Computer Simulation of Projectile Motion

Arlen Cox  
Computational Physics  
February 12, 2004

1 Introduction

This document is the presentation of results of computer simulations of projectile motion. The problem is stated as such:

A cannon shoots a cannon ball at 700 m/s. Given ideal conditions, what is the farthest the ball can travel? How high can it go? Which angles produce these results?

How do non-ideal conditions affect the flight of the cannon ball? What effect does air resistance have on the ball? How much does altitude affect air resistance?

These questions will be answered throughout this document.

2 Theory

2.1 Ideal Case

An object’s motion can be described using the following differential equation:

\[ F = m \frac{d^2y}{dt^2} \]  

(1)

This type of motion can be directly applied to projectile motion by using applying this to a 2D motion.

\[ \vec{F} = m \frac{d^2 \vec{y}}{dt^2} \]  

(2)

A falling object can then be described simply in terms of gravity since

\[ F = mg \]  

(3)

\[ mg = m \frac{d^2 \vec{y}}{dt^2} \]  

(4)

\[ g = \frac{d^2 \vec{y}}{dt^2} \]  

(5)
Solving this differential equation for the vertical motion using \( y(0) \) for the initial height and \( y'(0) \) for the initial velocity gives

\[
y(t) = \frac{1}{2} gt^2 + y'(0)t + y(0)
\]  
(6)

From this a maximum height for an initial value of 700 m/s can be calculated by taking the derivative and setting it equal to zero (where the velocity will be zero).

\[
\frac{dy}{dt} = gt + y'(0)
\]  
(7)

\[
0 = -9.81t + 700 \quad \text{or} \quad t = 71.35s
\]  
(8)

\[
y_{max} = 24974.5m
\]  
(9)

To calculate the maximum distance travelled a combination of horizontal velocity and vertical velocity must contrived. Since time to hit the ground is twice that of the time it take to reach the peak the time to ground must be

\[
t = 2\frac{-v_y(0)}{g}
\]  
(11)

The distance the projectile will travel will be

\[
d = v_x(0)t
\]  
(12)

It follows that the set of equations to be maximized is as follows

\[
d = 2v_x(0)\frac{-v_y(0)}{g}
\]  
(13)

\[
v_x(0) = 700\cos(\theta)
\]  
(14)

\[
v_y(0) = 700\sin(\theta)
\]  
(15)

To find the maximum distance the derivative must be taken and set equal to zero.

\[
\frac{d}{d\theta}\left(1400\cos(\theta)\frac{-700\sin(\theta)}{g}\right) = 0
\]  
(16)

Solving this yields \( \theta = 45 \).

### 2.2 Air Friction

The addition of air friction changes the force equation from simply \( F_{air} = mg \) to:

\[
\vec{F}_{air} = -c\rho A_{OB}JV^2\dot{V}
\]  
(17)
Using this property the system can be represented as a series of four single order differential equations:

\[
\begin{align*}
\frac{dV_x}{dt} &= -B_2 |V| V_x \quad (18) \\
\frac{dV_y}{dt} &= -B_2 |V| V_y \quad (19) \\
\frac{dx}{dt} &= V_x \quad (20) \\
\frac{dy}{dt} &= V_y \quad (21)
\end{align*}
\]

where \( B_2 = c \rho A_{OBJ} \).

### 2.3 Altitude Considerations

The addition of altitude into the equation adds thinning air into the equation. This simply changes the \( \rho \) value to count for altitude changes. In this case the force value is:

\[
F_{\text{air}} = -c \rho e^{-\frac{\mu}{\rho}} A_{OBJ} V^2 \dot{V} \quad (22)
\]

This simply makes the differential equations:

\[
\begin{align*}
\frac{dV_x}{dt} &= -B_2 e^{-\frac{\mu}{\rho}} |V| V_x \quad (23) \\
\frac{dV_y}{dt} &= -B_2 e^{-\frac{\mu}{\rho}} |V| V_y \quad (24) \\
\frac{dx}{dt} &= V_x \quad (25) \\
\frac{dy}{dt} &= V_y \quad (26)
\end{align*}
\]

### 3 Simulations

#### 3.1 Implementation

The simulations performed simulate the three above situations (Sections 2.1, 2.2, 2.3). The simulations were set up using a first order euler method and a second order Runga-Kutta method.

The differential equations were set up with the following code for the first order euler method:

```c
void simulate1st(object *o, double dt, int friction, int altitude) {
    o->v.x += fx(*o, friction, altitude)*dt;
}
```
The differential equations were set up similarly using the midpoint (Runga-Kutta 2nd order) method:

```c
void simulate2nd(object *o, double dt, int friction, int altitude)
{
    object temp;
    memcpy(&temp, o, sizeof(object));
    temp.v.x += fx(temp, friction, altitude)*dt/2;
    temp.v.y += fy(temp, friction, altitude)*dt/2;
    temp.l.x += temp.v.x*dt/2;
    temp.l.y += temp.v.y*dt/2;
    o->v.x += fx(temp, friction, altitude)*dt;
    o->v.y += fy(temp, friction, altitude)*dt;
    o->l.x += temp.v.x*dt;
    o->l.y += temp.v.y*dt;
}
```

The $F_X(X,Y,V_X,V_Y)$ function was implemented as follows to take advantage of the various options:

```c
double fx(object o, int friction, int altitude)
{
    double force = o.m*0;
    double thisB2 = o.B2;
    if(altitude)
    {
        thisB2 = o.B2*exp(-o.l.y/10000.0);
    }
    if(friction)
    {
        if(o.v.x > 0)
            force += -thisB2*magnitude(o.v)*o.v.x;
        else
            force += thisB2*magnitude(o.v)*o.v.x;
    }
    return force;
}
```

For timesteps in simulation, the known ideal case is verified using the simulation, using timesteps small enough to eliminate most of the error. In the case of the 2nd order, this can be with a timestep of one second, but with a timestep that large the simulation is not smooth, so a smaller timestep will be used.
3.2 Verification

The simulations described above need to be verified using the ideal case to be shown valid. The ideal case (Section 2.1) can be compared to the simulation results by plotting them together on a single graph. This can be seen in figure 1.

As can be seen in Figure 1, a timestep of .1 yields reasonable results, whereas larger timesteps introduce a large amount of error.

When using the second order method, the results are much different. This demonstrates the improvement in the algorithm. Figure 2 shows the same timesteps as the first order only with the second order algorithm.

Table 1 shows how much error each method has for each timestep.

4 Results

Taking friction and altitude into account demonstrates how inaccurate to reality the ideal calculations are. Figure 3 shows the ideal shot at 45 degrees and 700 m/s and the same shot including friction is much different. It also shows that when the variation in air thickness with altitude is included the results change again, but less so than from ideal to friction.
Figure 2: Shows the improvements gained by using a second order algorithm and shows the timesteps needed to produce accurate results.

With the added friction the maximum distance is achieved at 38.17 degrees instead of 45 degrees. This occurs due to a more optimal travel not being that of high altitude. However, when air thickness is taken into account (depending on altitude), altitude becomes much more beneficial and increases the angle for maximum distance to 45.93 degrees. The trajectories for various angles can be seen in Figures 4 and 5. The program used to calculate these maximum distances can be seen in Section 6.2.

5 Conclusion

This experiment using computers to calculate trajectories shows how inaccurate simulations can be if even one first order effect is not counted. The differences evident in Figure 3 are so huge that any real simulation couldn’t avoid these without completely destroying any validity of results. This really demonstrated why it is important to compare simulation results with a known result to ensure correctness. If any other elements have been left out of this simulation, it could be very incorrect. This would be the case if the projectile were shot so high that gravity would change, or if the curve of the earth should be taken into account for calculating distance, or if the wind were blowing. This simulation deals with
<table>
<thead>
<tr>
<th>$\Delta t$</th>
<th>Ideal</th>
<th>1st Order</th>
<th>2nd Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>10s</td>
<td>12,487.25m</td>
<td>10,033.73m</td>
<td>12,486.23m</td>
</tr>
<tr>
<td>1s</td>
<td>12,487.25m</td>
<td>12,240.98m</td>
<td>12,486.23m</td>
</tr>
<tr>
<td>.1s</td>
<td>12,487.25m</td>
<td>12,462.52m</td>
<td>12,487.24m</td>
</tr>
<tr>
<td>.01s</td>
<td>12,487.25m</td>
<td>12,484.78m</td>
<td>12,487.25m</td>
</tr>
<tr>
<td>.001s</td>
<td>12,487.25m</td>
<td>12,487.01m</td>
<td>12,487.25m</td>
</tr>
</tbody>
</table>

Table 1: This table shows the convergence of the two algorithms to the target ideal. They show the maximum height of each shot using each method.

none of these cases, so it is still somewhat an ideal simulation, but it does at the very least provide insight into what would happen in ideal conditions.

6 Source Code Listing

6.1 projectile.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>

/*global for gravity */
double g = -9.81;

/* vector struct stores a 2-D vector */
struct s_vector
{
    double x;
    double y;
};
typedef struct s_vector vector;

/* object struct stores a projectile's attributes */
struct s_object
{
    vector v;
    vector l;
    double m;
    double A;
    double B2;
};
typedef struct s_object object;

/* simulate performs the differential equations */
void simulate1st(object *o, double dt,int friction,int altitude);
void simulate2nd(object *o, double dt,int friction,int altitude);

/* returns the magnitude of a vector */
double magnitude(vector v);

/* polartocart converts polar to cartesian coordinates */
vector polartocart(double r, double t);

/* degtorad converts degrees to radians */
double degtorad(double d);

int main(int argc, char** argv)
```

7
Figure 3: This shows three shots all at 700 m/s and 45 degrees incline. One is the ideal case. One includes air friction and one includes air thickness variations.

```c
/* declare variables */
int printstep;
object o;
double dt;
double t;
int count;
int friction;
int altitude;
int i;
int gotcount;
double velocityval;
double maxheight;
int order;
/* initialize variables */
gotcount = 0;
count = 0;
printstep = 100;
o.l.x = 0;
o.l.y = 0;
o.v = polartocart(700.0,degtorad(45.0));
/* must be set to 1 for correct results */
o.m = 1;
o.A = 3;
o.E2 = 4e-5;
dt = .001;
t = 0.0;
friction = 0;
altitude = 0;
maxheight = 0.0;
```
Figure 4: This shows the effect of changing the angle of launch in a friction environment.
Figure 5: This shows the effect of changing the angle of launch in a friction environment taking into account changes in air density with altitude.
```c
} else if(!strcmp("-t",argv[i]))
{
  i++;
  if(i < argc)
  {
    dt = atof(argv[i]);
  } else {
    printf("invalid arguments\n try -u for help\n");
    exit(0);
  }
} else if(!strcmp("-o",argv[i]))
{
  i++;
  if(i < argc)
  {
    order = atoi(argv[i]);
  } else {
    printf("invalid arguments\n try -u for help\n");
    exit(0);
  }
} else {
  /* it is a number */
  if(gotcount == 0)
  {
    velocityval = atof(argv[i]);
    gotcount++;
  } else if(gotcount == 1)
  {
    o.v = polartocart(velocityval,degtorad(atof(argv[i])));
    gotcount++;
  }
}
if(gotcount != 2 && gotcount != 0)
{
  printf("invalid arguments\n try -u for help\n");
  exit(0);
}
printf("# initial conditions\n");
printf("# initial velocity (x,y),o.v.x,o.v.y\n");
printf("# friction: %d\n", friction);
printf("# altitude air thickness: %d\n", altitude);
if(friction)
  printf("# B2 constant: %f\n", o.B2);
printf("\n\n# x value\ty value\ttime\tx velocity\ty velocity\n");
/* print out the initial values */
printf("%.10f\t%.10f\t%.10f\t%.10f\t%.10f\n", o.l.x,o.l.y,t,o.v.x,o.v.y);
/* start euler loop */
while(o.l.y >= 0)
{
  /* do a timestep */
  if(order == 2)
    simulate2nd(&o,dt,friction,altitude);
  else if(order == 1)
    simulate1st(&o,dt,friction,altitude);
  printf("\n\n\n# x value\ty value\ttime\tx velocity\ty velocity\n");
```
/* output data to file */
if(count % printstep == 0)
{
    printf("%.10f\t%.10f\t%.10f\t%f\t%f\n",o.l.x,o.l.y,t,o.v.x,o.v.y);
    count = 0;
}

if(o.l.y > maxheight)
    maxheight = o.l.y;

/* increment time */
count++;
t += dt;

printf("# maximum range = %f\n",o.l.x);
printf("# maximum height = %f\n",maxheight);
return 0;
}

/* calculates friction in the x direction */
double fx(object o,int friction,int altitude)
{
    double force = o.m*0;
    double thisB2 = o.B2;

    if(altitude)
    {
        thisB2 = o.B2*exp(-o.l.y/10000.0);
    }

    if(friction)
    {
        if(o.v.x > 0)
        {
            /* force += -o.B2*sqrt(o.v.y*o.v.y+o.v.x*o.v.x)*o.v.x; */
            force += -thisB2*magnitude(o.v)*o.v.x;
        }
        else
        {
            /* force += o.B2*sqrt(o.v.y*o.v.y+o.v.x*o.v.x)*o.v.x; */
            force += thisB2*magnitude(o.v)*o.v.x;
        }
    }

    return force;
}

/* calculates friction in the y direction */
double fy(object o,int friction,int altitude)
{
    double force = o.m*g;
    double thisB2 = o.B2;

    if(altitude)
    {
        thisB2 = o.B2*exp(-o.l.y/10000.0);
    }

    if(friction)
    {
        if(o.v.y > 0)
        {
            /* force += -thisB2*sqrt(o.v.y*o.v.y+o.v.x*o.v.x)*o.v.y; */
            force += -thisB2*magnitude(o.v)*o.v.y;
        }
        else
        {
            /* force += thisB2*sqrt(o.v.y*o.v.y+o.v.x+o.v.x)*o.v.y; */
            force += thisB2*magnitude(o.v)*o.v.y;
        }
    }

    return force;
}

/* returns the magnitude of a vector */
double magnitude(vector v)
```c
{ return sqrt(v.x*v.x + v.y*v.y); }

/*/ does a single timestep simulation using a first order method */
void simulate1st(object *o, double dt, int friction, int altitude)
{
    o->v.x += fx(*o, friction, altitude)*dt;
    o->v.y += fy(*o, friction, altitude)*dt;
    o->l.x += o->v.x*dt;
    o->l.y += o->v.y*dt;
}

/*/ does a single timestep simulation using a second order method */
void simulate2nd(object *o, double dt, int friction, int altitude)
{
    /* l is the location (x,y) 
    v is the velocity (x,y) 
    a is the acceleration (x,y) */
    /* perform midpoint method */
    /* calculate d* based on dt/2 */
    /* use that dt to find y */
    object temp;
    /* copy o to temp for modification */
    memcpy(&temp, o, sizeof(object));
    temp.v.x += fx(temp, friction, altitude)*dt/2;
    temp.v.y += fy(temp, friction, altitude)*dt/2;
    temp.l.x += temp.v.x*dt/2;
    temp.l.y += temp.v.y*dt/2;
    o->v.x += fx(temp, friction, altitude)*dt;
    o->v.y += fy(temp, friction, altitude)*dt;
    o->l.x += temp.v.x*dt;
    o->l.y += temp.v.y*dt;
}

/*/ converts polar coordinates to a cartesian vector */
vector polartocart(double r, double t)
{
    vector v;
    v.x = r*cos(t);
    v.y = r*sin(t);
    return v;
}

/*/ converts degrees to radians */
double degtorad(double d)
{
    return d*M_PI/180.0;
}

6.2 maxdist.pl
#!/bin/perl
use strict;
my $angle = 1;
my $angleend = 89;
my $increment = 1;
my @line;
my $maxdist = 0;
my $maxangle = 0;
my $args = "-f -o 2 -t .0001 -p 100000 | tail -2 | head -1";
while($increment > .0001)
{
    for($angle -= $angleend; $angle = $angle+$increment)
{ open(MYFILE, "./projectile 700 $angle $args |");
    @line = split("\n", <MYFILE>);
    if(0 + $line[1] > $maxdist)
    {
        $maxdist = 0 + $line[1];
        $maxangle = $angle;
    }
    my $dist = 0+ $line[1];
    print "$angle\t$dist\t$increment\n";
    close(MYFILE);
}
$angle = $maxangle - $increment;
$angleend = $maxangle + 2 * $increment;
$increment = $increment / 10;
print "Angle for Maximum Distance: $maxangle\n";