Math 206, Spring 2007, Computer Lab 1
(Revised 4 February 2007)

1. Introduction

This lab introduces you to the mathematical software package Mathematica. The goals are modest; I just want you to see that it can help with common tasks in differential geometry, such as

- visualizing curves and surfaces in $\mathbb{R}^3$,
- automating tedious algebraic computations, and
- making numerical approximations when necessary.

Later labs will build on the skills here, and you are also welcome to use Mathematica on your other assignments. Although no mathematician should rely on computers, no mathematician should forgo them entirely, either.

Mathematica is accessible in all public computer labs on campus, such as the one on the bottom floor of the Teer building (where the math, physics, and engineering library is). This lab is designed for use with version 5.1 or 5.2 of the software. You're welcome to do these labs in some other, comparable system, such as Maple, if you clear it with me well ahead of time.

2. Using the Software

Unless you are already quite familiar with Mathematica, I recommend that you type in all of the examples in this lab. You should also play around, trying variations of your own.

When you start up Mathematica, you are presented with a *notebook*. This is a place to enter computations, see the results, type comments or explanations, construct larger programs in pieces, etc. A notebook consists of a sequence of *cells*. Each cell has a definite purpose: input, output, explanatory text, etc. First, try typing

```
    2 + 3
```

and hitting Enter (or Shift-Enter, on some systems). Mathematica reformats this entry as an input cell, passes the input to its *kernel* (computational engine), receives an answer back from the kernel, and displays the answer in an output cell. Interactive programs like this are commonly called *interpreters* or *read-eval-print loops*. At this point, your notebook should show an input cell and its subsequent output cell:

```
In[1]:= 2 + 3
Out[1]= 5
```

Here's another command to try:

```
    D[t^3 + 2 t + 1, t]
```

It produces

```
In[2]:= D[t^3 + 2 t + 1, t]
Out[2]= 2 + 3 t^2
```

As I'm sure you've figured out, the `D` function takes in two arguments — a function and a variable — and symbolically differentiates the function with respect to the variable. If you like, you can store the answer in a variable of your choosing. Try this:

```
    myfunc = D[t^3 + 2 t + 1, t]
```

With the answer stored in a variable, you can subsequently use it whenever and however you want. If you just want to see the answer again, enter

```
    myfunc
```

If you want to antidifferentiate it, enter

```
    Integrate[myfunc, t]
```

All of Mathematica's built-in functions and constants have names beginning with upper-case letters, as in `D`, `Integrate`, `Pi`, `ParametricPlot`, etc. When naming your own variables, always begin with lower-case letters, as in `myfunc`, `a`, `curvatureNormal`, to avoid confusion.

Of course, you can save your Mathematica notebook to a file on your computer whenever you want, and then open it later and resume work. When you resume, your cells are not automatically run for you; you have to hit Enter (or Shift-Enter) in each cell to activate it. You can even have multiple notebooks open at the same time, activating commands from any of them as you like.

There's one key point to understand here: All notebooks connect to the same kernel. The notebooks are not isolated from each other. If you enter `a = 13` in one notebook, close that notebook, open up some other notebook and enter `a` in it, then Mathematica will report that `a` is `13`.

Storing values in variables is extremely helpful when building your own functions and programs (as we do later in this lab). Unfortunately, it can sometimes cause confusion. For example, enter the following two commands. By the way, you can enter them into one cell, if you like; Mathematica will deal with them in order.

```
    t = 3
    D[t^3 + 2 t + 1, t]
```

Mathematica reports an error. If $t = 3$, then $t^3+2t+1 = 34$, so it thinks that you're trying to differentiate the function 34. That in itself isn't a problem. The problem is that you're differentiating with respect to 3, since $t = 3$, and that doesn't make any sense. This is not a defect in Mathematica. It's just an example of a computer taking what you say literally.

This is an important point to remember, especially if you haven't done much programming before: Computers are entirely literal and not nearly as smart as you are. They are incapable of judgment. You have to tell them exactly what you want, as if talking to a four-year old (who just happens to know hundreds of math functions).

If Mathematica ever behaves strangely and you can't figure out what's wrong with it, there's a good chance that there are variables with set values that you've forgottten about. Try saving your work, quitting the program, restarting it, and running exactly the cells you want again. (Restarting is a heavy-handed solution, but I'm trying to keep things simple here.)

3. The Mathematica Language

So you know how to use the Mathematica interpreter and user interface. Now let's talk a little more about the language itself.

The good news is that Mathematica syntax is extremely simple. In fact, *everything* in Mathematica can be written in the form $f[x_1, x_2, \dots, x_n]$, where f is some function and the x_i are inputs to it. To a math student this should look reasonable. (The square brackets [] are used instead of parentheses () because parentheses are used for too many other things.) For example, `2 + 3` is really represented as `Plus[2, 3]` inside Mathematica; the former is just a handy formatting convention to help the human user.

Lists occur frequently in Mathematica; for instance, they're used to represent row vectors. A list can be entered using curly braces, like this list of my five favorite numbers:

```
faves = {0, 1, I, E, Pi}
```

The curly braces, like the `+` symbol above, are really a formatting shortcut for humans. To see what that list really looks like to Mathematica, enter

```
FullForm[faves]
```

It tells you that lists are really constructed using the `List` function, and that complex numbers are constructed using the `Complex` function. Notice that these functions are nested inside one another — in other words, they are composed, just as functions in math are composed.

To obtain numerical approximations, use the `N` function:

```
N[Pi]
N[faves]
```

Notice that `N` is happy to take in a single number or a list of numbers. Many Mathematica functions automatically handle lists like this. Here is how you pick off the first element in the list:

```
faves[[1]]
```

What is the result of the following command?

```
(faves[[3]])^2
```

In math, matrices are essentially vectors of vectors (all of the same dimension). Mathematica knows this. Try

```
a = {{1, 2, 0}, {-1, 2, 1}, {1, 1, 1}}
MatrixForm[a]
Det[a]
a.a
```

Now let's do something more interesting — namely, have Mathematica generate the formula for a 2×2 determinant with entries a, b, c, d, regarded as variables. There is a problem here, since we've already set the variable `a` to a particular value. To fix that, clear the value in `a` using the command

```
Clear[a]
```

Then enter

```
    matrix = {{a, b}, {c, d}}
    Det[matrix]
```

You should get $ad - bc$, which is the formula for the determinant. At this point, if you enter

```
    c = 1
    Det[matrix]
```

then Mathematica will output $ad - b$. It's using the known value for c, but it's manipulating the other variables formally, since it doesn't have values for them.

It's important to understand the distinction between Mathematica variables and mathematical variables. The former are used to store values (this happens in all programming languages) whereas the latter exist precisely to avoid specifying a particular value. If anything, Mathematica variables are really more like mathematical constants, although you're allowed to redefine these constants whenever you want.

To get an idea of why we care about such nitpicking formalities, try

```
    func = t^3 + 2 t + 1
    func[2]
```

I'm trying to evaluate the function $t^3 + 2t + 1$ on the value 2. It seems reasonable, right? But Mathematica doesn't understand what I mean, because it regards `t^3 + 2 t + 1` as a formula — a string of symbols to be manipulated formally — and unlike humans it doesn't automatically recognize that the formula represents a function — a sequence of operations to be performed on numbers. If you want, you can work around this by entering

```
    func /. t -> 2
```

This says, "Give me the formula `func`, but with all occurrences of `t` replaced with `2`." Mathematica replaces `t` with `2` in `func` and automatically simplifies the resulting formula, down to the very simple formula `13`.

To define your own function in Mathematica, try this instead:

```
    betterfunc[t_] := t^3 + 2 t + 1
    betterfunc[2]
```

The underscore `_` on the left-hand side is highly significant. It tells Mathematica that you are defining a *pattern.* After you establish this pattern, whenever Mathematica encounters an expression of the form `betterfunc[...]` it will automatically replace it with `t^3 + 2 t + 1` and then replace all occurrences of `t` with whatever `...` is (using the same `/.`, `->` mechanism as above, in fact). When you define a function in Mathematica, you're really defining a pattern for formula manipulation. Deep in its guts, all Mathematica ever does is manipulate formulas and simplify them for you.

If you want to learn more about Mathematica, you can read references and tutorials on the web by searching for "Mathematica documentation". Also, you can look up any function in Mathematica's Help menu. Our goal here isn't to master the software, but just to use it for differential geometry.

4. Visualizing Curves

In this section we draw some curves. Try entering

```
ParametricPlot[{Cos[t], Sin[t]}, {t, 0, 2 Pi}]
```

This plots the $\mathbb{R}^2$-valued function $\vec{\alpha}(t) = (\cos t, \sin t)$ for values of t from 0 to 2π. If you click on the plot, then a rectangle should appear around it. You can make the plot larger or smaller by dragging the corners of the rectangle.

If your Mathematica is like mine, it inexplicably plots with different scales on x and y axes, so that the trace looks like a non-circular ellipse (it should be a circle). To fix it, try this:

```
ParametricPlot[{Cos[t], Sin[t]}, {t, 0, 2  Pi}, AspectRatio -> 1]
```

That extra bit at the end is called an *option* in Mathematica; it's an optional argument to the `ParametricPlot` function that lets you control the relationship between the x and y scales. Many functions can be used in various ways with various arguments and options; you can look up any function in the Help menu.

For our next example, try the helix $\vec{\alpha}(t) = (\cos t, \sin t, t)$:

```
ParametricPlot3D[{Cos[t], Sin[t], t}, {t, 0, 4 Pi}]
```

To view it from a different angle, try the `ViewPoint` option:

```
ParametricPlot3D[{Cos[t], Sin[t], t}, {t, 0, 4 Pi}, ViewPoint->{2, 3, -2}]
```

A more interesting curve, that is more difficult to draw by hand, is the twisted cubic $\vec{\alpha}(t) = (t^3, t^2, t)$:

```
ParametricPlot3D[{t^3, t^2, t}, {t, -1, 1}, ViewPoint->{2, 2, 0}]
```

You can change the viewpoint as before, but unfortunately the picture is always static. I wish that Mathematica's 3D drawing facilities were more interactive, so that you could rotate the 3D picture by dragging it with your mouse, for example. The good news is that there are several free software packages available for this; if you're using Mathematica on your own computer, you can search the web for "Mathematica rotate 3D", download one or two to your computer, and try them out. To my knowledge, Duke's computers don't have any of them.

So instead we'll settle for animating the 3D image in a non-interactive way. Enter the following commands, all in one cell if you like — they will be executed in order. Make sure to get the quotation marks right; there are double-quotes `"` and left-quotes `‘` (the latter is probably above the Tab key on your keyboard).

```
Needs["Graphics‘Animation‘"]
plot = ParametricPlot3D[{t^3, t^2, t}, {t, -1, 1}];
SpinShow[plot]
```

The first command loads a special library or *package* for animating graphics. The second command stores our plot into a variable `plot` (yes, even graphics can be stored in variables). The semicolon at the end means that although you want the command to be carried out, you don't want Mathematica to show you the output. The third command creates more plots by spinning the first one around in 3D. You can look at the rotated plots one after another, or you can animate them, as follows.

Look on the right-hand side of your notebook. There should be a bunch of long, square brackets that indicate how all of the cells in your notebook are grouped together. For example, input cells are usually grouped with their corresponding output cells. You can select one or more cells by dragging the mouse along their brackets (with the button held down).

Right now, select all of the cells that contain the rotated plots generated by `SpinShow`. Then go to the Cell menu and choose Animate Selected Graphics. This produces an animation in the spot where the first of the rotated plots used to be. To stop the animation, just click on it.

You can read about other Mathematica animation commands by searching the web for "Mathematica animation". They all follow this same system of "generate a list of images, select them, then do Cell: Animate Selected Graphics to animate them".

By the way, using packages such as `Graphics‘Animation‘` can sometimes lead to bizarre variable conflicts. You must activate `Needs["Graphics‘Animation‘"]` before you activate a command from it, such as `SpinShow`. If you accidentally activate `SpinShow` first, then Mathematica thinks you want to define your own variable called `SpinShow`, and then it won't even load `Graphics‘Animation‘` properly after that, so you have to restart Mathematica and try again.

5. Symbolic Computation

In this section we use Mathematica for symbolic computation. Let's begin by entering some handy functions for dealing with curves.

```
normSquared[vector_] := Dot[vector, vector];
norm[vector_] := Sqrt[normSquared[vector]];
ddt[curve_] := D[curve, t];
speed[curve_] := norm[ddt[curve]];
tangent[curve_] := (1 / speed[curve]) * ddt[curve];
curvatureNormal[curve_] := (1 / (normSquared[ddt[curve]])^2) *
    Cross[Cross[ddt[curve], ddt[ddt[curve]]], ddt[curve]];
curvature[curve_] := norm[curvatureNormal[curve]];
normal[curve_] := (1 / curvature[curve]) * curvatureNormal[curve];
binormal[curve_] := Cross[tangent[curve], normal[curve]];
torsion[curve_] := -(1 / normSquared[Cross[ddt[curve], ddt[ddt[curve]]]])
    * Det[{ddt[curve], ddt[ddt[curve]], ddt[ddt[ddt[curve]]]}];
```

All I'm doing here is translating standard math formulas into Mathematica syntax. For example, the first function takes in a vector and dots it with itself, producing the square of the norm of the vector. The second function gives the norm of the vector. There's actually a built-in `Norm` function — which is why Mathematica warns you that the name `norm` is similar to a preexisting one — but `Norm` uses a lot of unnecessary absolute value functions, so I prefer this one. The other functions are identical or similar to ones in your book or homework. Make sure you understand how each is defined; it's basically a matter of being very careful about parentheses () and brackets [], and using built-in functions like `Cross` that you can look up in the Help menu. Notice that these formulas do not assume that `curve` is parametrized by arc length.

Now, in another cell, enter

```
alpha[t_] := {3 Cos[t], 3 Sin[t], 0};
Simplify[tangent[alpha[t]]]
Simplify[normal[alpha[t]]]
Simplify[binormal[alpha[t]]]
Simplify[speed[alpha[t]]]
Simplify[curvature[alpha[t]]]
Simplify[torsion[alpha[t]]]
```

The `Simplify` function used here does a pretty good job of simplifying complicated algebra. You should recognize this curve $\vec{\alpha}(t)$. Verify all of the output to make sure that it's right.

Now let's graph the curvature and torsion of the twisted cubic.

```
alpha[t_] := {t^3, t^2, t};
k = curvature[alpha[t]];
ParametricPlot[{t, k}, {t, -2, 2}]
tau = torsion[alpha[t]];
ParametricPlot[{t, tau}, {t, -2, 2}]
```

Finally, let's use the `NIntegrate` function to numerically approximate the arc length of the graph of the sine function:

```
alpha[t_] := {t, Sin[t], 0};
v = speed[alpha[t]];
arcLength = NIntegrate[v, {t, 0, 2 Pi}]
```

Having a computer crank through tedious arithmetic and algebra for you is wonderful. On the other hand, the computer doesn't show you how it arrives at its answer, and the answer might not be in the form most convenient for your purposes. (Mathematica actually provides methods for customizing its symbolic algorithms, but they're way beyond the scope of this lab.) Furthermore, doing a calculation by hand often helps you understand and remember whatever it is that you're studying. I urge you to exercise judgment in using this tool; use a computer for calculations only if you're sure that you could do them yourself.

6. Numerical Approximation

In this section we'll explore the fundamental theorem of the local theory of curves, which says that for any prescribed curvature $k(s)$ and torsion $\tau(s)$ there's a curve $\vec{x}(s) = (x_1(s), x_2(s), x_3(s))$ with that curvature and torsion. (Do Carmo denotes the curve $\vec{\alpha}(s)$, but let's use $\vec{x}(s)$ instead, to save typing.)

Recall that this boils down to solving the Frenet equations

$$\begin{aligned}
t_1'(s) &= k(s)n_1(s),\\
t_2'(s) &= k(s)n_2(s),\\
t_3'(s) &= k(s)n_3(s),\\
n_1'(s) &= -k(s)t_1(s) - \tau(s)b_1(s),\\
n_2'(s) &= -k(s)t_2(s) - \tau(s)b_2(s),\\
n_3'(s) &= -k(s)t_3(s) - \tau(s)b_3(s),\\
b_1'(s) &= \tau(s)n_1(s),\\
b_2'(s) &= \tau(s)n_2(s),\\
b_3'(s) &= \tau(s)n_3(s),
\end{aligned}$$

subject to the desired initial conditions. Let's use the initial conditions

$$\begin{aligned}
\vec{x}(0) &= (0,0,0),\\
\vec{t}(0) &= (1,0,0),\\
\vec{n}(0) &= (0,1,0),\\
\vec{b}(0) &= (0,0,1).
\end{aligned}$$

Let's also throw in these three equations, so that the relevance to $\vec{x}$ is clear:

$$\begin{aligned}
x_1'(s) &= t_1(s),\\
x_2'(s) &= t_2(s),\\
x_3'(s) &= t_3(s).
\end{aligned}$$

The solution is always possible in theory but often difficult in practice. This sort of problem is ripe for numerical approximation. So let's translate the problem into Mathematica notation. For the domain of $\vec{x}(s)$ we'll take the open interval from `start` to `end`. Now type all of the following code into a single cell — but don't run it yet.

```
Needs["Graphics`Animation`"]
Clear[x1, x2, x3,
     t1, t2, t3,
     n1, n2, n3,
     b1, b2, b3,
     k, tau, s, start, end]
k[s] = ...;
tau[s] = ...;
start = ...;
end = ...;
funcs = {x1[s], x2[s], x3[s],
     t1[s], t2[s], t3[s],
     n1[s], n2[s], n3[s],
     b1[s], b2[s], b3[s]}
eqns = {x1'[s] == t1[s], x2'[s] == t2[s], x3'[s] == t3[s],
     t1'[s] == k[s] n1[s],
     t2'[s] == k[s] n2[s],
     t3'[s] == k[s] n3[s],
     n1'[s] == -k[s] t1[s] - tau[s] b1[s],
     n2'[s] == -k[s] t2[s] - tau[s] b2[s],
     n3'[s] == -k[s] t3[s] - tau[s] b3[s],
     b1'[s] == tau[s] n1[s],
     b2'[s] == tau[s] n2[s],
     b3'[s] == tau[s] n3[s]}
initconds = {x1[0] == 0, x2[0] == 0, x3[0] == 0,
     t1[0] == 1, t2[0] == 0, t3[0] == 0,
     n1[0] == 0, n2[0] == 1, n3[0] == 0,
     b1[0] == 0, b2[0] == 0, b3[0] == 1}
solns = NDSolve[Join[eqns, initconds], funcs, {s, start, end}]
alpha = {x1[s], x2[s], x3[s]} /. solns[[1]]
plot = ParametricPlot3D[alpha, {s, start, end}];
SpinShow[plot]
```

Let's take a moment to get an idea of what this program does. It begins by clearing out all of the relevant variable names, just to be safe. Then there's a little section for specifying `k[s]`, `tau[s]`, `start`, and `end`; you're going to customize those items in a moment, before you actually run the thing. Then there are the names of the functions for which we're solving in the ODE. Notice that $k(s)$ and $\tau(s)$ are not among them, because we're specifying these, not solving for them. Then comes the system of equations, followed by the initial conditions.

In the next line, we join the equations and initial conditions together into one list of equations, since that's how Mathematica wants them. We invoke Mathematica's numerical differential equation solver `NDSolve`, telling it all of the equations, all of the functions to find, and the independent variable and its interval.

As you will see in a moment (but not yet, because we haven't specified $k(s)$, etc.), the output from the `NDSolve` command is a bit cryptic. It's a list consisting of one item. That one item is a list of functions that numerically approximate the solutions of the ODE, implemented in Mathematica code. Since they're ugly and complicated, Mathematica won't bother to show them to you, but we don't care. All we want to do is pick off `solutions[[1]]`, which is the list of functions, and then use those functions to construct a curve. That's what we do in the third-to-last line. We make a vector-valued function consisting of `x1`, `x2`, and `x3`, but replacing (using the syntax `/.` mentioned above) each of those functions with its solution function from `NDSolve`. We call this new curve `alpha`, in honor of do Carmo (and to avoid confusion with the `x1`, `x2`, `x3` named above).

In the second-to-last line, we plot the thing, using the same `start` and `end` as was used to construct the solution. We end with a semicolon to suppress the output; you may want to add more semicolons, earlier in the program, to suppress other outputs. I care only about the final output, which is an animatable sequence of pictures using our old friend `SpinShow`.

Now that we understand the program, we are ready to have some fun by choosing various values for `k[s]`, `tau[s]`, `start`, and `end` and seeing what kind of curves we get.

(1) First there's the classic `k[s] = 1`, `tau[s] = 0`, `start = -Pi`, `end = Pi`. If you enter these values into the program above and evaluate the cell, what do you get? By the way, what property is shared by all curves that have $\tau(s) \equiv 0$?
(2) Next try `k[s] = 1`, `tau[s] = 1`, `start = -10`, `end = 10`. Try changing k and τ to other constants, and see how that affects the shape.
(3) Try `k[s] = s`, `tau[s] = 1`, `start = -10`, `end = 10`. Try changing τ to $1/10$, k to $2s$, etc. Now we're getting into more interesting curves, and they really respond to changes in the parameters.
(4) Try `k[s] = s`, `tau[s] = Sin[s]`, `start = -10`, `end = 10`.
(5) Try `k[s] = ArcTan[s]`, `tau[s] = Sin[s]`, `start = -10`, `end = 10`.

Okay, you get the idea. By the way, don't forget to change `start` and `end`, too. By making them asymmetric you can get asymmetric pictures, even when your functions exhibit symmetry. Due to the way the program specifies initial conditions, 0 must always be in your interval, but you can always shift the interval by replacing s with $s + 5$, say, in $k(s)$ and $\tau(s)$.

7. Troubleshooting

If your code is not working, even though you've checked it a few times:

(1) Save your notebook, quit Mathematica, reopen your notebook, and run just the cells that you want. This resets all of your variables, so it often clears up problems. It's also possible that Mathematica has bugs that restarting mitigates; I can't vouch for that.
(2) If you are entering many lines of code in one cell, then don't. Run each line individually, in its own cell, so that you can more easily see which lines generate which error messages.

If these still don't help, then send me e-mail containing the following.

(1) The first error message that Mathematica generates — or, if it doesn't generate any error messages but you know that the results are wrong, then send me the results and an explanation of why they're wrong.
(2) All of the code that you've run up to the point where that first error message happens. This should be as little code as possible — just the stuff required for the current computation, that you've run since restarting Mathematica.

8. Assignment

Future labs will assume that you've done everything in this lab, making a serious attempt to understand how all of the code works. However, I don't want you to hand in every little thing we've done. Just hand in these wrap-up questions, separately from your other homework for the week. You can either print your code and plots or you can carefully hand-copy them.

8.1. **Local Invariants of Curves.** Let $\vec{\alpha}(t) = (t, \sin t, 0)$ and $\vec{\beta}(t) = \vec{\alpha}(t) + \vec{n}(t)$, where $\vec{n}$ is the unit normal vector to $\vec{\alpha}$. Using the `ParametricPlot3D` command and the `Show` command (which you can look up in the Help menu), plot $\vec{\alpha}$ and $\vec{\beta}$ together on one plot. If your Mathematica is just like mine, then the plot is not quite right, for an understandable reason. Hand in a hand-corrected plot, along with a graph of the curvature of $\vec{\alpha}$.

(By the way, plotting $\vec{\alpha}$ and $\vec{\alpha} + k\vec{n}$, instead of $\vec{\alpha}$ and $\vec{\alpha} + \vec{n}$, produces a pretty picture for this $\vec{\alpha}$.)

(By another way, doing this with $\vec{\alpha}(t) = (t, t \sin t, 0)$ produces an interesting $\vec{\beta}$ curve. Its curvature is so complicated that my Mathematica had trouble graphing it.)

8.2. **Motion Under Gravity.** Suppose that a comet (or planet) of mass m is traveling along a path $\vec{\alpha}(t)$ and comes near a star of mass M fixed at the origin. There is an attractive gravitational force $\vec{F}$ between them of magnitude

$$|\vec{F}| = \frac{GMm}{r^2},$$

according to Newton's law of gravitation, where G is a positive constant. Newton's second law of motion says that the force on the comet relates to its acceleration $\vec{\alpha}''$ by

$$\vec{F} = m\vec{\alpha}''.$$

(We assume that m is negligible compared to M, so that the star is unaccelerated by the force; this is why we can fix it at the origin.)

A. Prove, on paper, that the acceleration of the comet is

$$\vec{\alpha}''(t) = -\frac{GM}{|\vec{\alpha}(t)|^3}\vec{\alpha}(t).$$

B. Assume for simplicity that $GM = 2$, and plot the solution curve for the differential equation from Part A, with initial conditions

```
x1[0] == 1, x2[0] == 0, x3[0] == 0,
x1'[0] == 0, x2'[0] == 1, x3'[0] == 0
```

following the numerical differential equation solution example above. Notice that you'll have to use second derivatives, as in `x''[t] == ....` You should generate enough of the plot to convince yourself (and me) that it's an ellipse with the sun at one focus (as was known to Kepler in 1605, supposedly after he tried 40 other curves — geniuses work hard). Hand in a plot and your Mathematica code.

8.3. **Prescribing Curvature and Torsion.** Going back to our numerical solutions of the Frenet equations, find your own parameters `k[s]`, `tau[s]`, `start`, and `end` that produce an interesting curve — the more interesting, the better. Hand in a plot of your curve from a viewpoint that shows how exciting it is, along with the parameters that produced it.

8.4. **Time Spent.** Please tell me how many hours you spent on this lab altogether. The answer does not affect your grade.