That's it, guys!

which is a little bit complicated. But if you are dealing with a lot of rotations, it is better to tranlate a matrix into a quaternion than do all the matrix multiplications.

- Quaternion addition is easy, just add scalar to scalar, and vector to vector.
- Quaternion multiplication is really easy. The formula is

$$[s_1, v_1][s_2, v_2] = [(s_1 s_2 - v_1 \bullet v_2), (s_1 v_2 + s_2 v_1 + v_1 \times v_2)]$$

- Quaternions have a very simular formula as rotation matrices; while for rotation matrices,

$$\dot{R}(t) = w(t) \times R(t)$$

for quaternions,

$$\dot{q}(t) = \frac{1}{2} w(t) q(t)$$

- In addition to smaller numerical drifts, quaternions are also easier and faster to compute. It is also perfect in use with spherical coordinates.

Quaternions - Getting Rid of Nasty Matrices

- If you have taken cs123(and you are a fan of Andy), you probably know about all those nasty orientation matrices and their manipulations.
- Now you can throw those away, because you are going to learn a new kind of mathematics : Quaternions.
- A quaternion consists only four numbers. A scalar and a 3-d vector. The format of a quaternion is (w,(x,y,z)), and is frequently written as [s,v], with s as the scalar w, and v be the vector with components x, y and z. Quaternions are always normalized($w^2 + x^2 + y^2 + z^2 = 1$).
- The vector gives the coordinates for the axis of rotation, while the scalar is determined by the angle rotated through.
- A matrix has nine numbers, while a quaternion has only 4 number to keep track of the 3 degrees of freedom. Clearly, as the degree of redundancy is far higher for matrices, a matrix experiences more numerical drifts than a quaternion. The only time when a quaternion experiences drifts is in its magnitude. Therefore that is why quaternions are always normalized to avoid the drift.
- To translate a quaternion into a rotation matrix, we have

$$\begin{bmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix}$$

- Therefore we not know that the solution of the equation is unbounded between t = 0.785 and t = 1.

- Therefore mistakes can be easily made by blindly plugging values into sets of equations. Even if something reasonable comes up, you might be wrong!

- This problem can be remedied by employing a combination of analytical and numerical work to carefully study the differetial equations.

3. Other minor problems

- Numerical Dependency. When solving second or higher order equations, if some solutions happened to grow really fast, even if they are linearly independent, they would be identical in the eyes of the computer. Try using higher precision machines.

- Solutions around singular points. When we are dealing with equations that have a lot of singular points, chances are that you will have to compute solutions which are really close to some of the singular points. If not taken carefully, extraneous computational error can occur frequently.

2. Existence

- Assuming that we have a nonlinear equation, for example,

$$y' = t^2 + y^2$$

$$y(0) = 1$$

- Suppose we try to compute a solution of this problem on the interval between 0 and 1 inclusive using the Runge-Kutta method.

Table 1: Calculation of the solutions for the above equation.(courtesy of Elementary Differential Equations and Boundary Value Problems by Boyce and Prima)

h	t =0.90	t = 1.0
0.1	14.02158	735.0004
0.05	14.27042	1.755613 * 10^4
0.01	14.30200	> 10^15

- Fromt the table, it's reasonable to say that the solution approaches 14.3 for t=0.90. But something strange is happening between 0.9 <= t <= 1.0

- Analytical work shows that the solution of the above equation is actually

y_1 = tan(t + pi/4)
y_2 = 1/(1-t).

For the first one, y -> infinity as t -> pi/4 = 0.785, and for the second one, y -> infinity as t -> 1.

Difficulties with Numerical Methods

1. Optimization

- Even though smaller h would result in a less truncation error; after a certain point, too many steps of computations over a certain interval would result in a significant round-off error. So the dilemma is....

Smaller h yields smaller truncation yet bigger round-off error
Bigger h yields bigger truncation yet smaller round-off error

- So we need to find the optimum value of h which to minimize the total error, which is the sum of the truncation and round-off error. That means that h is near the optimum when truncation and round-off error are about the same in magnitude.

- Finding a suitable h might require a lot of work, sometimes might be even more then the actual work consumed by numerical approximation methods.

- Mathematically, k_{n1} is the slope at the left end of the interval, k_{n2} is the slope at the midpoint, k_{n3} is a second approximation of the slope at the midpoint, and k_{n4} is the slope at the right end of the interval. $(k_{n1}+2k_{n2}+2k_{n3}+k_{n4})/6$ is the average slope over the interval.

- Local truncation error is proportional to h^5
- Global truncation error is at most a constant times h^4

- That means if we improve h by 1/10, then the error bound will be reduced to between 0.00001! Though the math is lengthy, and it requires a few more computations on the machine. However, it simplicity and accuracy allows Runge-Kutta to be one of the widely used method for solving differential equations numerically.

- By the way, if the equation rings a bell to you, it is because that is very closely related to the famous Simpson's rule that all of you have learned in high school.

-You may say, wow, it is so simple to solve any first-order differential equation! However, there are a few technical difficulties arise by using numerical approximation.....

The Runge-Kutta Method - Run and Cut?????

- The concept behind Runge-Kutta is basically the same as the methods that we have discussed above.

- However, instead of taking only one or two points between the interval, this method involves a weighted average of values of f(t,y) taken at different points in the interval t_n <= t <= t_{n+1}.

- The complete equation is....

$$y_{n+1} = y_n + \frac{h}{6}(k_{n1} + 2k_{n2} + 2k_{n3} + k_{n4})$$

in which,

$$k_{n1} = f(t_n, y_n)$$

$$k_{n2} = f\left(\left(t_n + \frac{1}{2}h\right), \left(y_n + \frac{1}{2}hk_{n1}\right)\right)$$

$$k_{n3} = f\left(\left(t_n + \frac{1}{2}h\right), \left(y_n + \frac{1}{2}hk_{n2}\right)\right)$$

$$k_{n4} = f((t_n + h), (y_n + hk_{n3}))$$

- No sweat! Because we know that,

$$y'_n = f(t_n, y_n)$$

$$y''_n = f_t(t_n, y_n) + f_y(t_n, y_n)y'_n$$

- As you probably can see here, y' and y'' can be easily computed in the program, so the Three-Term Taylor Method is also feasible.

- The local truncation error of this method is the same as that of the improved Euler Method. However, this method is less used in programming because f_t and f_y may be more complicated than the function itself.

- But let's say someone is programming something that model exactly how the real physics works, then all the methods we have discussed is still not good enough for it.

- Therefore, the next question is, is there a method which can do a much better job than the three methods which we have discussed?

The Three-Term Taylor Method - a Variation of Euler

- In addition to the two Euler methods, the Three-Term Taylor Method can be used as a substitute. It rests on the idea of the Taylor expansion of a function.

$$\Phi(t_n + h) = \Phi(t_n) + \Phi'(t_n)h + \Phi''(t_n)\frac{h^2}{2} + terms$$

- Let's neglect everything after the third term. it will leave us with....

$$y_{n+1} = y_n + hy'_n + \frac{h^2}{2}y''_n$$

- But we have a first-order and a second-order differential!! What do we do?

doing an error analysis on this method?
- By using the same method of error analysis for the original Euler Method, the local truncation error in the improved Euler Method is........

$$\mathbf{h^3}$$

- Since we are only adding several arithmetic operation, this trade-off is definitely a good one. Therefore, if you need to solve differential equations in the future and afford to do a few more computations each round, the improved Euler is the way to go.

have.......

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

- The above equation is the improved Euler method! Taking the average of the values of the endpoints certainly helps the accuracy of this method.
- And if for the above equation f(t, y) doesn't depend on y, then we have the famous trapezoid rule fornumerical intergration.

- To implement the improved Euler Method is actually really easy. All you need is to change a few lines of code in the for loop.

for ....
$k_{i1} = f(t_i, y_i)$
$k_{i2} = f(t_i + h, y_i + hk_{i1})$
$y_{i+1} = y_i + (h/2)(k_{i1} + k_{i2})$
$t_{i+1} = t_i + h$
loop

- But you may wonder, is that baby really improved? How about

Improved Euler Method - Bigger and Better

- The plain old Euler Method is just not good enough for approximations........

- Therefore, instead of just getting the tangent from the n-th point of t and h, why don't we take the tangent from the point which is the midpoint between the n-th and the (n+1)-th points!

- If the above statement is too vague, no sweat! Let's take a look at the old method....

The Euler Method:

$$y_{n+1} = y_n + hf(t_n, y_n)$$

- Now if we take the average between the n-th and the (n+1)-th points, we have

$$y_{n+1} = y_n + \frac{f(t_n, y_n) + f(t_{n+1}, y_{n+1})}{2}h$$

- However, y_{n+1} is unknown to us, and this is very difficult to solve this equation. But by the definition of the old method, we

- Two practical comments on Error Analysis

1. To estimate the size of the truncation error, after completing the computations with a given h, repeat the computations with h/2. If the changes are greater than we can tolerate, then use a smaller step size or a more accurate method.

2. To estimate the size of the round-off error, we can first do a single-precision arithmetic, and then a double-precision arithmetic. Again, if the changes are greater than we can tolerate, then use a smaller step size or a more accurate method.

- Total Error of an approximation : it is bounded by the sum of the absolute value of the global truncation and global round-off error.

- So let's take the Euler Method, and let's do an error analysis on it........

Local Truncation Error = h^2

Global Truncation Error < h

(I am not going to show the proof here. But if you are mathematically INCLINED, go take am136 and you will know why.)

- This means that if you reduce the size of h by 1/10, your error bound is decreased by 1/100, but you have to do 10 more computations each loop. Here presents a good example of accuracy vs speed.

Error Analysis - The bad side of approximations

Several issues for error analysis :

- Convergence : Does the method we use can really do the job of approaching the correct solutions?

- Accuracy vs Speed : Do you need accurate answer or fast computation speed?

- Global Truncation Error :
1. the use of approximate formula to determine y_{n+1}
2. the input data at each step does not agree with exact solution

- Local Truncation Error :
1. No input data error
2. It is simpler to analyze than Global Truncation Error

- Global Round-off Error :
1. the difference between the exact value of y_n and the actual value of y_n computed by the computer
2. it clearly depends on a lot of other factors -- type of computer used, sequence of computation, method of rounding off, etc

$$y_{n+1} = y_n + hf(t_n, y_n)$$

- Here is the computer pseudocode for the Euler Method:

```
define f(t,y)
input initial values t0 and y0
input step size h and number of steps n
output t0 and y0
for i from 1 to n do
    k_i = f(t_i,y_i)
    y_i+1 = y_i + hk_i
    t_i+1 = t_i + h
    ouput t and y
next i
end
```

- if h is set to be too large, you are in BIG trouble
- this method is too primitive and inaccurate for programs which need really precise results.

The Euler Method - Unreliable, yet Quick

- Assuming that the differential equations are in the form of

$$\frac{dy}{dt} = f(t, y)$$
$$y(t_0) = y_0$$

- Given an initial value, we want to know what y_n is. And the easiest way to do it is by the Euler Method.

- Since t_0 and y_0 are known, the slope of the line tangent to the solution at t_0, which is $f(t_0,y_0)$ is also known. Therefore we can approximate y1 by moving along the tangent line from t_0 to t_1. Therefore it results in

$$y_1 = y_0 + f(t_0, y_0)(t_1 - t_0)$$

- If we assume that each step between t0, t1, t2..... is uniform(let's call it h, which is equal to $(t_{n+1} - t_n)$), then the general formula for the Euler Method would be........

Numerical Approximations!!

Advantages:
- Very useful in game prgramming
- Fast, and efficient way to model real world
- You can even do all approximations on some pocket calculators!!

Modelling the Real World

- For simple flight simulation games, what do you need?
- For exciting car racing games, what do you need?
- For dumb sports games, what do you need?
- For mysterious space shooting games, what do you need?
- For complex animal and human behavior games, what do you need?

You need

physics!!

- What is the most often mathematical form of equations you will encounter in physics? Differential Equations.
- Now is the time to refresh and sharpen your high school calculus!
- For example, if you are given a first-order differential equation for motion and the initial position of a particle, do you know where the particle will be ten seconds later?
- There are a lot of nice differential equation.
- Nice and easy differential equation:

$y' = t^2 + 3t + 2$

$y' = y/t$

- The above equations can be solved by hand first. Then we can plug in the formula for y directly into the program.
- However, diffrential equations aren't always nice, so what you need is.........

The Mathematics behind Σιμυλατιονσ

- Modelling the Not-so-Perfect Real World

- The Euler Method

- Error Analysis

- The Improved Euler Method

- The Three-Term Taylor Formula

- The Runge-Kutta Method

- Difficulties with Numerical Approximations

- Quaternions