

# **Classical Physics Simulation**

## **Seminar Report**

Submitted in partial fulfillment of the requirements for

### **Seminar**

of

VIII Semester BE (CSE)  
(Visveswaraiah Technological University)

Submitted by

**Sriram Kashyap M S**  
*IMS04CS095*

April 2008



**M. S. Ramaiah Institute Of Technology**

Department of Computer Science and Engineering

Bangalore-560054

## Certificate

This is to certify that Mr./Ms. *Sriram Kashyap M S, IMS04CS095* has satisfactorily completed the seminar on *Classical Physics Simulation* prescribed by Visveswaraiah Technological University for VIII Semester (CSE) in the year 2008.

Internal Guide

Examiners :

1.

2.

## **Acknowledgement**

I would like to thank Dr. V K Ananthashayana, Mrs. Anitha Kanvalli and Dr. K G Srinivasa for the guidance and support they provided during my preparation for the seminar. I would also thank the faculty and support staff of the Computer Science department of MS Ramaiah Institute of Technology, for their and support and encouragement.

## Abstract

Computers are widely used to simulate many physical, chemical and biological systems. The simulation of classical physics on computers is one such field of study.

Based on the applications, Physics simulations can either be non Real-Time (accurate simulations) or Real-Time (approximate simulations).

This report deals with the basic issues that have to be handled for real time physics simulations. ie: consumer level physics simulation software.

A further classification is based on whether the simulation is discrete or continuous. Discrete simulations offer more flexibility to the programmer and are mathematically simpler, and therefore, a brief introduction is given to continuous simulations and the topic of discrete simulation is elaborated upon.

Basic properties of a physical system like position, velocity and forces are modeled using vectors. Other scalar properties of a system like mass, friction and restitution are also simulated.

Using the above ideas, a slightly more complicated system is constructed. This is the Spring – Mass system. A spring with single free mass attached to one end and one fixed end, is demonstrated. Then a generalized spring mass system with multiple springs attached to multiple masses is also modeled and demonstrated.

The concept of collision detection is introduced to deal with the simple case of point masses colliding with line segments. An extension and optimization of this technique, using a two-phase collision detection approach is also introduced. This involves a coarse collision detection stage and a detailed collision detection stage, for dealing with complex systems.

Finally, more advanced simulation techniques such as Finite Element Method and Inverse Kinematics are introduced. Current trends in physics simulation like dedicated physics hardware and physics engines are discussed.

# Contents

• Acknowledgements		
• Abstract		
1. Introduction	:	06
2. Basic properties of Physical Systems	:	07
3. Relation between the basic parameters	:	08
4. Discrete Simulation	:	09
5. Implementing the discrete simulation	:	10
6. Spring Mass System	:	11
7. Runge-Kutta Method	:	12
8. Implementation of an RK4 solver.	:	13
9. Complex Simulations using Spring-Mass Analogy	:	15
10. Inverse Kinematics	:	16
11. Conclusion	:	17
12. Bibliography	:	18
13. Slides	:	19

# Introduction

Computers are widely used to simulate many physical, chemical and biological systems. The simulation of classical physics on computers is one such field of study. The first electronic computer, the ENIAC was used to calculate trajectories of ballistic missiles. These complex calculations involved several parameters like density of atmosphere at different altitudes, wind currents, the forces due to rotation of the earth etc, which cannot be easily factored in during manual calculations.

These days, the major supercomputing applications are in fields like protein folding, weather forecasting, crash testing for vehicles, wind tunnel simulations etc.

Apart from these High Performance Computing fields, physics simulations have found their way onto consumer level hardware like desktops and mobile devices. Here the applications for physics simulations lie in interactive environments like games, and also in new intuitive and highly responsive interfaces like those found on the iPhone, and on the Beryl window manager for Linux.

While some of these applications are non real-time, and demand a high degree of accuracy, the consumer oriented physics simulations are often real-time approximations of the actual physical phenomena. The remainder of the report deals with real-time simulation of classical physics, with the main application in interactive graphics.

## Basic Properties of Physical Systems

A physical system has the following properties:

- Mass
- Position
- Velocity
- Acceleration
- Force

Out of these, acceleration is a derived attribute, because we can obtain the value of acceleration given the force acting on the object and its mass, using the equation :

$$F = M * A$$

The advantage of choosing force over acceleration is that force can be directly applied to multiple objects.

Mass is represented as a scalar quantity and the remaining parameters are all vector quantities in 2 or 3 dimensions (depending on the scenario).

Therefore, a point object can be represented in a user-defined data type as follows:

```
struct vector{  
    float x,y,z;  
}  
struct pointObject{  
    vector position,velocity,force;  
    float mass;  
};
```

## Relation between the basic parameters

Force is equal to mass times acceleration:

$$F = M \times a$$

Velocity is the first derivative of position.

$$dx/dt = v$$

Acceleration is the first derivative of velocity.

$$dv/dt = a$$

Integrating the above equation between the limits  $v=U$  and  $v=V$ , we arrive at the relation:

$$V = U + at$$

This combined with the second equation, gives us:

$$x = Ut + 0.5 * a * t^2$$

These equations represent Newton's laws of motion. They hold true for both scalar and vector values of position, velocity and acceleration.

The differential equations described above can be used for discrete event based simulation for physical systems.

Similarly, their closed form solutions can be used for a continuous simulation of the system.



## Discrete simulation

The discrete event simulation involves solving the differential equations previously discussed, using numerical methods.

There are two well known numerical methods to solve such equations:

Finite Difference Method (Euler's Method):

This involves approximating the definition of derivative of a function:

$$F'(x) = \lim_{(h \rightarrow 0)} [F(x+h) - F(x)] / h$$

The approximation involves dropping the limit on the above equation, to get:

$$F'(x) = [F(x+h) - F(x)] / h$$
$$F(x+h) = h * F'(x) + F(x)$$

This allows us to calculate the value of the function F at a new point (x+h), when we know the value at x, and the value of F' (the derivative) at x.

This can be applied to physics simulation as follows:

Consider Velocity (v) is a function of time v(t). Then, we have:

$$v(t+\Delta t) = \Delta t * v'(t) + v(t).$$

We know that  $v'(t)$  = acceleration at time t.

Similarly, considering position as a function of time x(t), we have:

$$x(t+\Delta t) = \Delta t * x'(t) + x(t).$$

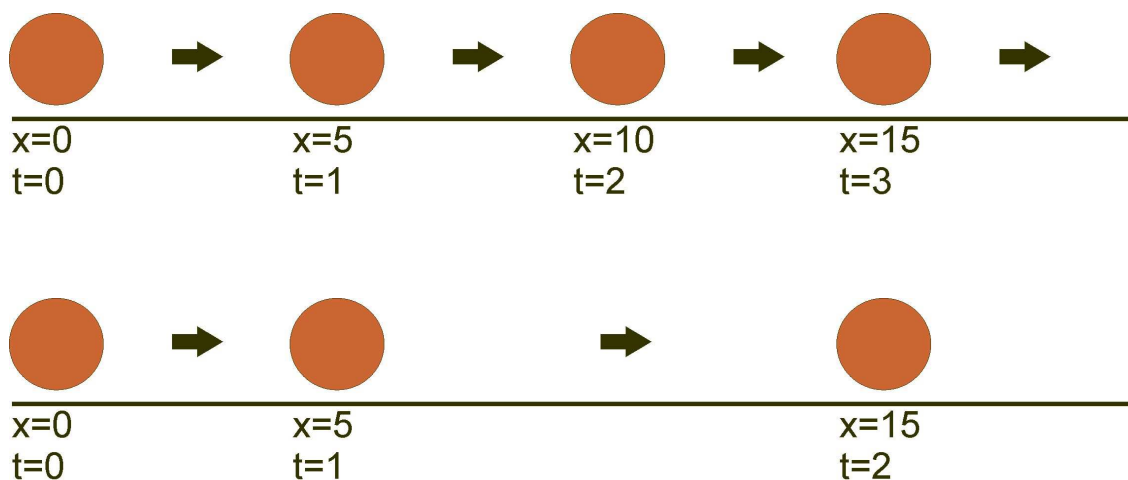
Again,  $x'(t)$  is the velocity of the object at time t.

## Implementing the discrete simulation

The above approximations for velocity and position as a function of time can now be used, in conjunction with the force equation to implement a simulation in any programming language. Consider the following C implementation:

```
const int timeStep = 1;
while(1)
{
    screen.clear();
    vector acceleration= object.force / object.mass;
    object.velocity = (timeStep * acceleration) + object.velocity;
    object.position = (timeStep * velocity) + object.position;
    object.draw(screen);
    sleep(timeStep);
}
```

Illustration with constant velocity, then constant acceleration:



## Spring-Mass Systems

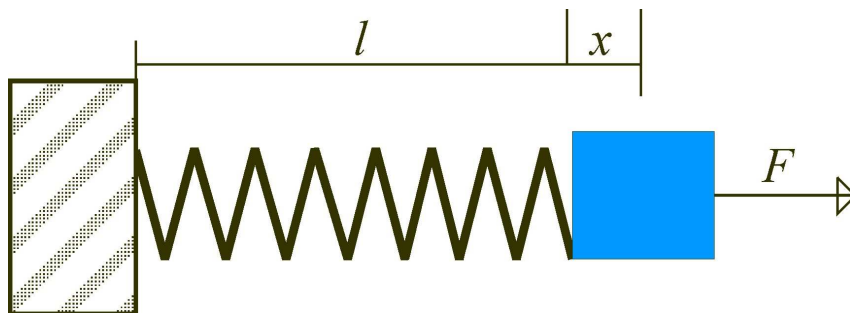
The behavior of a spring mass system is described by Hooke's Law.

$$F = -kx$$

Here  $F$  is the force exerted on the mass by the spring, when an extension or compression of  $x$  units is affected on the spring. The value  $k$  is the spring constant. The natural length of the spring is  $l$  units. The mass of the object attached to the spring is  $M$ . The other end of the spring is attached to an unyielding wall.

A modification of the above equation adds damping to the system. The form of damping we use here is viscous drag, where the force is inversely proportional to the velocity of the mass. A value  $d$  is the damping coefficient and it is multiplied by the velocity to get the force. Therefore, the spring mass equation becomes:

$$F = -kx - dv$$



To solve equations of the above type, ie: for oscillating systems, we need to use a slightly more accurate solution for the differential equations. The Euler method can cause the system to wildly oscillate out of equilibrium.

## Runge Kutta Method: A Better Approximation

A problem with the finite difference method is that if the time step is large, the method can produce some undesirable results. These results can manifest even if the parameters like velocity, or acceleration are large. The system tends to become highly unstable, even though in actual physical systems, for the same parameters, the system would have been stable.

To mitigate these problems, another popular numerical method is used. This is the fourth order Runge-Kutta method (RK4).

Let an initial value problem be specified as follows.

$$y' = f(t, y), \quad y(t_0) = y_0.$$

Then, the RK4 method for this problem is given by the following equations:

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4) \\ t_{n+1} &= t_n + h \end{aligned}$$

where  $y_{n+1}$  is the RK4 approximation of  $y(t_{n+1})$ , and

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\ k_4 &= f(t_n + h, y_n + hk_3) \end{aligned}$$

The main advantage of this method is that the function value is approximated halfway between the time intervals. This reduces any problems caused by large parameter values.

## Implementation of RK4 Solver

Below is the implementation of a Runge Kutta 4<sup>th</sup> order solver, which simulates a damped one dimensional spring. The initial condition is the extension of the spring mass system, which is given as input vars(0).

vars(1) and vars(2) are velocity and acceleration components respectively.

At each stage, a temporary output position, velocity and acceleration are all calculated.

These values are stored in the inp() array, and passed as inputs to the next stage of computation. The program calls the step method at each time interval. The step method takes the current system state as input and returns the changes in the respective state variables as output. The x() array stores input variables position, velocity and acceleration, and the change() array returns the change in position and change in velocity as its first 2 elements.

```
Public Const N = 2
```

```
Public inp(N) As Double
```

```
Public k1(N) As Double
```

```
Public k2(N) As Double
```

```
Public k3(N) As Double
```

```
Public k4(N) As Double
```

```
Public vars(N) As Double
```

```
Public Const springK = 1
```

```
Public Const springD = 1
```

```
' Runge-Kutta method for solving differential equations
```

```
' vars = array of variables
```

```
' N = number of variables in array
```

```

Public Sub step(stepSize As Double)
    evaluate vars, k1 ' evaluate at time t
    For i = 0 To N
         $inp(i) = vars(i) + k1(i) * stepSize / 2$ 
    Next i
    evaluate inp, k2

    For i = 0 To N
         $inp(i) = vars(i) + k2(i) * stepSize / 2$ 
    Next i

    evaluate inp, k3

    For i = 0 To N
         $inp(i) = vars(i) + k3(i) * stepSize$ 
    Next i
    evaluate inp, k4

    For i = 0 To N
         $vars(i) = vars(i) + (k1(i) + 2 * k2(i) + 2 * k3(i) + k4(i)) * stepSize / 6$ 
    Next i
End Sub

Public Sub evaluate(x() As Double, change() As Double)
    change(0) = x(1)
     $change(1) = -springK * x(0) - springD * x(1)$ 
End Sub

```

## Complex Simulations using Spring-Mass analogy

It is possible to extend the spring mass system as an approximation to more complex physical systems like soft body dynamics simulations, and height field based fluid simulations.

A system with multiple masses attached together in a mesh, by multiple springs, gives the illusion of soft body physics. This method is similar to the finite element method of simulating complex systems as a grid based approximation.

The analogy can be taken even further to simulate a height-field of water, to generate realistic waves that superimpose, and reflect off obstacles. This is done by considering the height field to be a grid of nodes, equally spaced, and connected to their neighbors by springs. When one of the nodes is displaced, the neighbors also get displaced in an oscillatory manner due to the springs connecting them. These oscillations travel throughout the grid causing the height field to look like waves in water.

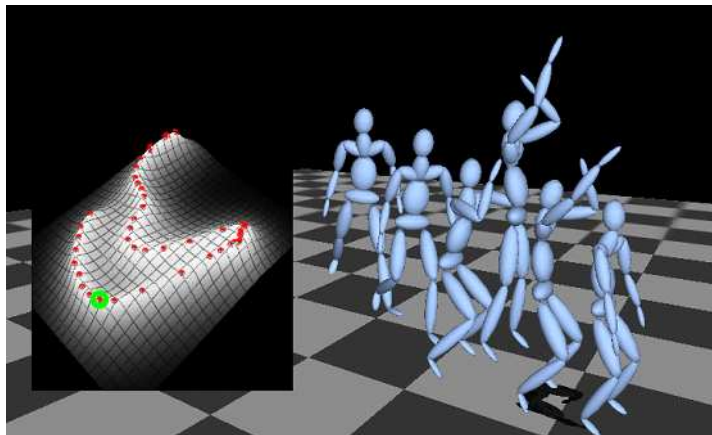


# Inverse Kinematics

Inverse kinematics is the process of determining the parameters of a jointed flexible object (a kinematic chain) in order to achieve a desired pose. Inverse kinematics are also relevant to game programming and 3D animation, where a common use is making sure game characters connect physically to the world, such as feet landing firmly on top of terrain.

An articulated figure consists of a set of rigid segments connected with joints. Varying angles of the joints yields an indefinite number of configurations. The solution to the inverse kinematics problem is to find the joint angles given the desired configuration of the figure (i.e., end effectors). The general case has no analytic solution.

Inverse kinematics is a tool utilized frequently by 3D artists. It is often easier for an artist to express the desired spatial appearance rather than manipulate joint angles directly. For example, inverse kinematics allows an artist to move the hand of a 3D human model to a desired position and orientation and have an algorithm select the proper angles of the wrist, elbow, and shoulder joints.





## Conclusion

Computers these days are being used to perform increasingly realistic simulations of physical phenomena. Even in the realm of physics simulations, there are several branches apart from those discussed in this report. Thermodynamics simulations involving behavior of materials under stresses due to high temperature is one such field. Another is the field of optics simulation for realistic graphics and rendering technologies. Graphics algorithms that were once restricted to the most powerful workstations and render farms, are now being used on desktop hardware in computer games and other interactive simulations. Dedicated graphics hardware has fuelled this movement.

Another upcoming area is that of dedicated physics hardware. Two of the industry's major players, Intel (which acquired Havok) and nVidia (which acquired Ageia) are entering the physics hardware market in a big way.

Physics simulations are also seen on a smaller scale on day to day interactive systems like the i-Phone. Modern desktop computing interfaces also have the basic elements of physics integrated into them.

At a larger scale, the industry has great demand for physics simulations for complex problems like weather forecasting, flight simulators, crash testing of vehicles, testing of new materials and alloys, all of which serve to reduce the cost of research, development and training.

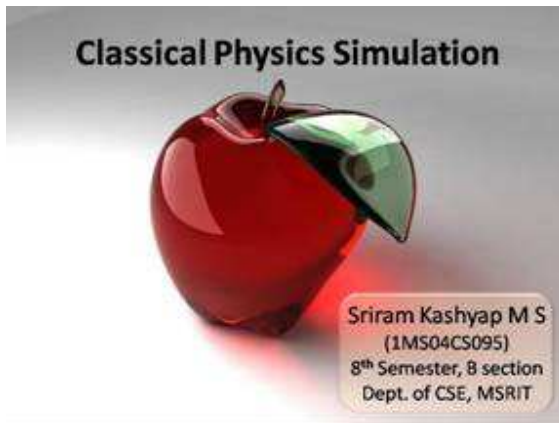
## Bibliography

- <http://www.myp physiclab.com> (Physics simulations as Java Applets)
- [http://en.wikipedia.org/wiki/Numerical\\_ordinary\\_differential\\_equations](http://en.wikipedia.org/wiki/Numerical_ordinary_differential_equations)
- <http://www.harveycartel.org/metanet/tutorials/tutorialA.html> (Collision detection and handling)
- [http://www.sv.vt.edu/classes/MSE2094\\_NoteBook/97ClassProj/num/wid as/history.html](http://www.sv.vt.edu/classes/MSE2094_NoteBook/97ClassProj/num/wid as/history.html) (Finite Element Method)
- [http://freespace.virgin.net/hugo.elias/models/m\\_ik.htm](http://freespace.virgin.net/hugo.elias/models/m_ik.htm) (Inverse Kinematics tutorial)

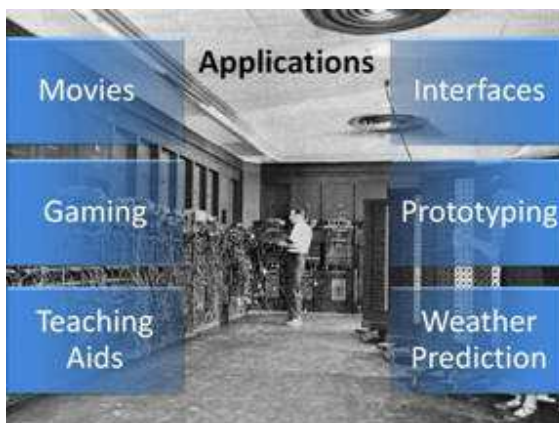
### Commercial Physics Solutions:

- <http://www.ageia.com/> (Ageia PhysX SDK and Hardware- nVidia)
- <http://www.havok.com/> (Havok Physics Engine- Intel)
- <http://www.pixeluxentertainment.com/> (Digital Molecular Matter: Real time FEM on desktop hardware)

# Slides



## Scope of Classical Physics



## Jerk, snap, crackle and pop!!

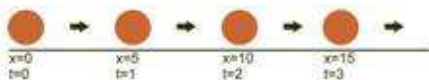
### Properties:

- Mass :  $M$
- Position :  $x$
- Velocity :  $dx/dt$
- Acceleration?? :  $d^2x/dt^2$
- Force??

```
struct vector3d
{
    float x, y, z;
};

struct physicsObject
{
    float mass;
    vector3d position;
    vector3d velocity;
    vector3d force;
};
```

## Moving



```
while (1){
    position = position + velocity*timeScale;
    sleep(timeScale);
}
```

## Moving Faster



```
while (1){
    velocity = velocity + acceleration*timeScale;
    position = position + velocity*timeScale;
    sleep(timeScale);
}
```

# Slides

But... something is wrong!!

$$v = dx/dt$$

$$\Delta x = x_2 - x_1$$

$$\lim_{\Delta t \rightarrow 0} \Delta x / \Delta t = dx/dt$$



So, is  $\Delta x = dx$ ??

What about velocity?

Is  $\Delta v = dv$ ??

Solving problems: The easy way

- Finite Difference Method (Euler method)

$$\lim_{h \rightarrow 0} [f(x+h) - f(x)] / h = f'(x)$$

The approximation involves dropping the limit.

So...

$$[f(x+h) - f(x)] / h = f'(x)$$

This formula works well for small values of h. A fine grained simulation with several, small steps of h is more accurate.

The Advantage:

1. Very general technique. It works as long as each  $n^{\text{th}}$  derivative can be expressed as a function of the lower order (n-1) derivatives.
2. Fast, and simple. The fastest way of solving diff. eqns.



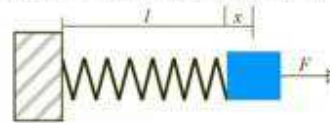
## Springs

A spring is an object that can be temporarily deformed.

It exerts a force proportional to this deformation, and tries to get back to its natural state.



"As the extension, so the force"



$$F = -kx - dv$$

$k$  = Spring Constant

$d$  = Damping coefficient

$x$  = Extension of the spring

$F$  = Force exerted by spring on its ends

$v$  = velocity of motion of the end point

Solving problems: The hard way

- 4<sup>th</sup> order Runge-Kutta:

Assume the initial value problem:

$$y' = f(t, y), \text{ and } y(t_0) = y_0$$

Then, the numerical solution is given by:

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

Here  $k_1, k_2, k_3$  and  $k_4$  are slopes at the beginning, middle and end of the interval 'h'. These slopes are averaged by giving higher weights to the slopes at the middle.



Continuous Simulation

Is it possible to simulate continuous physics on a computer?

Newton's Laws:

$$v = u + a * t$$

$$s = u * t + 0.5 * a * t^2$$

Simply calculate the values of v and s, and keep incrementing t by the required time scale.

Isn't this simpler?

It is simpler, but with these problems:

1. Closed form solution needed for the system.
2. How do we handle discontinuities in space?

$$x(t) = \sqrt{x_0^2 + (v_0/c)^2} \sin(ct + \tan^{-1} \frac{cx_0}{v_0}) \quad \text{Where } c = \sqrt{k/m}$$

# Slides

## Collision Detection

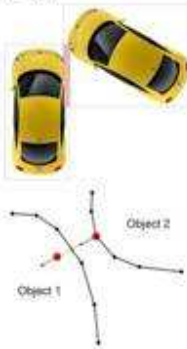
It involves checking at each time instant, for overlapping or intersecting objects.

### Broad Phase:

Here, each object is approximated by a "Bounding Box". These boxes are checked repeatedly for distance between each other and overlaps.

### Narrow Phase:

The position of each point is tracked with respect to each wall. When the point gets too close to the wall, or is going to cross it, a collision occurs.



## Restitution and friction

### Handling Collision:

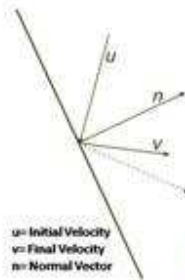
**Step 1:** Separate the components of velocity parallel and perpendicular to the wall. Let these components be  $V_p$  and  $V_n$ .

**Step 2:** Multiply  $V_p$  by a Friction Factor.

**Step 3:** Multiply  $V_n$  by a Restitution Factor.

The final velocity is the sum of these:

$$V_{final} = V_p * friction + V_n * restitution$$



## Dots and crosses

A refresher on Vector basics:

Let  $v1$  and  $v2$  be vectors in 3D space.

They have components in x, y and z axes.

### Dot Product:

$$v1.x * v2.x + v1.y * v2.y + v1.z * v2.z$$

Scalar

### Cross Product:

$$\begin{aligned} x &= v1.y * v2.z - v1.z * v2.y \\ y &= v1.z * v2.x - v1.x * v2.z \\ z &= v1.x * v2.y - v1.y * v2.x \end{aligned}$$

Vector

## Soft Body Dynamics

A simplified version of the Finite Element Method can be used to simulate soft bodies, that can deform.

This technique involves approximating the entire object as a mesh consisting of several nodes, each having mass, and connected together by springs.

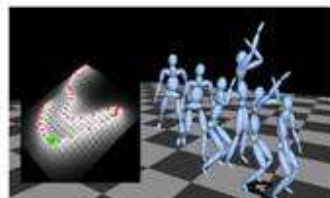


## Inverse Kinematics

Inverse Kinematics (IK) is the process of determining the parameters of a jointed flexible object (a kinematic chain) in order to achieve a desired pose.

IK techniques are used in Robotics, to move the arms (joints) of a robot to the required position.

Inverse kinematics are also relevant to game programming and 3D animation.



## Bibliography

- <http://www.mypysicslab.com>
- [http://en.wikipedia.org/wiki/Numerical\\_ordinary\\_differential\\_equations](http://en.wikipedia.org/wiki/Numerical_ordinary_differential_equations)
- <http://www.harveycartel.org/metanet/tutorials/tutorialA.html> (object collisions)
- [http://www.sv.vt.edu/classes/MSE2094\\_Notebook/97ClassProj/num/widas/history.html](http://www.sv.vt.edu/classes/MSE2094_Notebook/97ClassProj/num/widas/history.html) (Finite Element Method)
- <http://www.ageia.com/> (Ageia PhysX SDK and Hardware - nVidia)
- <http://www.havok.com/> (Havok Physics Engine - Intel)
- <http://www.pixluxentertainment.com/> (Digital Molecular Matter)