Numerical Methods for Solving Differential Equations

Euler's Method

Theoretical Introduction

Throughout this course we have repeatedly made use of the numerical differential equation solver packages built into our computer algebra system. Back when we first made use of this feature I promised that we would eventually discuss how these algorithms are actually implemented by a computer. The current laboratory is where I make good on that promise.

Until relatively recently, solving differential equations numerically meant coding the method into the computer yourself. Today there are numerous solvers available that can handle the majority of classes of initial value problems with little user intervention other than entering the actual problem. However, occasionally it still becomes necessary to do a some customized coding in order to attack a problem that the prewritten solvers can't quite handle.

This laboratory is intended to introduce you to the basic thinking processes underlying numerical methods for solving initial value problems. Sadly, it probably won't turn you into an expert programmer of numerical solver packages (unless some miracle occurs.)

The laboratory introduces you to a very simple technique for handling **first order initial value problems**. Like so many other concepts in mathematics, it is named after Leonhard Euler (1707-1783), perhaps the most prolific mathematician of all time. Before we can begin to describe **Euler's Method**, we must first make sure that we understand the nature of these approximate numerical solutions that his idea makes it possible for us to find.

The Nature of Numerical Solutions

From the point of view of a mathematician, the *ideal* form of the solution to an initial value problem would be a **formula** for the solution function. After all, if this formula is known, it is usually relatively easy to produce any other form of the solution you

may desire, such as a **graphical solution**, or a **numerical solution** in the form of a table of values. You might say that a formulaic solution contains the recipes for these other types of solution within it. Unfortunately, as we have seen in our studies already, obtaining a formulaic solution is not always easy, and in many cases is absolutely impossible.

So we often have to "make do" with a numerical solution, i.e. a table of values consisting of points which lie along the solution's curve. This can be a perfectly usable form of the answer in many applied problems, but before we go too much further, let's make sure that we are aware of the shortcomings of this form of solution.

By it's very nature, a numerical solution to an initial value problem consists of a **table of values** which is **finite** in length. On the other hand, the true solution of the initial value problem is most likely a whole continuum of values, i.e. it consists of an *infinite* number of points. Obviously, the numerical solution is actually leaving out an infinite number of points. The question might arise, "With this many holes in it, is the solution good for anything at all?" To make this comparison a little clearer, let's look at a very simple specific example.

An Example

Say we were to solve the initial value problem:

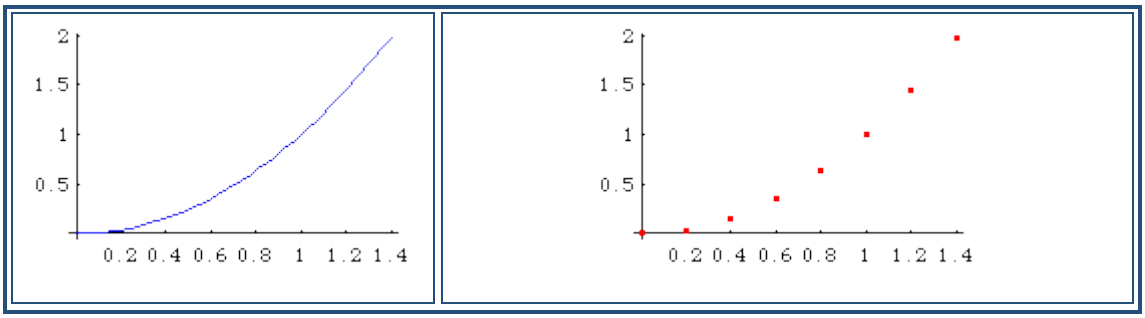
$$y' = 2x$$

$$y(0) = 0$$

It's so simple, you could find a formulaic solution in your head, namely $y = x^2$. On the other hand, say we were to use a numerical technique. (Yes, I know we don't know how to do this yet, but go with me on this for a second!) The resulting numerical solution would simply be a table of values. To get a better feel for the nature of these two types of solution, let's compare them side by side, along with the graphs we would get based on what we know about each one:

Formulaic Solution	Numerical Solution
$y = x^2$	

x	0.0	0.2	0.4	0.6	0.8	1.0	1.2	1.4
y	0.00	0.04	0.16	0.36	0.64	1.00	1.44	1.96



Notice that the graph derived from the **formulaic** solution is **smoothly continuous**, consisting of an infinite number of points on the interval shown. On the other hand, the graph based on the numerical solution consists of just a bare eight points, since the numerical method used apparently only found the value of the solution for *x*-increments of size 0.2.

Using Numerical Solutions

So what good is the numerical solution if it leaves out so much of the real answer? Well, we can respond to that question in several ways:

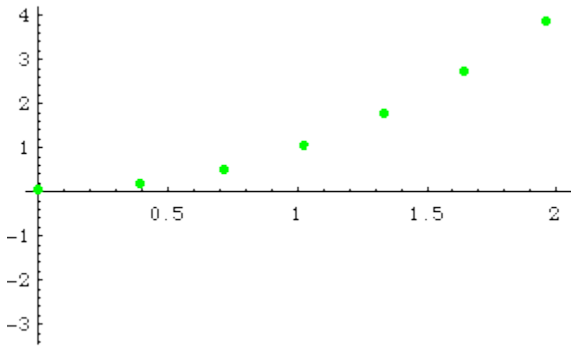
- The numerical solution still looks like it is capturing the general trend of the "real" solution, as we can see when we look at the side-by-side graphs. This means that if we are seeking a qualitative view of the solution, we can still get it from the numerical solution, to some extent.
- The numerical solution could even be "improved" by playing "join-the-dots" with the set of points it produces. In fact this is exactly what some solver packages, such as *Mathematica*, do do with these solutions. (*Mathematica*produces a join-the-dots function that it calls`InterpolatingFunction`.)
- When actually *using* the solutions to differential equations, we often aren't so much concerned about the nature of the solution at *all* possible points. Think about it! Even when we are able to get formulaic solutions, a typical use we make of the formula is to substitute values of the independent variable into the formula in order to find the values of the solution at specific points. Did you hear that? Let me say it again: **to find the values of the solution at specific points**. This is exactly what we can *still* do with a *numerical*solution.

The Pitfalls of Numerical Solutions

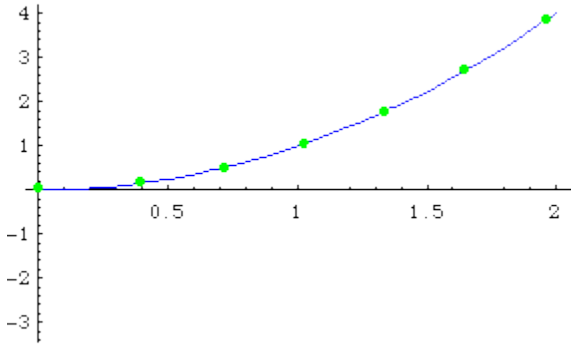
One last word of warning, however, before we move on to actually finding numerical solutions. In a problem where a numerical solution would really be necessary, i.e. one which we can't solve by any other method, there is no formulaic solution for us to

compare our answers with. This means that there is always an element of doubt about the data we produce by using these numerical techniques.

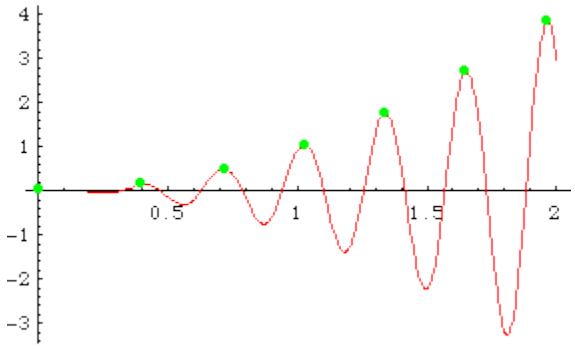
Say, for example, you obtained a set of numerical data as a solution, which led to the following graph:



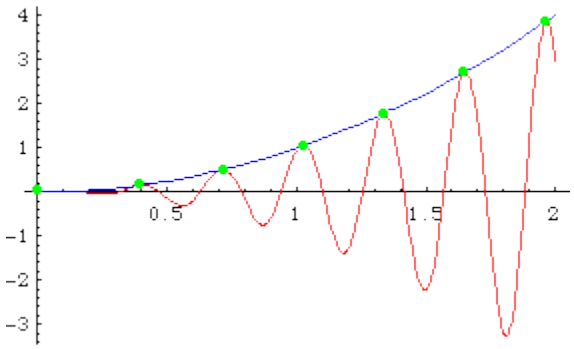
Any reasonable observer of this picture would, in the absence of any other evidence, assume that the underlying solution had a graph that looks something like this:



In other words, you'd play join-the-dots visually. But how do you know that the *actual* underlying solution doesn't look like this:

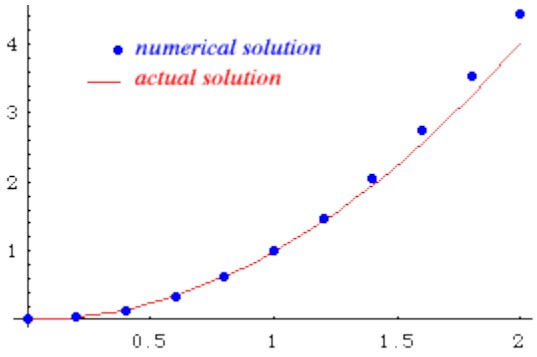


Notice that this graph fits the data points just as well as the first attempt we made at joining the dots, see:



So how can you tell whether or not your data is leading you to the wrong conclusion? There are ways, both qualitative and quantitative, that can be used to help in making this kind of decision. A whole field of study called *numerical analysis* is dedicated to answering this sort of question. Suffice it to say, that in reality, the kind of error I've just illustrated with the above pictures never occurs. I deliberately used an "off-the-wall" example to get your attention.

In reality, the kind of errors we need to be careful of are much more subtle. What tends to happen with numerical solutions is that the *calculated points drift further and further away from the actual solution* as you move further and further from the point defined by the initial condition. This can lead you to make assumptions about the actual solution which aren't true.



This difficulty can be overcome to some degree by calculating these points *closer together*. The penalty for this, of course, is that *more* points must be found, so the computer has to spend more time finding the solution. Most computer solvers try to strike a compromise between the accuracy inherent in using more points, and the extra time required to calculate the additional points. (A third issue involves machine round-off errors, but we won't even start to talk about that here.)

Now that we have been exposed a little to the type of solutions that we'll be finding, and the problems inherent in this kind of solution, it's **time we found out how they can be generated**.

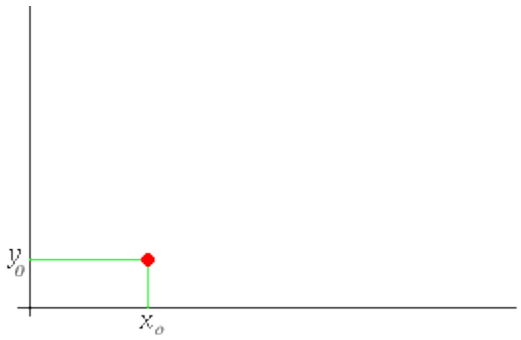
Developing Euler's Method Graphically

In order to develop a technique for solving first order initial value problems numerically, we should first agree upon some notation. We will assume that the problem in question can be algebraically manipulated into the form:

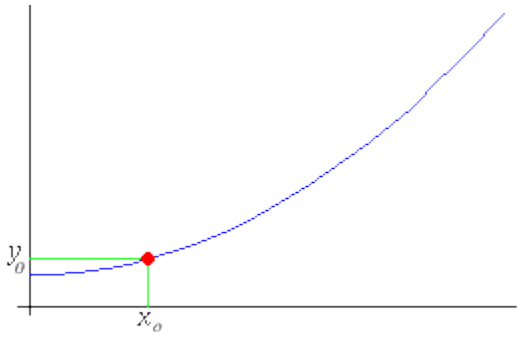
$$y' = f(x, y)$$

$$y(x_0) = y_0$$

Our goal is to find a *numerical solution*, i.e. **we must find a set of points which lie along the initial value problem's solution**. If we look at this problem, as stated above, we should realize that we actually already know *one* point on the solution, namely the one defined by the initial condition, (x_0, y_0). Possibly, the picture of this tiny piece of information looks something like this:



Now, remember we don't really know the true solution of the problem, or we wouldn't be going through this method at all. But let's act as if we *do* know this elusive solution for a moment. Let's pretend that it's "ghostly graph" could be superimposed onto our previous picture to get this:



Again, the blue graph of the true solution, shown above, is actually *unknown*. We've drawn a picture of what it *might* look like just to help us think.

Since we're after a set of points which lie along the true solution, as stated above, we must now derive a way of generating more solution points in addition to the solitary initial condition point shown in red in the picture. How could we get more points?

Well, look back at the original initial value problem at the top of the page! So far we have only used the initial condition, which gave us our single point. Maybe we should consider the possibility of utilizing the other part of the initial value problem—the differential equation itself:

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

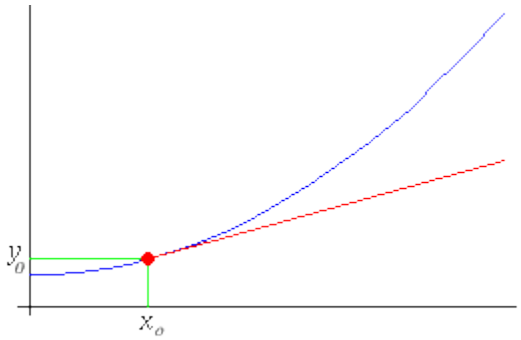
Remember that one interpretation of the quantity y' appearing in this expression is as the *slope of the tangent line* to the function y. But, the function y is exactly what we are seeking as a solution to the problem. This means that we not only know a point which lies on our elusive solution, but we also know a formula for its slope:

$$\textit{slope of the solution} = f(x, y)$$

All we have to do now is think of a way of using this slope to get those "other points" that we've been after! Well, look at the right hand side of the last formula. It looks like you can get the slope by substituting values for x and y into the function f. These values should, of course, be the coordinates of a point lying on the solution's graph—they can't just be the coordinates of any point anywhere in the plane. Do we know of any such points—points lying on the solution curve? Of course we do! The initial condition point that we already sketched is exactly such a point! We could use *it* to find the slope of the solution at the initial condition. We would get:

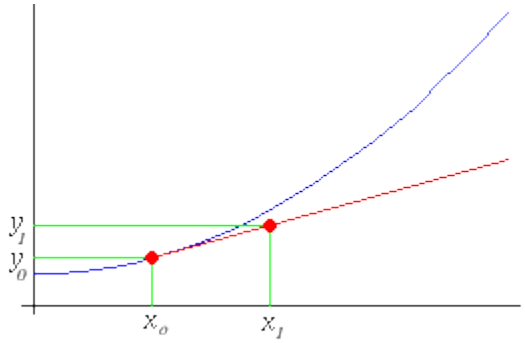
$$\textit{slope of the solution at } (x_0, y_0) = f(x_0, y_0)$$

Remembering that this gives us the *slope of the function's tangent line at the initial point* we could put this together with the initial point itself to build the tangent line at the initial point, like this:



Once again, let's remind ourselves of our goal of finding more points which lie on the true solution's curve. Using what we know about the initial condition, we've built a tangent line at the initial condition's point. Look again at the picture of this line in comparison with the graph of the true solution in the picture above. If we're wanting other points along the path of the true solution, and yet we don't actually have the true solution, then it looks like using the tangent line as an approximation might be our best bet! After all, at least on this picture, it looks like the line stays pretty close to the curve if you don't move too far away from the initial point.

Let's say we move a short distance away, to a new x-coordinate of x_1. Then we could locate the corresponding point lying on our tangent line. (We can't do this for the curve—it's a ghost, remember!) It might look something like this:

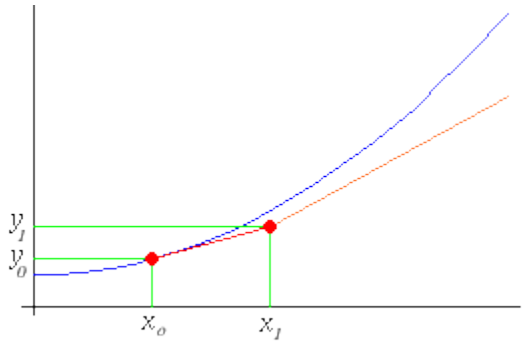


Notice that our new point, which I've called (x_1, y_1), isn't too terribly far away from the true value of the solution at this x-coordinate, up on the curve.

So we now have *two* points as part of our numerical solution:

- (x_0, y_0): an exact value, known to lie on the solution curve.
- (x_1, y_1): an approximate value, known to lie on the solution curve's tangent line through (x_0, y_0).

We must now attempt to continue our quest for points on the solution curve (though we're starting to see the word "on" as a little optimistic—perhaps "near" would be a more realistic word here.) Still glowing from our former success, we'll dive right in and repeat our last trick, constructing a tangent line at our new point, like this:

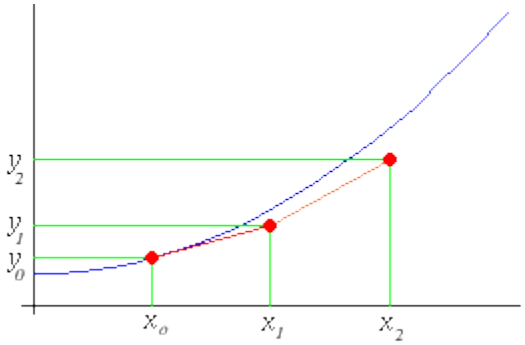


You should immediately recognize that there's a problem, and I've cheated to overcome it! Since our new point didn't actually lie on the true solution, we can't actually produce a tangent line to the solution at this point. (We don't even know where the true solution actually is anymore—the blue curve in the picture is just a thinking aid.) But we can still substitute our new point, (x_1,y_1), into the formula:

$$\textit{slope of the solution} = f(x,y)$$

to get get the slope of a *pseudo*-tangent line to the curve at (x_1,y_1). We hope that our approximate point, (x_1,y_1), is close enough to the real solution that the *pseudo*-tangent line is pretty close to the unknown real tangent line.

We now attempt to use this new *pseudo*-tangent line to get yet another point in the approximate solution. As before, we move a short distance away from our last point, to a new x-coordinate of x_2. Then we locate the corresponding point lying on our *pseudo*-tangent line. The result might look something like this:



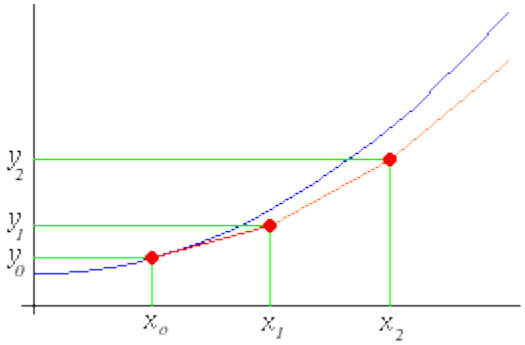
We now have *three* points in our approximate solution:

- (x_0, y_0): an exact value, known to lie on the solution curve.
- (x_1, y_1): an approximate value, known to lie on the solution curve's tangent line through (x_0, y_0).

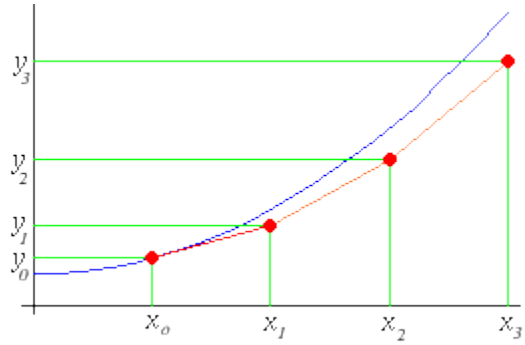
- (x_2, y_2): an approximate value, known to lie on the solution curve's pseudo-tangent line through (x_1, y_1).

As you can see, we're beginning to establish a pattern in the way we are generating new points. We could continue making new points like this for as long as we liked, but for the sake of this illustration let's find just one more value in the approximate solution.

We make another *pseudo*-tangent line, this time through (x_2, y_2), like this:



and we make another short jump to an x-coordinate of x_3, and locate the corresponding point on our latest *pseudo*-tangent line, like this:



So the list of points in our approximate numerical solution now has four members:

- (x_o, y_o): an exact value, known to lie on the solution curve.
- (x_1, y_1): an approximate value, known to lie on the solution curve's tangent line through (x_o, y_o).
- (x_2, y_2): an approximate value, known to lie on the solution curve's pseudo-tangent line through (x_1, y_1).
- (x_3, y_3): an approximate value, known to lie on the solution curve's pseudo-tangent line through (x_2, y_2).

Looking over the picture one last time, we see an example of how the numerical solution—the red dots, might compare with the actual solution. As we stated in the introduction to this laboratory, a weakness of numerical solutions is their tendency to drift away from the true solution as the points get further away from the initial condition point. One way of minimizing (but not eliminating) this problem is to make sure that the jump-size between consecutive points is relatively small.

Now that you've seen the pictorial version of Euler's Method for finding numerical solutions, you should have a better chance of understanding the derivation of the formulas used by the method. So, **let's go** and derive them...

Deriving the Euler's Method Formulas

Reminder: We're solving the initial value problem:

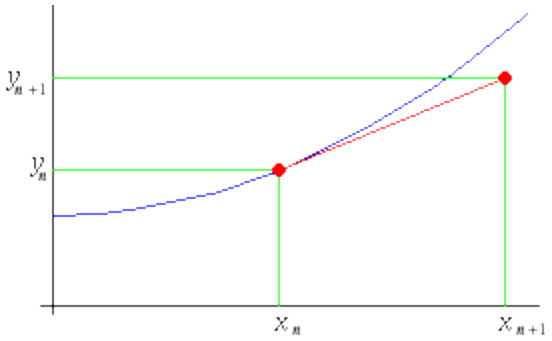
$$y' = f(x, y)$$

$$y(x_0) = y_0$$

As we just saw in the graphical description of the method, the basic idea is to use a known point as a "starter," and then use the tangent line through this known point to jump to a new point. Rather than focus on a particular point in the sequence of points we're going to generate, let's be generic. Let's use the names:

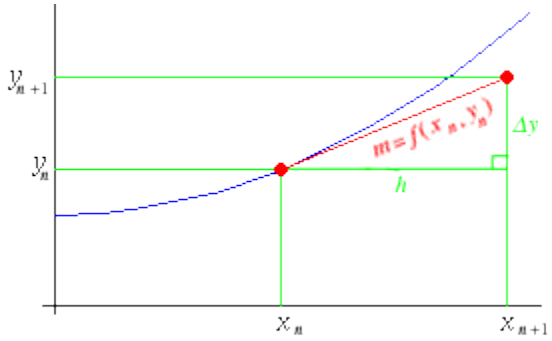
- (x_n, y_n) for the known point
- (x_{n+1}, y_{n+1}) for the new point

Our picture, based on previous experience, should look something like this:



(Though the proximity of the true solution to the point (x_n, y_n) is, perhaps, a little optimistic.)

Our task here is to find formulas for the coordinates of the new point, the one on the right. Clearly it lies on the tangent line, and this tangent line has a known slope, namely $f(x_n, y_n)$. Let's mark on our picture names for the sizes of the x-jump, and the y-jump as we move from the known point, (x_n, y_n), to the new point. Let's also write in the slope of the tangent line that we just mentioned. Doing so, we get:



The formula relating x_n and x_{n+1} is obvious:

$$x_{n+1} = x_n + h$$

Also, we know from basic algebra that *slope = rise / run*, so applying this idea to the triangle in our picture, the formula becomes:

$$f(x_n, y_n) = \Delta y / h$$

which can be rearranged to solve for Δy giving us:

$$\Delta y = h\, f(x_n, y_n)$$

But, we're really after a formula for y_{n+1}. Looking at the picture, it's obvious that:

$$y_{n+1} = y_n + \Delta y$$

And, replacing Δy by our new formula, this becomes:

$$y_{n+1} = y_n + h\, f(x_n, y_n)$$

And that's it! We've derived the formulas required to generate a numerical solution to an initial value problem using Euler's Method. Let's **go and look** at a summary of the method.

Summary of Euler's Method

In order to use Euler's Method to generate a numerical solution to an initial value problem of the form:

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

$$y(x_o) = y_o$$

we decide upon what interval, starting at the initial condition, we desire to find the solution. We chop this interval into small subdivisions of length h. Then, using the initial condition as our starting point, we generate the rest of the solution by using the iterative formulas:

$$x_{n+1} = x_n + h$$

$$y_{n+1} = y_n + h\, f(x_n, y_n)$$

to find the coordinates of the points in our numerical solution. We terminate this process when we have reached the right end of the desired interval.

A Preliminary Example

Just to get a feel for the method in action, let's work a preliminary example completely by hand. Say you were asked to solve the initial value problem:

$$y' = x + 2y$$

$$y(0) = 0$$

numerically, finding a value for the solution at $x = 1$, and using steps of size $h = 0.25$.

Applying the Method

Clearly, the description of the problem implies that the interval we'll be finding a solution on is [0,1]. The differential equation given tells us the formula for $f(x, y)$ required by the Euler Method, namely:

$$f(x, y) = x + 2y$$

and the initial condition tells us the values of the coordinates of our starting point:

- $x_o = 0$
- $y_o = 0$

We now use the Euler method formulas to generate values for x_1 and y_1.

The x-iteration formula, with $n = 0$ gives us:

$$x_1 = x_o + h$$

or:

$$x_1 = 0 + 0.25$$

So:

$$x_1 = 0.25$$

And the y-iteration formula, with $n = 0$ gives us:

$$y_1 = y_o + h\, f(x_o, y_o)$$

or:

$$y_1 = y_o + h\,(x_o + 2y_o)$$

or:

$$y_1 = 0 + 0.25\,(0 + 2*0)$$

So:

$$y_1 = 0$$

Summarizing, the second point in our numerical solution is:

- $x_1 = 0.25$
- $y_1 = 0$

We now move on to get the next point in the solution, (x_2, y_2).

The x-iteration formula, with n=1 gives us:

$$x_2 = x_1 + h$$

or:

$$x_2 = 0.25 + 0.25$$

So:

$$x_2 = 0.5$$

And the *y*-iteration formula, with $n = 1$ gives us:

$$y_2 = y_1 + h\, f(x_1, y_1)$$

or:

$$y_2 = y_1 + h\, (x_1 + 2y_1)$$

or:

$$y_2 = 0 + 0.25\, (0.25 + 2*0)$$

So:

$$y_2 = 0.0625$$

Summarizing, the third point in our numerical solution is:

- $x_2 = 0.5$
- $y_2 = 0.0625$

We now move on to get the fourth point in the solution, (x_3, y_3).

The *x*-iteration formula, with $n = 2$ gives us:

$$x_3 = x_2 + h$$

or:

$$x_3 = 0.5 + 0.25$$

So:

$$x_3 = 0.75$$

And the y-iteration formula, with $n = 2$ gives us:

$$y_3 = y_2 + h\, f(x_2, y_2)$$

or:

$$y_3 = y_2 + h\,(x_2 + 2y_2)$$

or:

$$y_3 = 0.0625 + 0.25\,(0.5 + 2*0.0625)$$

So:

$$y_3 = 0.21875$$

Summarizing, the fourth point in our numerical solution is:

- $x_3 = 0.75$
- $y_3 = 0.21875$

We now move on to get the fifth point in the solution, (x_4, y_4).

The x-iteration formula, with $n = 3$ gives us:

$$x_4 = x_3 + h$$

or:

$$x_4 = 0.75 + 0.25$$

So:

$$x_4 = 1$$

And the y-iteration formula, with $n = 3$ gives us:

$$y_4 = y_3 + h\, f(x_3, y_3)$$

or:

$$y_4 = y_3 + h\,(x_3 + 2y_3)$$

or:

$$y_4 = 0.21875 + 0.25\,(0.75 + 2*0.21875)$$

So:

$$y_4 = 0.515625$$

Summarizing, the fourth point in our numerical solution is:

- $x_4 = 1$
- $y_4 = 0.515625$

We could summarize the **results** of all of our calculations in a tabular form, as follows:

n	x_n	y_n
0	0.00	0.000000
1	0.25	0.000000
2	0.50	0.062500
3	0.75	0.218750
4	1.00	0.515625

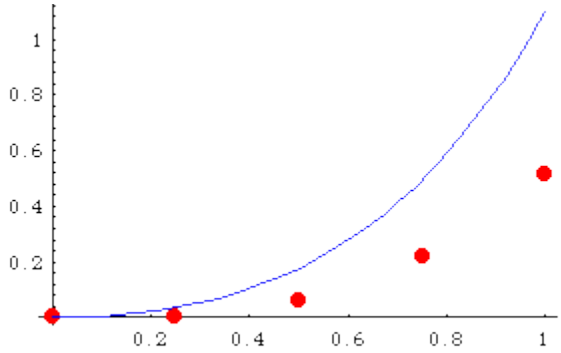
A question you should always ask yourself at this point of using a numerical method to solve a problem, is "How accurate is my solution?" Sadly, the answer is "Not very!" This problem can actually be solved without resorting to numerical methods (it's linear). The true solution turns out to be:

$$y = 0.25\,e^{2x} - 0.5\,x - 0.25$$

If we use this formula to generate a table similar to the one above, we can see just how poorly our numerical solution did:

x	*y*
0.00	0.000000
0.25	0.037180
0.50	0.179570
0.75	0.495422
1.00	1.097264

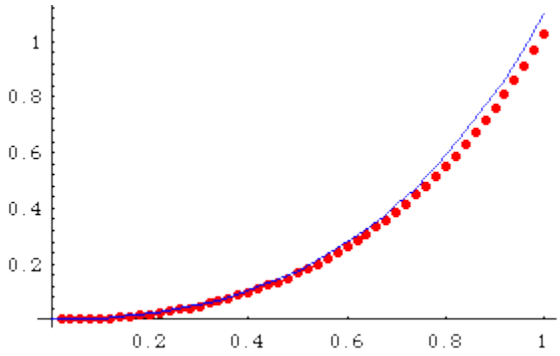
We can get an even better feel for the inaccuracy we have incurred if we compare the graphs of the numerical and true solutions, as shown here:



The numerical solution gets worse and worse as we move further to the right. We might even be prompted to ask the question "What good is a solution that is this bad?" The answer is "Very little good at all!" So should we quit using this method? No! **The reason our numerical solution is so inaccurate is because our step-size is so large.** To improve the solution, shrink the step-size!

By the way, the reason I used such a large step size when we went through this problem is because we were working it by hand. When we move on to using the computer to do the work, we needn't be so afraid of using tiny step-sizes.

To illustrate that Euler's Method isn't always this terribly bad, look at the following picture, made for exactly the same problem, only using a step size of $h = 0.02$:



As you can see, the accuracy of this numerical solution is much higher than before, *but so is the amount of work needed!* Look at all those red points! Can you imagine calculating the coordinates of each of them by hand?

Heun's Method

Theoretical Introduction

In the last lab you learned to use Euler's Method to generate a numerical solution to an initial value problem of the form:

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

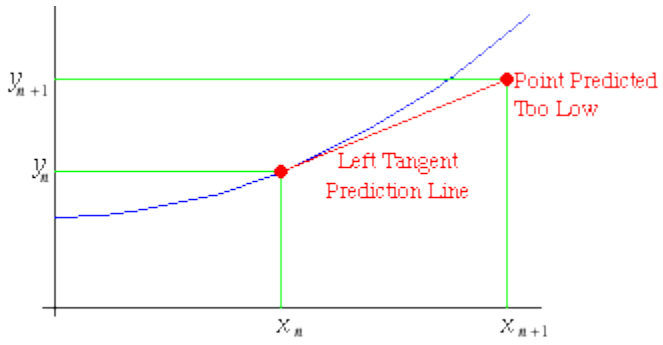
$$y(x_0) = y_0$$

Now it's time for a confession: In the real-world of using computers to derive numerical solutions to differential equations, no-one actually uses Euler's Method. Its shortcomings, discussed in detail in the last lab, nameley its inaccuracy and its slowness, are just too great. Though it is of some historical interest, the primary reason we bother to discuss it at all is that while it is a fairly simple algorithm to understand, it still enables us to start thinking more clearly about algorithmic approaches to the problem in general. We will now develop a better method than Euler's for numerically solving this same kind of initial value problem, but we'll use Euler's method as a foundation. For this reason Heun's Method is sometimes referred to as the Improved Euler Method.

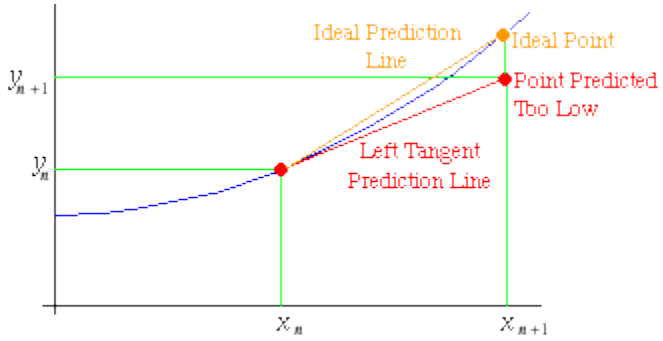
So where should we begin? Let's start by asking what it is about Euler's Method that makes it so poor at its job. Recall that the basic idea is to use the tangent line to the actual solution curve as an estimate of the curve itself, with the thought that provided we don't project too far along the tangent line on a particular step, these two won't drift too far apart. In reality, this turns out to be asking too much. Even when extremely small step sizes are used, over a large number of steps the error starts to

accumulate and the two solutions part company! We can even go so far as to theoretically predict what kind of error the method will introduce.

Where the actual solution curve is **concave-up**, its tangent line will **underestimate**the vertical coordinate of the next point, as seen in the image below.



Ideally, we would like our "prediction line" to **hit** the solution curve right at its next predicted point, as shown below.

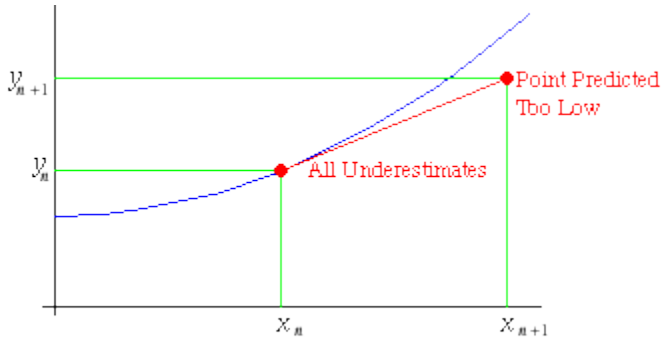


(Of course, where the actual solution curve is **concave-down**, the situation is reversed, and the tangent line will **overestimate** the vertical coordinate of the next point.)

However, we should remind ourselves that our discussion is purely theoretical. We do **not** really know the actual solution to the differential equation we are solving. If we did, then why would we bother trying to find a numerical solution? So in reality there is no quick way for us to know at any given stage of our numerical solution derivation whether or not the actual solution curve is concave-up or concave-down, and hence, there is no quick way to know whether or not our next calculated point is an over- or under-estimate. And remember, we don't even have any guarantee that the concavity of the curve remains consistent. It may actually change from concave-up to concave-down at some point within the domain of our desired solution.

Now let's think about how we might begin to fix this problem of the actual solution curve and the numerical solution drifting apart. Can we beat the "concavity problem?" Somehow we need to correct the over- or under-estimate problem that Euler's Method inevitably encounters. The Heun algorithm cleverly addresses this correction requirement. Let's take our concave-up example from above, and consider it more carefully this time.

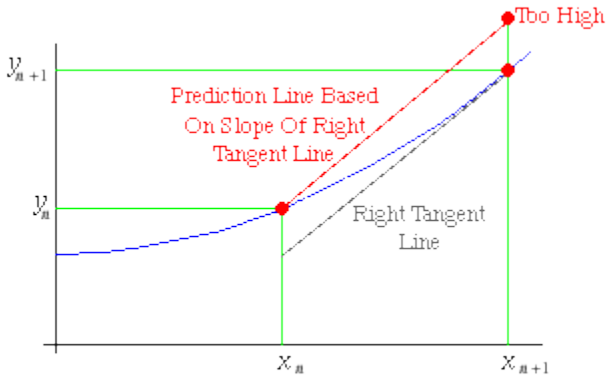
Instead of focussing on the initial, left end-point where we formed our tangent line, consider the interval spanned by the tangent line segment as a whole. As we've already noted, the tangent line from the left underestimates the slope of the curve for the entire width of the interval from the current point to the next predicted point, and as a result of this, the points along the tangent line have vertical coordinates which are all underestimates of those of the points lying along the actual solution curve, including the all-important right end-point of the interval under consideration. (After all, it is this point which forms the next component of our evolving numerical solution.)



To overcome this deficiency we would need to have used a line with a **greater slope** in order to more accurately predict the coordinates of the next point in our numerical solution. But how much steeper should our "prediction line" be made? Some fixed percentage? Some absolute amount? Well, no, that wouldn't make sense, as differing solutions have differing curvatures, and hence would require differing correction factors. No, if we're to make our "prediction line" steeper we need to do it in some logical way which takes into account, at least to some extent, the actual shape of the solution curve.

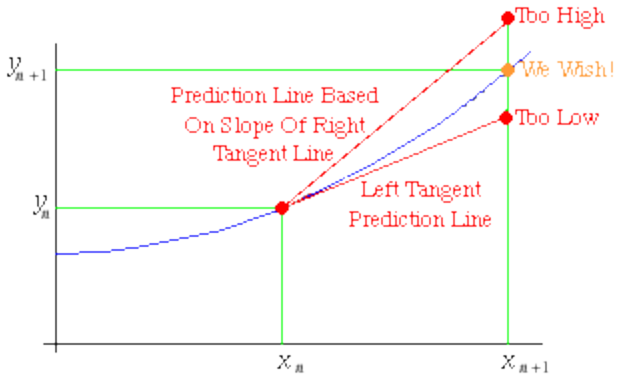
Heun's clever approach to this problem is to consider the tangent lines to the solution curve at **both ends** of the interval we are investigating. As we have already established, the tangent line to the curve at the **left** end-point of the interval is **not steep enough** for accurate predictions. However if we consider the tangent line to the curve at the **right** end-point, *(assuming we can find it somehow—more on this later,)* it has the opposite problem. Look at the plot below. It shows the right hand

tangent line, but since we wish to predict the next point by extrapolating from the known point we already have, i.e. the left end-point, we need to construct a prediction line based on the right tangent line's *slope alone*. We don't use the right tangent line itself to make our prediction, since, in a sense, it's "already there" at the right end of the interval which we are spending so much time trying to predict. So we create a line passing **through** our **known point** at the **left** end of the interval, and we give it the **slope** of the **tangent line** passing through the **right endpoint**. (You may need to read that sentence again!)

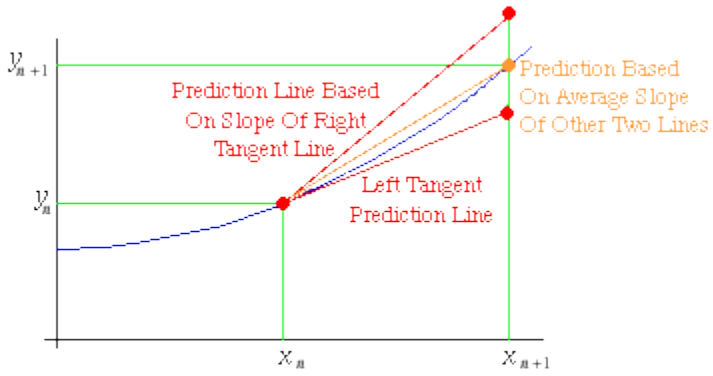


Obviously its slope is **too steep** to be used as the slope of our "ideal prediction line," and results in an **overestimate** if it is used. A few moments thought should confirm for you that this situation will always be true for any interval over which the solution curve is concave up.

Now look at the relationship between the errors made by both our left and right tangent line predictions. One is **underestimating** the *y*-coordinate of the next point, and the other is **overestimating** it.



The point we really want, i.e. the "ideal point" discussed earlier, appears to lie approximately halfway between these erroneous estimates based on tangent lines. In order to get this "ideal point" we still need to ride along an "ideal prediction line," but what should we use as this line's slope? If you've really been following the foregoing discussion you can probably guess by now what slope we'll use. We will take the **average** of the slopes of the left and right tangent lines that we've spent so much time discussing already. (After all, a line with a slope which is the average slope of a couple of lines predicting both too high and too low respectively, should be closer to the correct value than either of them!)



So that's the basic idea behind Heun's method—using a prediction line whose slope is the average of the slopes of the tangent lines at either end of the interval.

Now we have to tackle a problem that I hinted at earlier: We don't actually *know* the right end-point of the interval, and therefore there is no way to know the slope of the tangent line at that point! (If we *did* know the right end-point's coordinates then we

wouldn't be wasting our time trying to find them with this algorithm!) Recall that in the last lab we learned how to use Euler's method to find the *approximate* location of the next point along a solution curve? That's what we'll do with Heun's method! We'll use Euler's method to roughly estimate the coordinates of the next point in the solution, and once we have this information, we'll re-predict (or correct) our original estimate of the location of the next solution point by using the method of averaging the slopes of the left and right tangent lines that we so carefully developed above.

(As a side note, numerical analysts would refer to **Euler**'s method as a **predictor algorithm**, whereas **Heun**'s method is described as a **predictor-corrector algorithm**. Can you see why they would use this kind of nomenclature?)

OK, since we've managed to conceptualize the Heun algorithm graphically, it's now time to develop the method symbolically so that we can more easily see how to write the *Mathematica* code for it.

We already have our iterative formulas from the previous lab on Euler's method to get the ball rolling. These were:

$$x_{n+1} = x_n + h$$

$$y_{n+1} = y_n + h\, f(x_n, y_n)$$

where $f(x, y)$ was the right-hand side of the original differential equation, and h is the width of the subdivisions we are planning to use. Also, (x_n, y_n) represented the known (left) point, and (x_{n+1}, y_{n+1}) represented the new (right) point. Remember, the reason we *care* about these formulas for our new Heun method is that we'll be using Euler's method to make a rough prediction of the location of the predicted next point so that these coordinates may be used for our estimate of the slope of the tangent line at the right end of the interval in question.

So let's get those **tangent line slopes** figured out formulaically.

The **left end-point** is simply the current point, (x_n, y_n), so the **slope of the tangent line at the left end-point**, just as with Euler's method, is the right hand side of the original differential equation evaluated at this point:

$$slope_{left} = f(x_n, y_n)$$

Now for that **right end-point** that we've been worrying about so much. As we said just a few seconds ago, Euler gives a rough prediction of its location as being at **coordinates**:

$$(x_{n+1}, y_{n+1}) = (x_n + h, y_n + h\, f(x_n, y_n))$$

Recalling that one of our fundamental assumptions, based on our interpretation of the original differential equation, is that the quantity $f(x, y)$ on the right hand side of the equation can be thought of as the slope of the solution we seek at any point (x, y), we may now combine this idea with the rough, Euler estimate of the next point to give us an estimate of the **slope of the tangent line at the right end-point**. This would be:

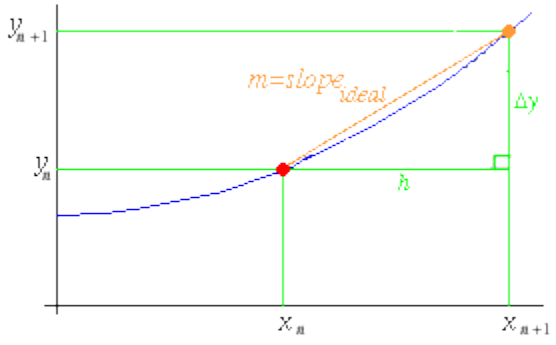
$$slope_{right} = f(x_n + h, y_n + h\, f(x_n, y_n))$$

Next, recall from the discussion and the graphs we considered above that the slope of our "ideal prediction line" is the *average* of the left and right tangent slopes whose formulas we have just found. In other words:

$$slope_{ideal} = (1/2)\,(slope_{left} + slope_{right})$$

All that is left to do now is to utilize this slope in creating the "ideal prediction line" itself, and then use this "ideal prediction line" to actually find the coordinates of the right hand end-point. To a great extent the remainder of the work is simply a repetition of what we saw as we developed the formulas for Euler's method in the last lab.

Consider the following illustration. We wish to predict the right hand end-point's coordinates. In order to get them it would be sufficient to find the value of Δy.



Using the elementary idea that ***slope = rise / run***, we derive the following formula immediately from the picture:

$$slope_{ideal} = \Delta y / h$$

which is easily rearranged to give us a formula for Δy, namely:

$$\Delta y = h\ slope_{ideal}$$

Finally, then, we can predict the coordinates of the **next point** in our numerical solution. The next ***x*-coordinate** is simply the current x-coordinate plus the step size, h. The next ***y*-coordinate** is the current y-coordinate plus Δy. Formulaically, this would be:

$$x_{n+1} = x_n + h$$

and

$$y_{n+1} = y_n + \Delta y$$

Replacing Δy by the value we just found for it above, this becomes:

$$y_{n+1} = y_n + h\ slope_{ideal}$$

And replacing $slope_{ideal}$ by the average of the left and right tangent slopes found earlier, this is transformed into:

$$y_{n+1} = y_n + (1/2)\ h\ (slope_{left} + slope_{right})$$

One last substitution, and our formula is complete. We recently found that:

$$slope_{left} = f(x_n, y_n)$$

and that

$$slope_{right} = f(x_n + h, y_n + h\ f(x_n, y_n))$$

so if we substitute these values into the above equation for y_{n+1} we get:

$$y_{n+1} = y_n + (1/2)\ h\ (f(x_n, y_n) + f(x_n + h, y_n + h\ f(x_n, y_n)))$$

or, cleaning things up slightly:

$$y_{n+1} = y_n + (h/2)\ (f(x_n, y_n) + f(x_n + h, y_n + h\ f(x_n, y_n)))$$

At last! We're done with our theory for Heun's method. Summarizing the results, the iteration formulas for Heun's method are:

$$x_{n+1} = x_n + h$$

$$y_{n+1} = y_n + (h/2)\,(f(x_n, y_n) + f(x_n + h, y_n + h\, f(x_n, y_n)))$$

The Runge-Kutta Method

Theoretical Introduction

In the last lab you learned to use Heuns's Method to generate a numerical solution to an initial value problem of the form:

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

$$y(x_0) = y_0$$

We discussed the fact that Heun's Method was an improvement over the rather simple Euler Method, and that though it uses Euler's method as a basis, it goes beyond it, attempting to compensate for the Euler Method's failure to take the curvature of the solution curve into account. Heun's Method is one of the simplest of a class of methods called **predictor-corrector algorithms**. In this lab we will address one of the most powerful predictor-corrector algorithms of all—one which is so accurate, that most computer packages designed to find numerical solutions for differential equations will use it by default—the **fourth order Runge-Kutta Method**. (For simplicity of language we will refer to the method as simply the *Runge-Kutta Method* in this lab, but you should be aware that Runge-Kutta methods are actually a general class of algorithms, the fourth order method being the most popular.)

The Runge-Kutta algorithm may be very crudely described as "Heun's Method on steroids." It takes to extremes the idea of correcting the predicted value of the next solution point in the numerical solution. *(It should be noted here that the actual, formal derivation of the Runge-Kutta Method will **not** be covered in this course. The calculations involved are complicated, and rightly belong in a more advanced course in differential equations, or numerical methods.)* Without further ado, using the same notation as in the previous two labs, here is a summary of the method:

$$x_{n+1} = x_n + h$$

$$y_{n+1} = y_n + (1/6)(k_1 + 2k_2 + 2k_3 + k_4)$$

where

$$k_1 = h\, f(x_n, y_n)$$

$$k_2 = h\, f(x_n + h/2, y_n + k_1/2)$$

$$k_3 = h\, f(x_n + h/2, y_n + k_2/2)$$

$$k_4 = h\, f(x_n + h, y_n + k_3)$$

Even though we do not plan on deriving these formulas formally, it is valuable to understand the geometric reasoning that supports them. Let's briefly discuss the components in the algorithm above.

First we note that, just as with the previous two methods, the Runge-Kutta method iterates the x-values by simply adding a fixed step-size of h at each iteration.

The y-iteration formula is far more interesting. It is a weighted average of four values—k_1, k_2, k_3, and k_4. Visualize distributing the factor of 1/6 from the front of the sum. Doing this we see that k_1 and k_4 are given a weight of 1/6 in the weighted average, whereas k_2 and k_3 are weighted 1/3, or twice as heavily as k_1 and k_4. (As usual with a weighted average, the sum of the weights 1/6, 1/3, 1/3 and 1/6 is 1.) So what are these k_i values that are being used in the weighted average?

k_1 you may recognize, as we've used this same quantity on both of the previous labs. This quantity, $h\, f(x_n, y_n)$, is simply Euler's prediction for what we've previously called Δy—the vertical jump from the current point to the next Euler-predicted point along the numerical solution.

k_2 we have never seen before. Notice the x-value at which it is evaluating the function f. $x_n + h/2$ lies halfway across the prediction interval. What about the y-value that is coupled with it? $y_n + k_1/2$ is the current y-value plus half of the Euler-predicted Δy that we just discussed as being the meaning of k_1. So this too is a halfway value, this time vertically halfway up from the current point to the Euler-predicted next point. To summarize, then, the function f is being evaluated at a point that lies halfway between the current point and the Euler-predicted next point. Recalling that the function f gives us the *slope* of the solution curve, we can see that evaluating it at the halfway point just described, i.e. $f(x_n + h/2, y_n + k_1/2)$, gives us an estimate of the slope of the solution curve at this halfway point. Multiplying this slope by h, just as with the Euler Method before, produces a prediction of the y-jump made by the actual solution across the whole width of the interval, only this time the predicted jump is

not based on the slope of the solution at the left end of the interval, but on the estimated slope halfway to the Euler-predicted next point. Whew! Maybe that could use a second reading for it to sink in!

k_3 has a formula which is quite similar to that of k_2, except that where k_1 used to be, there is now a k_2. Essentially, the f-value here is yet another estimate of the slope of the solution at the "midpoint" of the prediction interval. This time, however, the y-value of the midpoint is not based on Euler's prediction, but on the y-jump predicted already with k_2. Once again, this slope-estimate is multiplied by h, giving us yet another estimate of the y-jump made by the actual solution across the whole width of the interval.

k_4 evaluates f at $x_n + h$, which is at the extreme right of the prediction interval. The y-value coupled with this, $y_n + k_3$, is an estimate of the y-value at the right end of the interval, based on the y-jump just predicted by k_3. The f-value thus found is once again multiplied by h, just as with the three previous k_i, giving us a final estimate of the y-jump made by the actual solution across the whole width of the interval.

In summary, then, each of the k_i gives us an estimate of the size of the y-jump made by the actual solution across the whole width of the interval. The first one uses Euler's Method, the next two use estimates of the slope of the solution at the midpoint, and the last one uses an estimate of the slope at the right end-point. Each k_i uses the earlier k_i as a basis for its prediction of the y-jump.

This means that the Runge-Kutta formula for y_{n+1}, namely:

$$y_{n+1} = y_n + (1/6)(k_1 + 2k_2 + 2k_3 + k_4)$$

is simply the y-value of the current point plus a weighted average of four different y-jump estimates for the interval, with the estimates based on the slope at the midpoint being weighted twice as heavily as the those using the slope at the end-points.

As we have just seen, the Runge-Kutta algorithm is a little hard to follow even when one only considers it from a geometric point of view. In reality the formula was not originally derived in this fashion, but with a purely analytical approach. After all, among other things, our geometric "explanation" doesn't even account for the weights that were used. If you're feeling ambitious, a little research through a decent mathematics library should yield a detailed analysis of the derivation of the method.