Simulation

Source: http://www.myphysicslab.com/runge_kutta.html
Source: http://en.wikipedia.org/wiki/List_of_Runge%E2%80%93Kutta_methods

Simulation usually involves integration, for example for an object in free fall. In that case, we would have the variables s, v, a, and t. The most basic way of implementing integration in a computer program is Euler's method, which is basically choosing a very small time interval h and then incrementing each variable by their derivative multiplied by h.

Say we have a "vector" $\overline{\mathbf{X}} = (s, v, a)$, where $\overline{\mathbf{X}}$ is a "vector" containing all of the variables in the system. Next, let $\overline{\mathbf{X}'} = (v, a, 0)$, where $\overline{\mathbf{X}'}$ is a "vector" containing the derivative of every variable in the system.

Suppose that $\overline{\mathbf{X}}_n$ is the value of the variables at time t and that $\overline{\mathbf{X}}_{n+1}$ is the value of the variables at time $t + h$. Then Euler's method is basically

$$\overline{\mathbf{X}}_{n+1} = \overline{\mathbf{X}}_n + h\,\overline{\mathbf{X}'}_n$$

This can be generalized for any system of variables. Suppose that there are m variables x_1, x_2, ..., x_m, each of which vary over time. In the free fall example, x_1 would be position and x_2 would be velocity. Suppose there are m differential equations for these m variables

$$x_1' = f_1(t_n, x_1, x_2, ..., x_m)$$

$$x_2' = f_2(t_n, x_1, x_2, ..., x_m)$$

$$...$$

$$x_m' = f_m(t_n, x_1, x_2, ..., x_m)$$

Note that each function takes every other variable as an argument even though we don't necessarily need to use every variable. For example, in the free fall example, the derivative of acceleration would just be 0, which is not dependent on either position or velocity. But for ease of programming, we send all of those variables in as arguments anyway. Rather than cherry-picking which variable this derivative function will require, it's a lot easier to just send all of them and let the function decide which ones to use.

These equations can be summarized in vector form as

$$\overline{\mathbf{X}'} = \overline{\mathbf{f}}(t_n, \overline{\mathbf{X}})$$

where $\overline{\mathbf{X}} = (x_1, x_2, ..., x_m)$ and $\overline{\mathbf{f}} = (f_1, f_2, ..., f_m)$ in a loose "vector of functions" sort of concept. So say we label our time states $\overline{\mathbf{X}}_n$, $\overline{\mathbf{X}}_{n+1}$, which are separated by a time interval of length h. Then

$$\overline{\mathbf{X}}_n = (x_{1,n}, x_{2,n}, ..., x_{m,n})$$

$$\overline{\mathbf{X}}_{n+1} = (x_{1,n+1}, x_{2,n+1}, ..., x_{m,n+1})$$

So if we have the state of the simulation at time t_n as $\overline{\mathbf{X}}_n$, we want to compute the state a short time h later and put the results into $\overline{\mathbf{X}}_{n+1}$. And then do the same for $\overline{\mathbf{X}}_{n+1}$ and $\overline{\mathbf{X}}_{n+2}$, and keep going until a stopping condition is reached. This is either the user pressing the exit button on the top right corner of the screen, or some sort of calculated limit that the programmer puts in.

Integration Methods

We know that Euler's method is

$$\overline{\mathbf{X}}_{n+1} = \overline{\mathbf{X}}_n + h\,\overline{\mathbf{X}'}_n$$

but we would need a very small time step to achieve any sort of accuracy, and that is computationally expensive. There is a family of integration methods called Runge-Kutta methods, and Euler's method is the simplest of all of them. Euler's method is denoted RK1. Higher order RK methods use things like midpoint rule or Simpson's rule to achieve higher accuracy for the same time interval. For example, RK4, or "*the* Runge-Kutta method", does the following:

$$\overline{\mathbf{a}}_n = \overline{\mathbf{f}}(t_n, \overline{\mathbf{X}}_n)$$

$$\overline{\mathbf{b}}_n = \overline{\mathbf{f}}(t_n + \frac{h}{2}, \overline{\mathbf{X}}_n + \frac{h}{2}\,\overline{\mathbf{a}}_n)$$

$$\overline{\mathbf{c}}_n = \overline{\mathbf{f}}(t_n + \frac{h}{2}, \overline{\mathbf{X}}_n + \frac{h}{2}\,\overline{\mathbf{b}}_n)$$

$$\overline{\mathbf{d}}_n = \overline{\mathbf{f}}(t_n + h, \overline{\mathbf{X}}_n + h\,\overline{\mathbf{c}}_n)$$

$$\overline{\mathbf{X}}_{n+1} = \overline{\mathbf{X}}_n + \frac{h}{6}\,(\overline{\mathbf{a}}_n + 2\,\overline{\mathbf{b}}_n + 2\,\overline{\mathbf{c}}_n + \overline{\mathbf{d}}_n)$$

A general Runge-Kutta method can be specified using a Butcher tableau, which is of the form

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array}$$

The Runge-Kutta method for a particular Butcher tableau is defined as follows:

$$\overline{\mathbf{X}}_{n+1} = \overline{\mathbf{X}}_n + h\sum_{i=1}^{s} b_i k_i$$

$$k_i = \overline{\mathbf{f}}\left(t_n + c_i h, \overline{\mathbf{X}}_n + h\sum_{j=1}^{s} a_{ij} k_j\right)$$

In the solenoid simulator, t is not time but rather some parameter, but the general concept is the same. Our program only uses RK1, RK2 and RK4 because we couldn't find the Butcher tableau for any other ones. The Butcher tableau for the RK1 method that we used is:

0	0
	1

The Butcher tableau for the RK2 method that we used is:

0	0	0
$1/2$	$1/2$	0
	0	1

The Butcher tableau for the RK4 method that we used is:

0	0	0	0	0
$1/2$	$1/2$	0	0	0
$1/2$	0	$1/2$	0	0
1	0	0	1	0
	$1/6$	$1/3$	$1/3$	$1/6$

The RK1 is a first order method, called the "Forward Euler". It lacks stability, which means any error introduced accumulates in a positive feedback loop. It also has significant accuracy limits, so it is not an ideal integration method if any accuracy is desired.

The RK2 is a second order method, called the "Explicit midpoint method". It is significantly more accurate than RK1.

The RK4 is a fourth order method; the "original" Runge-Kutta method. It is more accurate than RK2.

Both RK2 and RK4 algorithms exhibit elements of numerical stability, which is good because then any errors introduced in the numerical integration tend to be damped out. The effect of a higher-order RK method is basically to provide the same level of accuracy for a larger time interval, which reduces computational cost.

Magnetism

Okay, about that t not being time business. See, we want to know the magnetic field of a particular system of wires and current in 3-space in one particular moment in time. Obviously in real life, this field would then induce emfs in the wires and then change the state of the wires which would then change the magnetic and electric fields and ugh that's too complicated. Our simulation is going to have solenoids floating in space not connected to a power source, so obviously it's not going to be accurate in that sense.

We are only concerned with $\vec{B}$ at this instant in time. So remember, t is not time. It is a parameter. This is useful because our wires are 1D functions in 3D, so the x,y,z coordinates can be computed as a function of this single parameter. "Regrettably, the variable t is often chosen to represent the parameter. One must remember that this has nothing to do with time!"(van Bemmel, 04 Sep 2014). In our case, it is rather fortunate that the parameter is t because that means we don't have to change notation (lazy).

First principles time. From Mr. van Bemmel's biotsavart primer, the law of Biot-Savart describes the differential magnetic field due to a current element by:

$$\mathrm{d}\vec{B} = \frac{\mu_0}{4\pi} \frac{i\,\mathrm{d}\vec{s} \times \hat{r}}{r^2}$$

Where:

$\mathrm{d}\vec{B}$ = the differential magnetic field at one specific reference point (T)
μ_0 = the permeability constant (exactly $4\pi \times 10^{-7}$ TmA^{-1})
i = the current in the current element (A)
$\mathrm{d}\vec{s}$ = the differential position on the current element (m)
$\hat{r}$ = the unit position vector pointing from the current element to the reference point (m)
r = the magnitude of the distance between the current element and the reference point (m)

To determine the $\vec{B}$ at a specific reference point, we sum all of these little $d\vec{B}$'s along the wire. Thus, the general algorithm is this: For each reference point, find the numerical integral of $d\vec{B}$ along each wire and sum the results of each wire to get the $\vec{b}$ at that point. We can sum the individual $\vec{B}$ caused by each wire because of the superposition principle: $\vec{B}$ is additive.

To compute this integral, we need the following variables:

1. $\vec{P}$ - the position of the reference point (m)
2. $\vec{s}$ - the position on the wire (m)
3. $\vec{B}$ - the magnetic field at the reference point (T)

We don't really need $\hat{r}$ since it can easily be computed from $\vec{P} - \vec{s}$. If we parameterize these variables with the parameter t, then

$$\vec{P}_0 = (P_x, P_y, P_z)$$

$$\vec{s}_0 = \text{get\_s}(t_i)$$

$$\vec{B}_0 = (0, 0, 0)$$

We can set $\vec{P}$ to whatever we want, and we begin $\vec{B}$ with the additive identity, the zero vector. We will denote the position on the wire with the function get_s(t_i) now, as this is dependent on the geometry of the wire. Here t_i represents the value of t that gives the start of the wire. We will get more into that later. Next, finding the derivatives with respect to t, we have:

$$\frac{\mathrm{d}}{\mathrm{d}t}\vec{P} = (0, 0, 0)$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\vec{s} = \text{diff\_s}(t)$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\vec{B} = \frac{\mu_0}{4\pi}\frac{i\,\frac{\mathrm{d}}{\mathrm{d}t}\vec{s} \times \widehat{(\vec{P}-\vec{s})}}{|\vec{P}-\vec{s}|^2}$$

The derivative of $\vec{P}$ is zero because it is constant. Also, because the current and permeability constant are... well, constant for each wire, we can save a bit of computational time by factoring these constants out of the integral, and multiplying the $\vec{B}$ by the constant only after the integral is done. Thus, we compute

$$\vec{B} = \sum \frac{\mu_0\, i_k}{4\pi} \int \frac{\frac{\mathrm{d}}{\mathrm{d}t}\vec{s} \times \widehat{(\vec{P}-\vec{s})}}{|\vec{P}-\vec{s}|^2}\,\mathrm{d}t$$

Where i_k is the current in the kth wire being integrated. We have almost everything we need for the Runge-Kutta algorithm to be implemented now. Let

$$\overline{\mathbf{X}} = (\vec{P}, \vec{s}, \vec{B})$$

$$\overline{\mathbf{X}'} = \left(\frac{\mathrm{d}}{\mathrm{d}t}\vec{P}, \frac{\mathrm{d}}{\mathrm{d}t}\vec{s}, \frac{\mathrm{d}}{\mathrm{d}t}\vec{B}\right)$$

Recall that a Runge-Kutta algorithm is of the form

$$\overline{\mathbf{X}}_{t+h} = \mathrm{rkn}(t, \overline{\mathbf{X}}_t, \overline{\mathbf{X}'}_t)$$

Thus, for each wire, we substitute $\overline{\mathbf{X}}_0$ and $\overline{\mathbf{X}'}_0$ and iterate until $t = t_f$, the end of the wire. So, the only thing to figure out now is the shape of the wire.

The shape of the wire

To be able to run this algorithm, we need to know the following:

1. $s(t)$ - the position on the wire as a function of t
2. $\frac{\mathrm{d}}{\mathrm{d}t}s(t)$ - the derivative of the wire position wrt t
3. t_i - the value of t at the start of the wire
4. t_f - the value of t at the end of the wire

Let's start easy. Say we have a straight line as a wire. Then we can define this wire with a position vector and a direction vector. If the direction vector is a unit vector, then $t_f - t_i$ gives the length of the wire. If $t_i = 0$, then the wire is a straight line extending from the position of the position vector along the direction vector for length t_f. Let $\vec{o}$ be the "origin" or position vector of the wire. Let $\hat{m}$ be the direction vector. Thus, for a straight wire:

1. $s(t) = \vec{o} + t\,\hat{m}$
2. $\frac{\mathrm{d}}{\mathrm{d}t}s(t) = \hat{m}$
3. $t_i = 0$
4. $t_f = L$

where L is the length of the wire.

So what about a solenoid? Well, it is a coil of wire of length L, radius r and n turns. Since it loops like a spiral, we could represent a solenoid with the equation of a helix:

$$s(t) = (\cos(t), \sin(t), t)$$

This describes a helix of radius 1, rising by 2π units per turn (Figure 1).

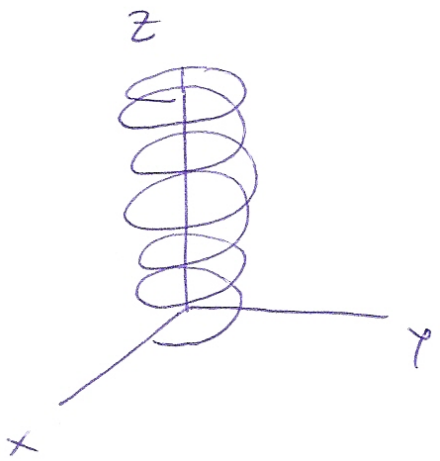

Figure 1: Looking at the origin from the z-axis, the helix traces out an ascending circular path starting at (1,0,0) and turning counterclockwise as t increases.

The helix can be generalized for radius r and turn density $\frac{L}{n}$ as follows:

1. $s(t) = \left(r\cos(t), r\sin(t), \dfrac{L}{2\pi n}t\right)$

Thus it follows that

2. $\dfrac{\mathrm{d}}{\mathrm{d}t}s(t) = \left(-r\sin(t), r\cos(t), \dfrac{L}{2\pi n}\right)$
3. $t_i = 0$
4. $t_f = 2\pi n$

Okay, now what if we want a solenoid pointing in an arbitrary direction? For the unmodified equation, the helix circles upwards in the +z direction. So let's define the default helix direction vector $\hat{a}$ to be $(0, 0, 1)$. If we want the solenoid to be pointing in the arbitrary direction $\hat{b}$, then we want to find a rotation matrix $\mathbf{R}$ that rotates $\hat{a}$ onto $\hat{b}$ (Figure 2).

Turning to the internet, the source http://math.stackexchange.com/questions/180418/calculate-rotation-matrix-to-align-vector-a-to-vector-b-in-3d/476311 answers this question. The following is copied verbatim:

Suppose you want to find a rotation matrix $\mathbf{R}$ that rotates unit vector $\hat{a}$ onto unit vector $\hat{b}$. Proceed as follows:

Let $\vec{v} = \hat{a} \times \hat{b}$

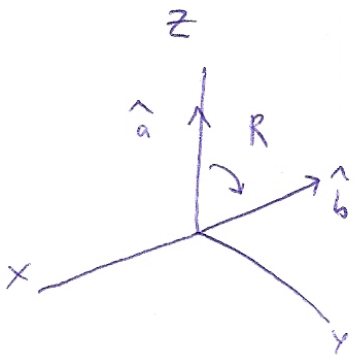

Figure 2: A rotation matrix $\mathbf{R}$ would rotate $\hat{a}$ onto $\hat{b}$. The rotated helix would circle counterclockwise along the $\hat{b}$ axis.

Let $s = \|v\|$ (sine of angle)

Let $c = \hat{a} \cdot \hat{b}$ (cosine of angle)

Then the rotation matrix R is given by:

$$\mathbf{R} = \mathbf{I} + [\vec{v}]_\times + [\vec{v}]_\times^2 \frac{1-c}{s^2},$$

where $[\vec{v}]_\times$ is the skew-symmetric cross-product matrix of v,

$$[\vec{v}]_\times \stackrel{\text{def}}{=} \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix}.$$

End verbatim. Also, $\mathbf{I}$ is the identity matrix.

This process fails with only two cases: If $\hat{a}$ is parallel with $\hat{b}$ and if $\hat{a}$ is antiparallel with $\hat{b}$. Thus, to account for these two cases, do the following:

1. If $s = 0$,
    a. If $c > 0$, set $\mathbf{R}$ to $\mathbf{I}$.
    b. If $c < 0$, set $\mathbf{R}$ to $-\mathbf{I}$.
    c. If $c = 0$, set $\mathbf{R}$ to $\mathbf{0}$.

Since this $\mathbf{R}$ is constant for any given solenoid, we only need to compute the value of $\mathbf{R}$ once when initializing the solenoid. Afterwards, we just reference it from memory. Furthermore, because rotation is a linear transformation, this does not make finding the derivative any more complicated so yay. Our rotated solenoid would circle counterclockwise if we look at the origin from $\hat{b}$, travelling up along $\hat{b}$ (Figure 3).

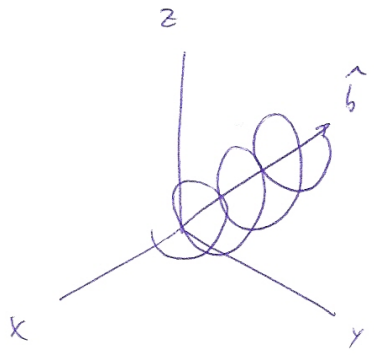

Figure 3: A rotated solenoid circling up along a $\hat{b}$ axis.

Thus, for a solenoid rotated away from the default helix by rotation matrix $\mathbf{R}$:

1. $s(t) = \mathbf{R} \begin{bmatrix} r\cos(t) \\ r\sin(t) \\ \frac{L}{2\pi n}t \end{bmatrix}$
2. $\frac{\mathrm{d}}{\mathrm{d}t}s(t) = \mathbf{R} \begin{bmatrix} -r\sin(t) \\ r\cos(t) \\ \frac{L}{2\pi n} \end{bmatrix}$
3. $t_i = 0$
4. $t_f = 2\pi n$

The only modification is that we have left multiplied the equations in 1 and 2 by the rotation matrix $\mathbf{R}$. So now that we have $s(t)$ and $\frac{\mathrm{d}}{\mathrm{d}t}s(t)$, we can use the Runge-Kutta algorithm of choice to integrate from t_i to t_f for each wire and sum the results of each wire to get the net $\vec{B}$ at point $\vec{P}$.

TODO:

How to determine the accuracy for a given RK algorithm and dt interval?

-Simulate a solved system, e.g. a length of wire or a circle of wire; compare residuals of different sim settings

-Compare simulated values with measured values in real life; compare residuals of different sim settings