can easily solve analytically, so we are able to compare the analytical and numerical approaches.

$$y' = 2x, \text{ with initial value } y(1) = 3$$

What function has the derivative $2x$? Easy: $y = x^2 + C$ for any constant, C . Now we plug in the initial condition and find that the constant $C = 2$, so solution for this initial value problem is

$$y = x^2 + 2$$

That's the *analytical solution*. It is exact. However, if we pretend that we cannot find this solution using calculus, what would the numerical solution be? It would be a table of **x-y** pairs that lie along, or very close to, the actual solution. An example of a numerical solution to this IVP might be:

x:	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
y:	3.00	3.20	3.42	3.66	3.92	4.20	4.50	4.82	5.16	5.52	5.90

There are two features about these numbers that you should note:

1. **The y-values are not all correct.** This has nothing to do with computer round-off error. It is error inherent in every numerical method. A characteristic feature of a numerical solution is that, even with a computer of infinite precision, it does not yield a precise result. By their nature, numerical techniques solve problems by approximation. Any computer round-off error must be added to this inherent numerical error when you assess the solution's usefulness.
2. **The y-values seem to get less accurate as they get farther from the initial conditions (1,3).** This is not always true, but is often true of numerical approximations to ODE solutions.

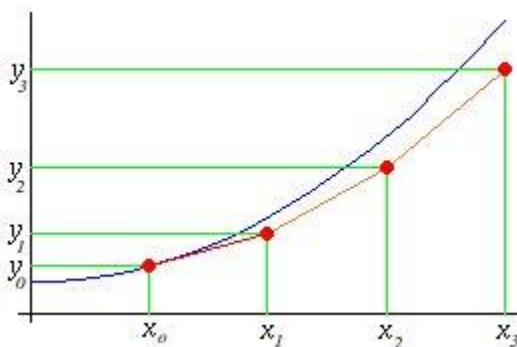
Remember, we would not use the numerical approach if we could solve the problem analytically. In our simple example, we can see exactly what needs to be done with the *bad* y values to make them *good*. But in practice we have *only* these bad values. Sometimes we can get an error estimate that gives an upper bound on just how bad the computed y-values are.

Don't despair. With today's computing power, we can devise numerical solutions whose precision is very good. So good, in fact, that they may as well be the actual functional values. Therefore, we are not disheartened by the inherent error in numerical analysis solutions to ODEs.

It is important to remember just what we are going to get as output of a numerical method. It will be a table like the one above, or some representation of that table, not a formula for the solution function.

5C.4.2 The Euler Method

Because time and space are short and this is not a course in numerical analysis or differential equations, I will give a very brief preamble that describes the logic behind the **Runge-Kutta (RK)** method of solving ODEs numerically. This is the historical and conceptual precursor of **RK**, namely the **Euler Method**.



From calculus we know that very close to any point on the graph of a function, we can approximate the function by the *tangent line* (which involves the first derivative). Well, the tangent lines are what we *do* have - that's what the ODE is - a bunch of tangent lines everywhere on the x - y plane. There is no shortage of *those*. So we use the tangent lines as if they were the actual solution, but we only use them for a short distance. Start with (x_0, y_0) and pick a small increment h to generate a sequence of x -values for our table: $x_1 = x_0 + h, x_2 = x_1 + h, \dots$

Close to x_0 we approximate the true, unknown, solution $y = f(x)$ at the initial value (x_0, y_0) by the straight line:

$$y = y_0 + y'(x_0) x$$

(Remember that y_0 and $y'(x_0)$ are provided instantly by the ODE.) If we plug our increment, h , into the above linear approximation this gives us a y_1 which is close to, but not really equal to, the true $y = f(x_1)$:

$$y_1 = y_0 + y'(x_0) h$$

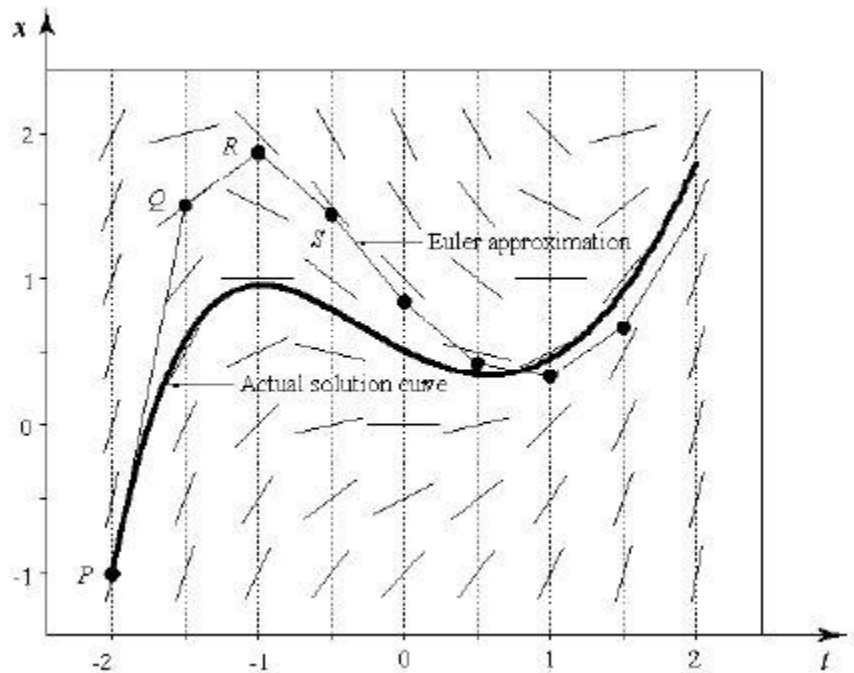
We now have the pair (x_1, y_1) . (See the dot directly above x_1 in the above graph, and notice that it lies a little below the actual curve.)

The ODE, again, will give us a slope, y' at this new position, (x_1, y_1) . and we use that to construct a *new* linear approximation to the solution. This can be used to get a value y_2 which moves our approximate solution a little further down the road to the point (x_2, y_2) . We continue to generate as many pairs as we wish using the simple formula:

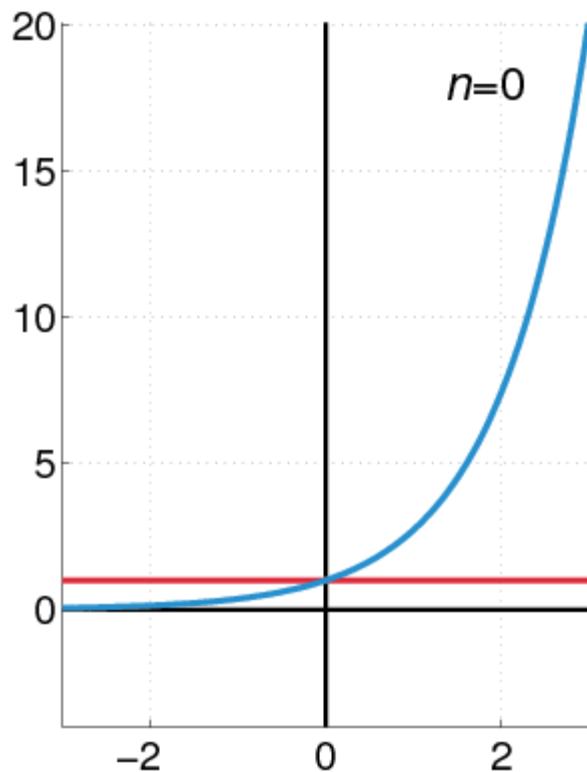
$$y_{k+1} = y_k + y'(x_k) h$$

As you can see from the diagram above, the subsequent tangent lines are really not tangent to the true solution, because as we move x a little further right, we are getting farther from the actual curve. But if we take smaller increments $h = x_{k+1} - x_k$ the error gets smaller.

What I just described is called the **Euler Method**, but it gives the idea of the algorithm we are going to use. **Runge-Kutta** uses higher order approximations -- parabolas and cubics -- rather than straight lines, which makes the approximation more precise. Stated in terms of the **Taylor series** of the solution function, **Euler** uses only the *linear term* (which is based on y') while **Runge-Kutta** uses fourth order terms (up through $y^{(4)}$, where the superscript



refers to the fourth *derivative*, not the fourth *power*). This means we can get a more accurate solution than *Euler* even if we use the same step-size, h . As you can see by the animated gif, below (from the Wiki page on *Taylor series*), the more terms you use in the Taylor series, the better the approximation to the true function. *Runge-Kutta* stops at the fourth order (using, effectively, the derivatives of order 1 through 4).



5C.4.3 The *4th Order Runge-Kutta* Algorithm

We generate the table of solutions progressively, starting with the initial condition as our first pair ((1,3) in the table above), and we create subsequent (x_k, y_k) pairs using values of previously computed pairs. We assume that we have the following initial value problem:

$$\begin{aligned}\frac{dy}{dx} &= G(x, y) \\ y(x_0) &= y_0\end{aligned}$$

We start from $x = x_0$ and continuing until we have reached some final value $x = x_{final}$.

Here is the rule for generating new pairs:

$$\begin{aligned}x_k &= x_k + h \\ y_k &= y_k + \frac{F_1 + 2F_2 + 2F_3 + F_4}{6}\end{aligned}$$

where the F_i are defined as follows:

$$\begin{aligned}F_1 &= h G(x_k, y_k) \\ F_2 &= h G\left(x_k + \frac{h}{2}, y_k + \frac{F_1}{2}\right) \\ F_3 &= h G\left(x_k + \frac{h}{2}, y_k + \frac{F_2}{2}\right) \\ F_4 &= h G(x_k + h, y_k + F_3)\end{aligned}$$

At this point, you have all the information you need to implement your own **RK** algorithm. The outline is relatively simple. (Well, almost -- if you are reading this from the first course in the computer science sequence, you will need to read ahead one module to next week's topic: *functions* -- AKA *methods* -- in order to make this all work.)

Here are the steps:

1. Hard-wire (i.e., incorporate the expression directly into your program) the function $G(x,y)$ defining the differential equation to be solved. You can do this as a simple Java or C++ function or method.
2. If the function $G()$ takes adjustable parameters that you want to play with by trial-and-error, make these global or static **constants** (*globals* are okay if they are constants) that are easy to find at the top of program so they can be quickly modified between compilations. *In the next section, there will be a parameter **beta** (β), in an epidemic modeling equation. That's an example of an adjustable parameter that could be set to a constant in your program.*
3. Write an **RK()** function, again as a Java or C++ function, that takes three parameters: the **old-x**, the **old-y** and the **h-increment**, and returns the new **y-value**.
4. Inside this **RK()** function, create the four helper values **F1** through **F4** (don't try to be clever and do this without helper variables - that's how you get fired from a job!) Simply translate the above algorithm directly into your **RK()** function.

5. In your **main()**, set an initial (x,y) value (or get it from the user) to establish the initial condition for your loop.
6. In your **main()**, set an **h -increment** (or get it from the user) to be used in the loop iteration to generate successive x -values.
7. In your **main()**, begin a loop that generates a new (x,y) from the previous (x,y) using the above expressions.
8. Decide what to do with each (x,y) as you generate it. If you don't need to save it in an array because, for example, you can use it instantly (display, graph, etc.) then don't. A good programmer would not use an array for no reason here. However, if you need a table for future use, then you may store into an array.
9. Stop when you have reached $x = x_{final}$.

This can be implemented in many different ways depending on the intent of the programmer and intended use of the program. Ideally, it would be set up in such a manner that could be easily re-used for many ODEs and even incorporated into a larger program that solved a system of ODEs.

Step back and consider where we landed. Despite some lofty science and heady expressions early in this module, we ended up with a rather simple algorithm that actually does the work. In a way, you might get the sense that the algorithm itself, once it has been established, can be programmed by a beginning programmer with only about three or four weeks of programming training. And you would be right! The development of the algorithm is the hard part - the program is the easy part. Once you understand this, you will be emboldened to research the hundreds of other numerical techniques in ODEs, PDEs, multivariate statistics, etc. and implement them as your discipline, major or research project dictates.

Case Study: H1N1 Epidemic Modeling



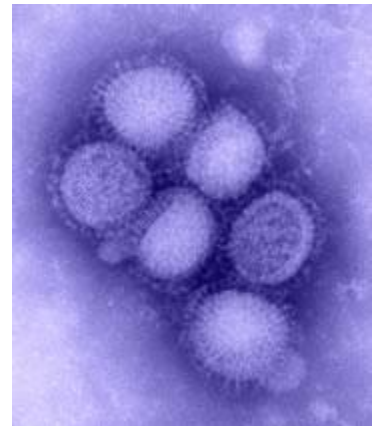
5C.5.1 Infectious Diseases

Let's apply the ODEs and the *Runge-Kutta* algorithm to a problem in epidemiology. The simplest model for an epidemic is one that assumes the following:

1. The population is split into two parts at any given time, t :
 - $x(t)$ = #susceptible individuals at time t
 - $y(t)$ = #infected individuals at time t

2. Infected = contagious.
3. The rate of growth of infected individuals, that is y' , is proportional to the product of xy , so $y' = \beta xy$ for some constant of proportionality, β . This is a reasonable assumption if the infected individuals interact actively and randomly with the entire population. If the # infected individuals doubles, then there will be twice as many of them touching the uninfected. Whatever the contagiousness coefficient, β , there should be twice as many new cases as a result. Similarly, if there are twice as many people in the uninfected population, more will be interacting with the infected, causing, again, twice as many new cases. So, if either of these numbers increases, the number of new infected will increase accordingly. And of course, if they both increase by, say, 2, then the number of new cases grows by $2*2 = 4$. This is the kind of analysis you need to do in your sleep if you do computer modeling.
4. The disease in question is mild in the sense that no appreciable number of individuals die as a result of contracting the disease. This is compatible with H1N1.
5. No one is immune.
6. Initially, at time $t = 0$, there are A individuals who have the disease, i.e., $y(0) = A$.
7. The total population is N , and this does not change over the period of the epidemic.

This is, as usual, an unrealistic model to use for all time, because it does not take into account the obvious reality of individuals healing and no-longer becoming infectious. Even so, there is a portion of the solution curve that matches actual population data for some highly infections but non-fatal diseases, H1N1 being an example (despite the fact that there are fatalities, when considering the size of the population, this is technically considered a mild, non-fatal disease for the purpose of mathematical modeling). To make this model more realistic and be applicable over a larger time interval to real epidemics, one would need to add more constraints and change the equations accordingly. However, the techniques do not change.



From above, we see that the ODE is:

$$y' = \beta xy$$

However, since $x + y = N$ for all time, we can change this to:

$$y' = \beta(N - y)y$$

We identify this immediately as a **first order non-linear ODE**, exactly the kind that we like to solve using *Runge-Kutta*. Yet this particular equation does have an analytic solution using some standard techniques learned in a course in elementary differential equations, and the solution, we would find, is:

$$y(t) = \frac{Na}{a + (N - a)e^{-\beta Nt}}$$

So, we have a nice single-function ODE on which we can test **RK** (*Runge-Kutta*) against the actual solution. Let's give it a shot. We use the following parameters which we can make global constants for the program:

- $N = 20$ million (total population)
- $A = 1$ (initial number of infected individuals = 1)
- $h = 5$ (5 days will be the sampling interval for **RK** initially)
- $\beta = .000000005$

For your optional assignment, you will estimate the solution using **RK** for about one year, starting at $t = 0$ and going to $t = 360$. Print intermediate results at intervals of **30** days. Each time you print, also display and compare with the true function $y(t)$ shown above to see how much error you get when you use **RK**. Try ***h*-intervals** of **2** days, **1** day and **.5** days (and less if you need to) to see how much error is produced when you sample at different ***h*-intervals**.

Let's review the main points here. We are seeking the solution function $y(t)$ which will tell us how many individuals are infected after t days. Although we have an actual formula for the simple model, we will use **RK** to generate the number of infected individuals at $t = 0$ days, $t = 30$ days, $t = 60$ days, etc. We will compare our answer with the actual value produced by the formula since we happen to know it. We will try an ***h*-interval** of **5** days (which has nothing to do with the 30 day reporting interval). Then we'll shorten this to **2** day increments, **1** day increments and even try less than one day increments (keeping the reporting interval at **30** days each time).

Here is one run using an undisclosed interval, ***h***:

```

C:\Windows\system32\cmd.exe
After 0 days, actual: 1 r.k.: 1
After 30 days, actual: 20.0855 r.k.: 20.0855
After 60 days, actual: 403.421 r.k.: 403.419
After 90 days, actual: 8099.8 r.k.: 8099.75
After 120 days, actual: 161441 r.k.: 161440
After 150 days, actual: 2.80976e+006 r.k.: 2.80973e+006
After 180 days, actual: 1.53304e+007 r.k.: 1.53303e+007
After 210 days, actual: 1.97012e+007 r.k.: 1.97012e+007
After 240 days, actual: 1.99849e+007 r.k.: 1.99849e+007
After 270 days, actual: 1.99992e+007 r.k.: 1.99992e+007
After 300 days, actual: 2e+007 r.k.: 2e+007
After 330 days, actual: 2e+007 r.k.: 2e+007
After 360 days, actual: 2e+007 r.k.: 2e+007
Press any key to continue . . .

```

Notice how remarkably accurate this is. For what value of h do you suppose we were able to get this level of accuracy? When you write the program (did I mention that you'll write the program?) you'll be surprised to see the answer.

See the text *Mathematical Modelling* [SIC] by Caldwell and Ram for more on this model.

Programming the Equations

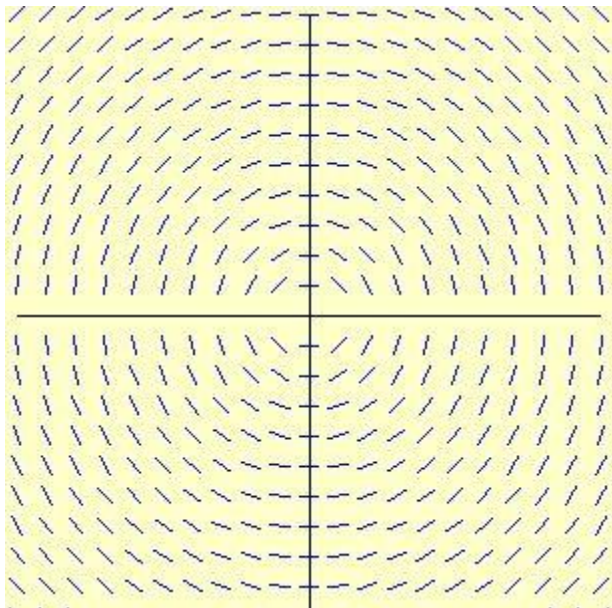
5C.6.1 An Example R-K Program Solved

Somehow, we need to bring this all together in a program. The idea is not to create a difficult problem that students need to solve in order to get a few measly points, but to demonstrate how easy it can be to set up a numerical solution in a Java program, and do it in such a way that we can use this solution for many similar problems in years to come.

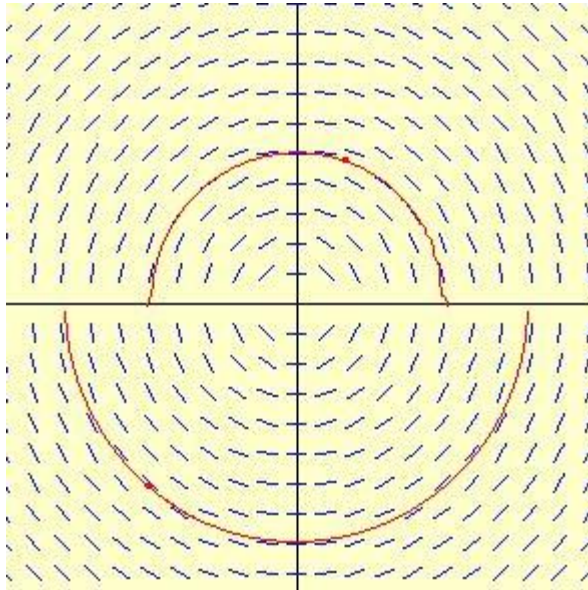
In order to prove that our numbers are reliable, we'll solve a very simple ODE that has an analytical solution to which we can compare our **RK** approximation. Here is the **G()** that defines our ODE:

$$\frac{dy}{dx} = \frac{x}{y}$$

This is an interesting ODE. It looks pretty simple. Before I show you the analytical solution, look at a picture of the slope field that it generates:



Guess what the solution curves are? Here are a couple using two different initial conditions:



Yes, they are *circles*. Of course, entire circles are not expressible as functions - the best we can do is declare a relation, such as $x^2 + y^2 = R^2$, which is not a function. We have to solve for y , and pick either a positive or negative square-root. And when we do that we only get *half* a circle (either the upper or lower half). Here is the initial condition:

$$y(0) = 1000$$

and analytical solution to this *initial value problem (IVP)* we will use for our program:

$$y(x) = \sqrt{1000^2 - x^2}$$

And this is all we need. We will use our **RK** algorithm to solve this *IVP*, and compare the results with the actual solution above.

5C.6.2 The Program

We start with the sample **main()**, removing everything except the raw intent of the program. We are looking to see how this client makes use of the methods (that we have yet to write):


```

public static void main(String[] args)
{
    // stuff, then ...

    for (x = x_start, ... ; x < x_end ; x += h_inc, ...)
    {
        // show the results periodically - we don't need to see them every h_inc
        if ( ... )
            System.out.println("Actual: " + Y_actual(x)
                               + "   r.k.: " + y_rk);

        // generate the next (x,y) pair
        y_rk = Y_runge_kutta(x, y_rk, h_inc);
    }
}

```

If we can digest this, then the rest will be easy.

- We have a loop that starts at some initial x -value, **x_start**, and goes until we reach some terminating x -value, **x_end**, and it increments in steps of **h_inc**. In math terms, we are trying to graph the function by plotting points from **x_start** to **x_end**. We can't graph *every* point, since there are infinitely many of them, so we sample at intervals of **h_inc**. Here's an example: **x_start = -15, x_end = 35, h_inc = 0.5**. We graph the function (i.e., report y -values) at **x = -15, -14.5 -14, -13.5 ... 34, 34.5**.
- We don't really need to see the results *every* time through the loop. We *do* need to do the **RK** computation every loop pass, because **RK** requires us to progressively generate pairs. But it would be too much to inspect every y -value, every pass, so we have an **if(...)** statement that will display, say, every 5 or 20 or 50 loop passes.
- Remember, we are going to do an example where we know the solution before-hand. This will be expressed by the function **Y_actual()**. Each time we display something, we show **Y_actual()** and compare that with our **RK** approximation, which is the number **y_rk**.
- In every loop pass, we need to compute a new **y_rk** using the previous **x_k**, **y_k** values. That's what the final statement in that loop does.

This, then, is the big picture.

As you may have discovered, when you write a program, there are a lot of ugly realities that make the beautifully simple ideas messy, and that's what happens next. I will add some parameters and constants that we can change easily in one place. They will define the initial conditions, frequency of the output, h -increment and loop termination. Here they are:

```

static final double RADIUS = 1000;
static final double INIT_Y = RADIUS;
static final double INIT_X = 0;
static final double H_INC = 5;

```

These constants presume that we know that the solution of the ODE is a circle with radius 1000 going through the point (0, 1000). That's the only reason we can give a constant a name like

RADIUS. Normally, we don't know the solution - that's why we write the program. From these basic values, we initialize some more convenient variables that are used in the loop:

```
h_inc = H_INC;
x_start = INIT_X;
y_start = INIT_Y;
x_end = RADIUS;
num_values_to_report = 20;
num_iterations = (int)((x_end - x_start)/ h_inc);
sample_rate = num_iterations/num_values_to_report;
if (sample_rate < 1)
    sample_rate = 1;
```

It's not crucial to study each of these statements, unless you plan on getting your hands dirty and modifying the program. Here is how the whole **main()** looks at this point, in all its gory detail:

```
import java.math.*;

public class Foothill
{
    // parameters and initial conditions that we can easily change here at the top
    static final double RADIUS = 1000;
    static final double INIT_Y = RADIUS;
    static final double INIT_X = 0;
    static final double H_INC = 5;

    public static void main(String[] args)
    {
        double h_inc, x_start, y_start, x_end, x, y_rk;
        int num_iterations, num_values_to_report, sample_rate, k;

        h_inc = H_INC;
        x_start = INIT_X;
        y_start = INIT_Y;
        x_end = RADIUS;
        num_values_to_report = 20;
        num_iterations = (int)((x_end - x_start)/ h_inc);
        sample_rate = num_iterations/num_values_to_report;
        if (sample_rate < 1)
            sample_rate = 1;

        for (x = x_start, y_rk = y_start, k = 0; x < x_end ; x += h_inc, k++)
        {
            // show the results periodically
            if ( k % sample_rate == 0)
                System.out.println("Actual: (" + (int)x + ", " + Y_actual(x) + ") "
                    + " r.k.: (" + (int)x + ", " + y_rk + ") ");

            // generate the next (x,y) pair
            y_rk = Y_runge_kutta(x, y_rk, h_inc);
        }
    }

    static double Y_runge_kutta(double x, double y_prev, double h_inc)
    {
        // to be done
    }

    static double Y_actual(double x)
    {
        // to be done
    }

    static double ODE(double x, double y)
    {
        // to be done
    }
}
```

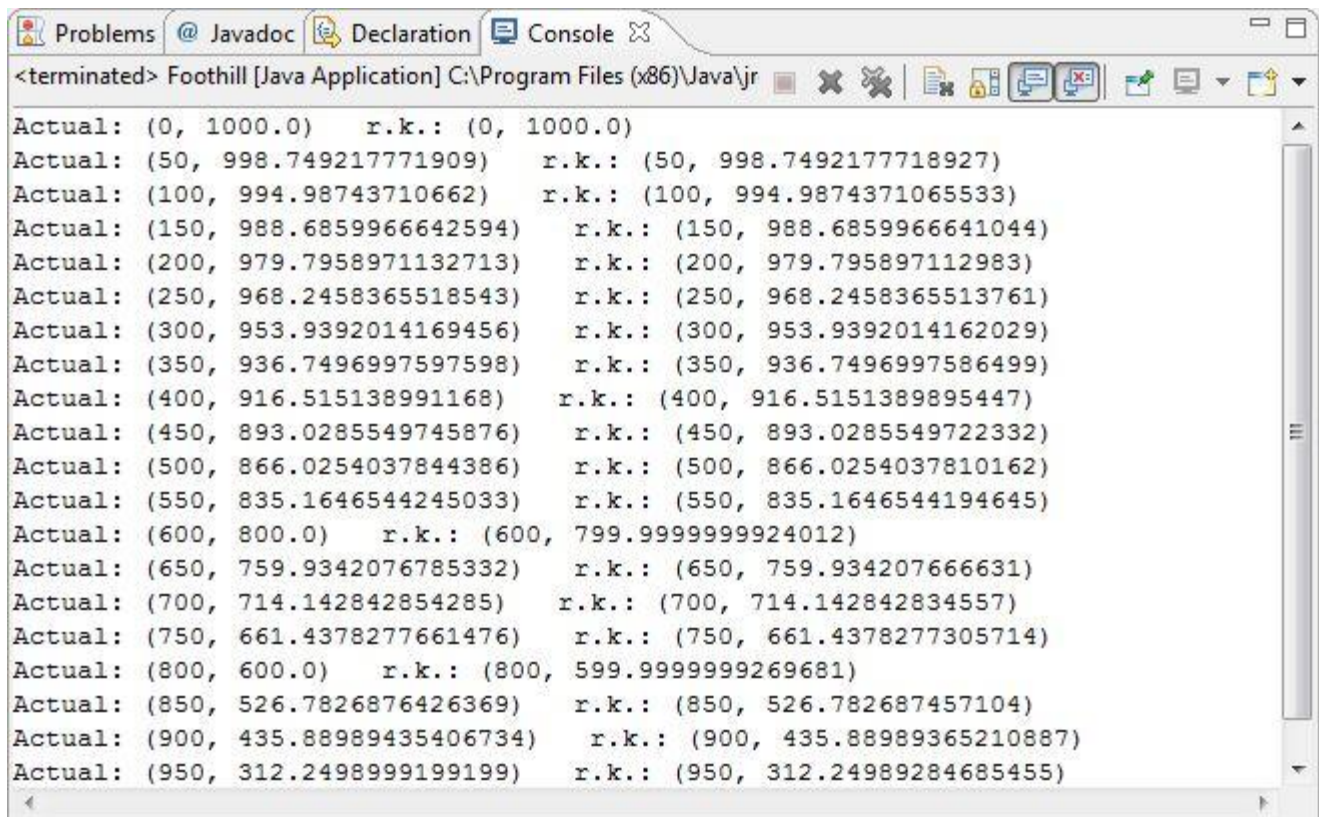
This leaves three functions to write:

- **Y_actual()**, the actual function that we know to be the solution to the ODE.
- **ODE()**, my **G()** - the function that defines the ODE (this is called from the **RK** method).
- **Y_runge_kutta()**, the function that takes the previous \mathbf{x}_k , \mathbf{y}_k values and generates a new \mathbf{y}_{k+1} from them.

Well, these three functions are described - if not actually written for you -- above and in the previous modules. I think I can safely leave those details to you.

5C.6.3 The Results

Run this program and try different values. Using my **h_inc** of **5**, here's what I get:



```
<terminated> Foothill [Java Application] C:\Program Files (x86)\Java\jr
Actual: (0, 1000.0)    r.k.: (0, 1000.0)
Actual: (50, 998.749217771909)    r.k.: (50, 998.7492177718927)
Actual: (100, 994.98743710662)    r.k.: (100, 994.9874371065533)
Actual: (150, 988.6859966642594)    r.k.: (150, 988.6859966641044)
Actual: (200, 979.7958971132713)    r.k.: (200, 979.795897112983)
Actual: (250, 968.2458365518543)    r.k.: (250, 968.2458365513761)
Actual: (300, 953.9392014169456)    r.k.: (300, 953.9392014162029)
Actual: (350, 936.7496997597598)    r.k.: (350, 936.7496997586499)
Actual: (400, 916.515138991168)    r.k.: (400, 916.5151389895447)
Actual: (450, 893.0285549745876)    r.k.: (450, 893.0285549722332)
Actual: (500, 866.0254037844386)    r.k.: (500, 866.0254037810162)
Actual: (550, 835.1646544245033)    r.k.: (550, 835.1646544194645)
Actual: (600, 800.0)    r.k.: (600, 799.9999999924012)
Actual: (650, 759.9342076785332)    r.k.: (650, 759.934207666631)
Actual: (700, 714.142842854285)    r.k.: (700, 714.142842834557)
Actual: (750, 661.4378277661476)    r.k.: (750, 661.4378277305714)
Actual: (800, 600.0)    r.k.: (800, 599.9999999269681)
Actual: (850, 526.7826876426369)    r.k.: (850, 526.782687457104)
Actual: (900, 435.88989435406734)    r.k.: (900, 435.88989365210887)
Actual: (950, 312.2498999199199)    r.k.: (950, 312.24989284685455)
```

Look at these numbers. At this point you should be in awe of **RK**: we are taking an **h_inc** of **5**. Not **.0005** or **.05**, but **5**! And with this much slop, we are getting nine significant figures of agreement between the actual solution and the **RK** approximation. This gives you an appreciation of the power of **RK**.