A Brief Introduction to Animation with MATLAB

Dmitry Savransky (dsavrans@princeton.edu)

August 9, 2010

1 Basic Animation Concepts

In this section, we will cover the basics needed to create simple MATLAB animations analogous to the traditional 'cel animation' techniques used in analog animation. This process essentially involves displaying slightly varying images quickly enough so that the brain is able to stitch them together into a seamless illusion of motion. To do this, we need to know:

- Dynamics
- 2D graphics
- Flow control

1.1 Dynamics

The reason we include dynamics in the list of topics required for animation, is that we need a way to describe *how* the objects we're animating are moving. We do this by figuring out the physics of our system, and solving for the equations of motion - topics that land squarely within the field of dynamics. Before you can animate a system, you have to identify its degrees of freedom, define coordinates to describe these, and find the trajectories of these coordinates for some initial conditions. Usually, this will involve numerical integration, but, occasionally, you can find analytical solutions for the trajectories of simpler systems. As a working example, let's consider the spring-pendulum, as shown in Figure 1. This system is composed of a point mass m and a spring with spring constant k and rest length r_0 which is constrained to swing within a plane.

The equations of motion are found quite simply from Newton's second law and are given by

$$\ddot{\theta} = -\frac{2\dot{r}\dot{\theta}}{r} - \frac{g\sin\theta}{r} \tag{1.1}$$

$$\ddot{r} = g\cos\theta + r\dot{\theta}^2 - \frac{k}{m}(r - r_0) \tag{1.2}$$

which we transform into the first order system of ODEs by defining the state

$$X \triangleq \begin{bmatrix} r & \dot{r} & \theta & \dot{\theta} \end{bmatrix}^T = \begin{bmatrix} X_1 & X_2 & X_3 & X_4 \end{bmatrix}^T \tag{1.3}$$

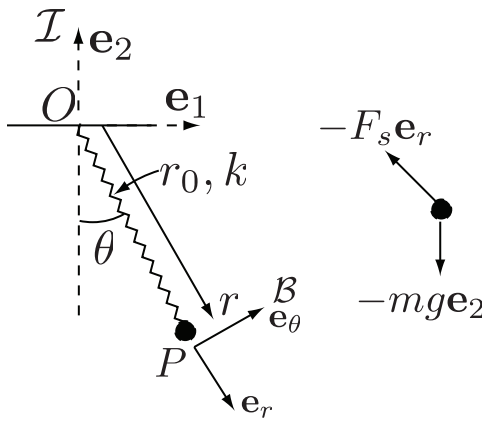

Figure 1: Reference Frames and FBD for the spring-pendulum.

whose first derivative is given by

$$\dot{X} = \begin{bmatrix} \dot{r} \\ r\dot{\theta}^2 + g\cos\theta - \dfrac{k}{m}(r - r_0) \\ \dot{\theta} \\ -\dfrac{2\dot{r}\dot{\theta}}{r} - \dfrac{g\sin\theta}{r} \end{bmatrix} = \begin{bmatrix} X_2 \\ X_2X_4^2 + g\cos X_3 - \dfrac{k}{m}(X_1 - r_0) \\ X_4 \\ -\dfrac{2X_2X_4}{X_1} - \dfrac{g\sin X_3}{X_1} \end{bmatrix}. \tag{1.4}$$

Having found the equations of motion, we now need to numerically integrate them to find the system trajectory. We use MATLAB's `ODE45` since it is able to converge for most systems. Of course, sometimes, you'll want to use one of the other MATLAB integrators, or even one of your own, but `ode45` is usually a good place to start. We will also use the nested function method for passing parameters to our integrator function (like k, m, and r_0). We simply place our integrator function inside of another function in which these variables are defined, as in Listing 1.

Listing 1: Spring-Pendulum integrator.

```
function [t,X] = spring_pendulum_int(tspan,x0,l_0,m,k)

%set constants and defaults
if ~exist('l_0','var'),l_0 = 0.5;end
if ~exist('m','var'),m = 1;end
if ~exist('k','var'),k = 5;end
g = 9.81; %acceleration due to gravity m/s^2

%set initial conditions and time span
if ~exist('x0','var'),x0 = [1,0,pi/4,0];end
if ~exist('tspan','var'),tspan = [0,20];end

%integrate equations of motion
[t,X] = ode45(@springpendulum_eq,tspan,x0);
```

```
15
    %integrator for equations of motion
        function dx = springpendulum_eq(t,x)

            %y = [r, rd, th, thd]
20          r = x(1);
            rd = x(2);
            th = x(3);
            thd = x(4);

25          dx = [rd;
                  r*thd^2 + g*cos(th) - k/m*(r-l_0);
                  thd;
                  -2*rd*thd/r - g*sin(th)/r];
        end
30  end
```

Note that we now have a function which will take a time array, initial conditions, and the three spring-pendulum parameters, and return the spring-pendulum trajectory.

1.2 2D Graphics

The two basic commands we use for 2D graphics are `plot` and `fill`. The former creates 2D scatter and line plots, the latter creates 2D polygons. Both take as inputs arrays of x and y Cartesian coordinates (`fill` also requires an input specifying color).

Example 1.1. Plot shaded and unshaded squares and circles
First, we define the coordinates for a unit circle and square:

```
theta = 0:pi/100:2*pi;
xcirc = sin(theta);
ycirc = cos(theta);
xsquare = [-1 -1 1 1 -1];
ysquare = [-1 1 1 -1 -1];
```

Note that we defined the circle parametrically (via the polar coordinate θ). Also notice that we need to define 5 points for the square to produce a closed figure.

Now, let's draw some shapes:

```
fill(5*xsquare,5*ysquare,'r');
axis square
hold on
fill(4*xcirc, 4*ycirc,'b');
plot(2*xsquare,2*ysquare,'g--');
plot(xcirc,ycirc,'c','LineWidth',2);
hold off
```

Example 1.1 demonstrates how to draw basic geometric shapes. There are four very important things to note here:

1. To draw a geometric shape, you only need to identify its vertices. The appearance of smooth curves (like circles) will depend on how finely you parametrize the shape.
2. The default axes MATLAB creates when you plot something will *not* have an aspect ratio that makes pixels the same length in the x and y dimension. To fix this, you need to use the `axis` command.
3. To put multiple different objects on the same plot, we use the `hold` command.
4. Both `plot` and `fill` can be modified with numerous optional property settings, like 'LineWidth'. For a complete list of these, look in the MATLAB documentation.

Almost any dynamical system can be represented as a collection of lines and basic shapes. For our spring-pendulum example, we can use as little as two: a line representing the spring (pendulum arm) and a circle for the pendulum bob. We can also add an element (for example, a square) to represent the pendulum attachment point.

Example 1.2. Draw a pendulum
We'll use elements from example 1.1 to draw a pendulum with $r = 10$ and $\theta = 45°$.

```
theta = 45*pi/180;
r = 10;
xp = r*sin(theta);
yp = -r*cos(theta);
fill(xsquare,ysquare,'g');
hold on
fill(xcirc+xp, ycirc+yp,'b');
plot([0,xp],[0,yp],'r'); axis square
```

To properly orient everything in your figures, it is important to remember that MATLAB defines the positive x and y directions to the right and up, respectively.

When creating figures, it's very important to take into account how the reference frame we were working with when setting up our dynamics problem corresponds to MATLAB's default axes. For the spring-pendulum, MATLAB's figure axes correspond to $(\mathbf{e}_1, \mathbf{e}_2)$ as shown in Figure 1, whereas we defined our pendulum position in the frame $(\mathbf{e}_r, \mathbf{e}_\theta)$ as $r\mathbf{e}_r$. To properly draw our results, we need a transformation table for the two frames

	$\mathbf{e}_1$	$\mathbf{e}_2$
$\mathbf{e}_r$	$\sin\theta$	$-\cos\theta$
$\mathbf{e}_\theta$	$\cos\theta$	$\sin\theta$

(1.5)

This table gives us the equations used to transform r and θ into Cartesian coordinates used in example 1.2.

1.3 Flow Control

Flow control refers to the ability to repeat commands multiple times or to exhibit different behaviors depending on the values of some variables. Repeating commands is known as looping, and MATLAB contains two types of loops: `for` and `while`

A `for` loop executes a group of commands a specified number of times and is written as:

```
for k = someVector
  % your code here, possibly depending on k
end
% more code
```

k will take on the value of each member of `someVector` (which can be any vector, or operation resulting in a vector), and run through all the code up to `end` using that value. This process will continue until it runs out of elements in `someVector`, at which point it will exit the loop and move on to the rest of the code. Thus, a `for` loop runs as many times as there are elements in `someVector`.

We can use a `for`loop to draw our spring-pendulum system at each time step of the integrator, thus producing motion, but we need one more element: the `pause` command. This tells MATLAB to wait for a given amount of time, allowing us to actually perceive the motion (otherwise, all of the images would flash by too fast to see). Putting all of this together results in the code in Listing 2.

Listing 2: Spring-Pendulum animator.

```
function spring_pendulum_anim(varargin)

%integrate equations of motion
[t,X] = spring_pendulum_int(varargin{:});

%extract length and angle in time
r = X(:,1);
th = X(:,3);

%find position of the pendulum bob:
rm = [r.*sin(th), -r.*cos(th)];

%find bounds:
maxr = max(abs(r));
xmin = min(rm(:,1))-maxr/10;
xmax = max(rm(:,1))+maxr/10;
ymin = min(rm(:,2))-maxr/10;
ymax = max([0,max(rm(:,2))])+maxr/10;

%create and clear figure
h = figure(1);
clf(h);

% create a circle and a square:
a = 0:pi/100:2*pi;
```

```
s = [-1 -1 1 1 -1;-1 1 1 -1 -1];

xscirc = maxr/50*sin(a);
yscirc = maxr/50*cos(a);
xssquare = maxr/50*s(1,:);
yssquare = maxr/50*s(2,:);

%step in time:
for i=1:length(t)
    %plot track so far:
    plot(rm(max([i-100,1]):i,1),rm(max([i-100,1]):i,2),'k--');
    hold on

    %connect mass and pivot:
    plot([0,rm(i,1)],[0,rm(i,2)],'r','LineWidth',2)

    %draw the pivot (centered at origin) and the mass:
    fill(xssquare,yssquare,'g');
    fill(xscirc+rm(i,1),yscirc+rm(i,2),'b');

    %set axes to proper values:
    axis equal;
    grid on;
    axis([xmin xmax ymin ymax]);
    hold off;
    %pause for length of time step
    if i < length(t)
        pause(t(i+1)-t(i));
    end
end
```

Notice how we used the actual distance between integrator time steps in our `pause`command to give our animation a realistic progression of time.

2 Animation via Graphics Objects

Every time you create something in a MATLAB figure, whether it is a line, or filled polygon, or even a new set of axes, you are actually creating an object with specific attributes. All MATLAB objects can be referred to via their handles, which gives us the ability to both get and set an object's properties. This allows for a different type of animation - one where we create the set of objects needed to describe our system, and then update the positions/appearences of those that represent moving parts of the system. This can be much more efficient than the method described in section 1, since it often requires less redrawing.

Every command we've used so far can actually be called with an output, which will return the handle of the created object. We can then access these objects' properties via the `get` command, and change them with the `set` command.

Example 2.1. Creating and handling objects
We'll use elements from example 1.1 to create and manipulate a fill object:

```
clf
h1 = fill(xsquare,ysquare,'g');
axis([-5 5 -5 5]);
get(h1)
```

The last command will produce a list of all of the properties of our fill object. Of particular interest are the 'XData' and 'YData' properties, which control the object's location within the axes. We can change these properties to move the object:

```
set(h1,'XData',xsquare+3,'YData',ysquare+3);
```

Notice that we didn't need to reset our axes, since we didn't add any new graphics objects, but merely moved an existing one. This is another way in which this method is more efficient.

We can modify our first animation to use this technique, resulting in the code in Listing **??**.

Listing 3: Better Spring-Pendulum animator.

```
function spring_pendulum_anim2(varargin)

%integrate equations of motion
[t,X] = spring_pendulum_int(varargin{:});

%extract length and angle in time
r = X(:,1);
th = X(:,3);

%find position of the pendulum bob:
rm = [r.*sin(th), -r.*cos(th)];

%find bounds:
maxr = max(abs(r));
xmin = min(rm(:,1))-maxr/10;
xmax = max(rm(:,1))+maxr/10;
ymin = min(rm(:,2))-maxr/10;
ymax = max([0,max(rm(:,2))])+maxr/10;

%create and clear figure
h = figure(1);
clf(h);

% create a circle and a square:
a = 0:pi/100:2*pi;
s = [-1 -1 1 1 -1;-1 1 1 -1 -1];

xscirc = maxr/50*sin(a);
yscirc = maxr/50*cos(a);
xssquare = maxr/50*s(1,:);
yssquare = maxr/50*s(2,:);
```

```
%create the pendulum components
%draw the pivot (centered at origin) and the mass:
fill(xssquare,yssquare,'g');
hold on
bob = fill(xscirc+rm(1,1),yscirc+rm(1,2),'b');

%connect mass and pivot:
arm = plot([0,rm(1,1)],[0,rm(1,2)],'r','LineWidth',2);

%plot track so far:
track = plot(rm(1,1),rm(1,2),'k--');

%set axes to proper values:
axis equal;
grid on;
axis([xmin xmax ymin ymax]);

%step in time:
for i=1:length(t)
    %update all moving elements
    set(bob,'XData',xscirc+rm(i,1),'YData',yscirc+rm(i,2));
    set(arm,'XData',[0,rm(i,1)],'YData',[0,rm(i,2)]);
    set(track,'XData',rm(max([i-100,1]):i,1),...
        'YData',rm(max([i-100,1]):i,2));

    %pause for length of time step
    if i < length(t)
        pause(t(i+1)-t(i));
    end
end
```