APPM 3050 Spring 2014
Andrew Gordon
Evan Roncevich
4/2/14

Introduction to Projectile Motion

1. *Introduction*

This project and the following code is a simulation of projectile motion in two dimensions. The projectile will be "fired" from a constant location in the Cartesian plane and the angle of firing will be adjusted via bisection until a target is hit. The target's virtual size is determined by a tolerance of error that will yield an accuracy in radians out to 5 significant figures. Calculating the path of a projectile will involve the integration of 4 coupled differential equations, taking into account both air resistance and wind.

The general method of solving this problem is to create a boundary value problem given our initial conditions and our ending conditions of some (X_T, Y_T). Then, we will use our X_T as a line over which to use bisection on our 2 initial guesses of θ_1, θ_2. Using Matlab's `ODE45` method of integration we will continue until we have either satisfied our tolerance of convergence or have determined we are unable to hit the target.

2. *Deriving initial conditions*

The motion of our projectile is determined by 4 coupled differential equations for the projectile: $\dot{x}, \dot{y}, \dot{\theta}, \dot{v}$. To create our boundary conditions we will use the following:

$$x_0 = 0$$
$$y_0 = 0$$
$$\theta_0 = arctan(\frac{v^2 \pm \sqrt{v^4 - g(g * x^2 + 2 * y * v^2)}}{g * x})$$
$$v_0 = 1600 m/s$$

$$x_{final} = X_T$$
$$y_{final} = Y_T$$

3 of these initial conditions are very simple: $(x, y) = (0, 0)$ denotes the origin, and v_0 is our initial firing velocity. Because we are using bisection, we need two initial θ, a known undershot and overshot. To determine an ideal initial shot, we can use the standard equation for projectile motion.

$$x = t * v * cos(\theta)$$

$$y = -.5 * g * t^2 + v * sin(\theta) * t$$
$$t = \frac{x}{v * cos(\theta)}$$

Through several derivations:

$$y = \frac{-g * x^2}{2 * v^2 * cos^2(\theta)} + x * tan\theta$$

$$tan(\theta) = \frac{v^2 \pm \sqrt{v^4 - g(g * x^2 + 2 * y * v^2)}}{g * x}$$

The θ derived represents the angle required to hit a target given there is no air resistance or wind. Because the equation for $\tan(\theta)$ has $\pm$, there are two roots present, meaning all targets can be hit in two ways with a lob shot or a direct shot dependent on which sign is used. Because air resistance is present and wind is not significant to disrupt the projectile flow, shooting at these two θ will always yield an undershot. The only possible location for an overshot is between the two angles, meaning the necessary angles to hit the target lie between the two derived angles. As a direct shot has a shorter path, meaning less computation time, the initial guesses take the lower bounded θ as a known undershot and the center angle between the upper and lower bound as a known overshot given the target is within range.

3. *Deriving equations for motion: The system without wind*

We will show the derivation for the system in the TNB reference frame without wind first and then add the necessary components to measure motion with a wind vector. The 4 coupled equations are:

$$\dot{x} = vcos(\theta)$$
$$\dot{y} = vsin(\theta)$$
$$\dot{v} = \frac{-C_d v^2}{m} - gsin(\theta)$$
$$\dot{\theta} = \frac{-g}{v}cos(\theta)$$

For a change to polar coordinates, we model the a particle using a triangle starting at the origin and along the x axis. The distance in the x direction is x; the height is y. We can easily find these quantities using $sin(\theta) = y/r$ and $cos(\theta) = x/r$. Thus for position,

$$\dot{x}(t) = \dot{r} * cos(\theta)$$
$$\dot{y}(t) = \dot{r} * sin(\theta)$$

and we know that $\dot{r} = v$, thus:

$$\dot{x}(t) = v * cos(\theta)$$
$$\dot{y}(t) = v * sin(\theta)$$

Solving for $\hat{\mathbf{i}}, \hat{\mathbf{j}}, \dot{v}$ and $\dot{\theta}$ in the TNB frame of reference:

In two dimensions the TNB frame involves the normal component and tangential component. To provide a change of coordinates from cartesian $\hat{\mathbf{i}}$ and $\hat{\mathbf{j}}$, we look at the projections of the two vectors onto the tangential and normal vectors, and using trigonometry find that:

$$\hat{\mathbf{i}} = cos(\theta)T - sin(\theta)N$$
$$\hat{\mathbf{j}} = sin(\theta)T + cos(\theta)N$$

Therefore, we plug these two equations into our original formula for motion:

$$v(t) = v * cos(\theta(t))\hat{\mathbf{i}} + v * sin(\theta(t))\hat{\mathbf{j}}$$
$$v(t) = v * cos(\theta(t))[cos(\theta)T - sin(\theta)N] + v * sin(\theta(t))[sin(\theta)T + cos(\theta)N]$$

Before taking a derivative and actually solving for $\dot{v}$ we must also find our forces for drag and gravity. As the projectile travels along its arc, it experiences a pull down in the $\hat{\mathbf{j}}$ direction and a pull opposite to its tangent vector T in the form of air resistance, with the air resistance written as the coefficient of drag multiplied by the velocity squared. With this in mind, the sum of the forces acting upon the projectile are:

$$\sum F = F_{drag} + F_{grav} = -mg\hat{\mathbf{j}} - C_d v^2 T = -mgsin(\theta)T - mgcos(\theta)N - C_D v^2 T$$

Now, we can take a derivative of our equation for motion with respect to θ and v, yielding:

$$\frac{d\theta}{dt}[mv] = v * cos(\theta(t))\dot{\theta} - v * sin(\theta(t))\dot{\theta} = \dot{\theta}[v * cos(\theta)\hat{\mathbf{j}} - v * sin(\theta)\hat{\mathbf{i}}]$$

$$\frac{dv}{dt}[mv] = -mgsin(\theta)T - C_d v^2 T$$

4. *Deriving equations for motion: The system with wind*

With the addition of some vector field representing wind, the equations for $\dot{v}$ and $\dot{\theta}$ must be adjusted:

$$\dot{x} = vcos(\theta)$$

$$\dot{y} = vsin(\theta)$$
$$\dot{v} = \frac{-C_d}{m}|V_{\frac{p}{a}}|(v - \alpha cos(\theta) - \beta sin(\theta)) - gsin(\theta)$$
$$\dot{\theta} = \frac{-Cd}{mv}|V_{\frac{p}{a}}|(\alpha sin(\theta) - \beta cos(\theta)) - \frac{g}{v}cos(\theta)$$

For these new equations, we need to calculate two things: Velocity of projectile relative to the air, and the velocity of the wind relative to the ground. These will be written as $V_{\frac{p}{a}}$, and $V_{\frac{a}{0}}$.

$$V_{\frac{a}{0}} = \alpha\hat{\mathbf{i}} + \beta\hat{\mathbf{j}}$$
$$V_{\frac{p}{a}} = \sqrt{(vcos(\theta) - \alpha)^2 + (vsin(\theta) - \beta)^2}$$

The first equation is simply the $\hat{\mathbf{i}}$ and $\hat{\mathbf{j}}$ components of the wind at any given point as they create a vector field relative to any point on the ground. The more complicated second equation represents the magnitude of the drag force, which now must involve the motion of x and y taking into account the wind blowing against the projectile. Finally, in $\dot{v}$ and $\dot{\theta}$ we take into account the velocity of the projectile being diminished by α and β as the projectile travels across the vector field.

5. *Coding: Stop conditions*

   Having elucidated our reasons for our initial guess above, we will go directly into our `ODE45` stop conditions. `ODE45` uses a 4th order Runge-Kutta, which becomes the most CPU-consuming process throughout the program's run-time. Adding stop conditions terminates the differential equation solver when a condition is met, specifically when we are certain whether a given angle is an overshot or undershot. The basis of the stop condition is checking whether an angle yields an undershot. Because gravity will always pull the projectile in the $\hat{\mathbf{j}}$ direction, which neither wind nor air resistance can counter, a projectile with a velocity vector directly at the target will always undershoot. For our purposes, we can rule that if the velocity vector of the target is equivalent to the distance vector, the projectile will undershoot. The undershot condition is derived from the equation for a line, $y - y_0 = m(x - x_0)$. Substituting Y_T and X_T in for y_0 and x_0, as well as the substitution m=$tan(\theta) = y/x$, yields our stop condition: if $tan(\theta)[(X_T - x_{current}) + y_{current} - Y_T] \leq 0$, stop integration. When the condition becomes less than 0, the projectile is no longer aimed over the target. The second condition stops `ODE45` when the projectile reaches the x-coordinate of the target. If the projectile reaches without hitting the first condition, the projectile an overshot and we can stop integrating.

   Now `ODE45` will never compute over the entire arch, instead only to a maximum of the x-position of the target and even shorter if an undershot.

6. *Coding: Bisection* The entirety of our Bisection routine is a simple while-loop governed by a *maxits* variable and a convergence check:

```
while(counter<maxits);

    thm = (th1+th2)/2;
    options = odeset('Events', @(t,x)ControlEvents(t,x,xt,yt), 'Refine', 12, 'R
    [t,x,te,xe,ie] = ode45(@MySystem, [tInit, tFin], [v0 thm 0 0], options);

    if(x(end,3)>=xt)
        th2=thm;

    else
        th1=thm;
    end
    if(abs(th2-thm)<tol)
```

```
            break;
        end
        counter=counter+1;
    end
```

In this loop, we simply check if we have an overshoot or undershoot, adjust $th1, th2$ and thm accordingly, and break if we have achieved our tolerance of convergence. Note that this does not explicitly call a `dmin` function but uses tol to check if the distance from the target is within an acceptable limit. Tolerance here is defined as:

`tol=.0005`

With this value found to give a correct value out to 4 significant figures.

7. *Coding: ODE 45 and MySystem*

`ODE45` is one of numerous numerical methods Matlab provides to integrate a system of equations. To define it, we modify `MySystem.m`, the first argument of `ODE45`. Before optimization, the code is defined as:

```
[a,b]= Wind(x(3),x(4));
v= x(1);
theta = x(2);

sysDot(1) = -.0025*vpa*(v-a*cos(theta)-b*sin(theta))-9.81*sin(theta);
sysDot(2) = -.0025*vpa*(a*sin(theta)-b*cos(theta))/v-9.81/v*cos(theta);
sysDot(3) = v*cos(theta);
sysDot(4) = v*sin(theta);
```

Here we can see the hard coding of elements such as $\frac{C_d}{m}$ and g, as well as accessing v, θ, α and β from our vector of the system and our `wind.m` file. The appendix gives the greater picture of all files working together to find θ_{final}, with the above code being the defining trait of this system of equations.

8. *Coding: Additional optimizations*

As noted in the stop conditions, the 4th order Runge-Kutta is the most computationally-intensize action in the system, meaning any performance improvements in MySystem.m will provide significant decreases in run-time. The most basic optimizations were in replacing all constant variables with the actual value. As shown in the above section,

instead of calling cd or g, the values were hard-coded into the program as 0.0025 and 9.81 respectively. Memory look-ups are far more expensive than hard-coded numbers, making our system have improved performance. Another improvement was saving cos(theta) and sin(theta) as variables so that the cosine and sine functions do not need to be called every time. Instead of calling each cos(theta) and sin(theta) 4 times each, saving them as variables only calls them once. These changes improved the performance dramatically. Other changes included moving complex computations outside of the bisection method to remove repeating calculations.

9. *Additional Features*

It is important for a user to determine whether the returned angle has a degree of confidence in the solution presented. Our program tests can determine if a target is out of range. There are two checks, the first checks if the target can be hit given an drag-less and windless environment, if this is impossible and not checked for, the program would crash as it would attempt to take the square root of a negative number. Along with that, because the initial upper-bound would be an overshot if hitting the target was possible, if the final angle converges on the initial upper-bound, The target cannot be hit. The program will alert user if either exists. Another feature added is that the system can hit a target in both the first and fourth quadrants. The tolerance was chosen by increasing the tolerance of `ODE45` until the angle was within 0.0005 of all solutions. This resulted in a Relative and Absolute tolerance of 0.001 in the odeset options. The bisection would also run until the tolerance is within 0.0005.

10. *Conclusion*

The method described in this report is one of many, many ways to hit a target in two dimensions. For example, we could utilize a different differential equation solver other than `ODE45` that would optimize the code for the specific given conditions of the problem. Likewise, we could use a different method for finding the target other than bisection. If we had a method that converged more quickly, such as Newton's Method, we could increase performance even more. We could even use a different programming technique such as an approach in which we memoize previous answers to save time, or use probability to offset the adverse condition of air resistance which is difficult to predict. We could even write the code in C++, which would likely make it run much faster due to elimination of compile time and hardware-specific optimizations that C offers. Once these techniques are mastered, this could be generalized to real-world situations in which we have to hit targets in 3 dimensions.

11. *Coding: Appendix*

```
%Andrew Gordon
%Evan Roncevich
%Differential equations for ODE45
function sysDot = MySystem(~,x)
    %sysDot= zeros(4,1);
    sint=sin(x(2));
    cost=cos(x(2));
    [a,b]= Wind(x(3),x(4));
    v= x(1);
    vpa=-.0025*sqrt((v*cost-a)^2+(v*sint-b)^2); %includes cd/m
    %sysDot(1) = vpa*(v-a*cost-b*sint)-9.81*sint;%vdot
    %sysDot(2) = (vpa*(a*sint-b*cost)-9.81*cost)/v; %thetadot
    %sysDot(3) = v*cost;
    %sysDot(4) = v*sint;
    sysDot=[vpa*(v-a*cost-b*sint)-9.81*sint; (vpa*(a*sint-b*cost)-9.8
end
```

```
%Andrew Gordon
%Evan Roncevich
%Angle Solver to hit target
function ret=Target(x,y)
    global xt;
    global yt;
    xt=x;
    yt=y;
    v0=1600;
    tInit = 0;
```

```
tFin = 100;
tol=.0005;
counter =0;
if(v0^4-9.81*(9.81*xt^2+2*yt*v0^2)<0);%definitely won't hit, woul
    disp('Failure, out of Range');
    ret=-1;
    return
end

thpart = sqrt(v0^4-9.81*(9.81*xt^2+2*yt*v0^2));%initial boundarie
th1= atan((v0^2-thpart)/(9.81*xt));
th2=atan((v0^2+thpart)/(9.81*xt));
th2=(th1+th2)/2;
thm=0;

options = odeset('Events', @ControlEvents, 'RelTol', 0.001, 'AbsT
overshot=false;
%bisection stuff
while(counter<30);
    thm = (th1+th2)/2;
    if(abs(th2-thm)<tol)%break if tolerance met
        %disp(counter);
        break;
    end
    [t,x] = ode45(@MySystem, [tInit, tFin], [v0 thm 0 0], options
    if(x(end,3)>=xt)%overshot
        th2=thm;
        overshot=true;%undershot
    else
        th1=thm;
```

```
        end
        counter=counter+1;
    end
    if(overshot==false)%should have gotten at least one overshot
        disp('Out of Range');
        ret=-1;
        return
    end
    %disp(thm);
    ret = thm;
end
```

```
%Andrew Gordon
%Evan Roncevich
%Stopping Conditions for ODE45
function[value,stopInteg,direction] =ControlEvents(~,x)%,xt,yt)
global xt;
global yt;
%xt=xi;
%yt=yi;
value(1) = tan(x(2))*(xt-x(3))+x(4)-yt;
value(2) = xt-x(3);

stopInteg(1)=1;
direction(1)=0;
stopInteg(2)=1;
direction(2)=-1;
end
```
