THE MEANS AND MOTIVATION FOR ANIMATING GRAPHICS IN ENGINEERING APPLICATIONS

Oleg Yakimenko

Naval Postgraduate School, Department of Mechanical and Astronautical Engineering,
700 Dyer Rd., Monterey, CA

oayakime@nps.edu (Oleg Yakimenko)

Abstract

The paper deals with animating the results of simulations of the dynamics of different engineering systems in the Mathworks' MATLAB development environment. It first reviews the basics of the handle graphics allowing accessing and dynamically changing any property of any graphics object the user-defined two- or three-dimensional plot might be composed of. It further introduces two methods available in MATLAB to animate these plots. The first one simply redraws the entire plot at each instant of time, captures it and adds to the movie, available to play with later on. The second one might involve more programming but it allows to dynamically vary only the portion of the plot, which is actually changing, leaving the rest of it untouched. If standalone versions of the created movies are needed, the paper presents a way MATLAB suggests to convert them into the standard audio/video interleave files playable outside MATLAB. The paper capitalizes upon pretty basic examples and then advances to several more complex cases, where both methods for creating movies were employed to animate graphics and virtual scenes. Four appendices contain complete professionally-written scripts emphasizing both techniques and teaching some programming tricks. The paper advocates using animations to prove feasibility of simulations and debug user-created programs and is thought to be useful for engineering students and researchers.

Keywords: Animation of systems' dynamics, MATLAB, Handle graphics.

Presenting Author's biography

Oleg Yakimenko received his MS degree in computer science from Moscow Institute of Physics and Technology, Russia in 1986, and his MS degree in aeronautical and astronautical engineering from Air Force Engineering Academy, Moscow, Russia (AFEA) in 1988. He received his first PhD degree in aeronautical and astronautical engineering from the same academy in 1991 and his second PhD degree in operations research in 1996. Dr. Yakimenko has progressed through all professorial ranks at AFEA and is currently teaching and conducting research for the U.S. Navy at NPS. His research interests include fight mechanics; guidance, navigation and control of manned and unmanned vehicles; high-fidelity modeling and real-time applications. He is an author of over 200 publications including several textbooks on programming and digital computations (numerical methods). Prof. Yakimenko is an Associate Fellow of American Institute of Aeronautics and Astronautics and Russian Academy of Sciences of Aviation and Aeronautics.

ISBN 978-3-901608-32-2

1 Introduction

Today the Mathworks' MATLAB/Simulink development environment [1] is widely used all over the world to develop different level of fidelity models and run simulations. This software proved to be very powerful and yet simple to use in different areas including all kinds of engineering.

Not long time ago the only output a student or engineer could rely on was the text output. Then, different languages (Basic, Fortran, Pascal, C) employed on personal computers offered different graphic packages to improve the representability of simulation results and allow more qualitative analysis.

In this respect MATLAB performed a real breakthrough allowing the creation of a wide range of different two- and three-dimensional (2D and 3D) graphics, no other language can possibly think of. Moreover, MATLAB offers an easy programmatic access to any basic drawing element the graphics object might be composed of (Fig. 1). It is possible because each instance of an object is associated with a unique identifier called a handle.

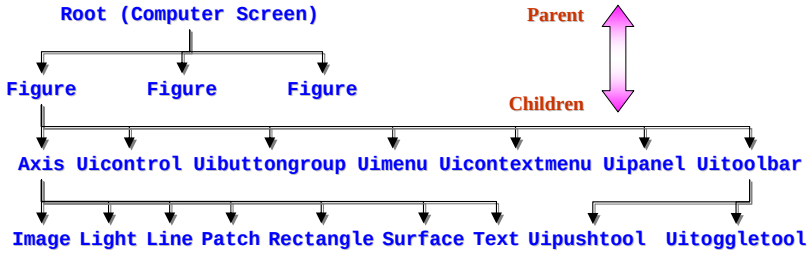

Fig. 1 Graphics objects hierarchy

Everybody knows that the following command

```
plot(sin(0:pi/20:pi))
```

creates a sinusoid. But only few people realize what specter of new possibilities brings the following modification of this command

```
h_line= plot(sin(0:pi/20:pi));
```

creating a handle `h_line` to this sinusoid. Using this handle, a user can easily manipulate the characteristics (called object properties) of the graphics object, sinusoid in this particular case.

Handle graphics opens absolutely new horizons in presenting and analyzing the results of simulations, as well as developing animations. It is widely used by MATLAB developers to create numerous demos (the list of which appears when a user types 'demo matlab' in the MATLAB Command window) and to better explain the essence of numerical algorithms (e.g., in [2]). Yet somehow, handle graphics has not been used widely by students and researches. Among more than 600 textbooks on MATLAB there are few that even mention this valuable feature (more recent ones however, like [3], [4] or [5], do introduce it).

Hence, the goal of this paper is to reintroduce handle graphics to students and practical engineers and show how simple it is to enhance the graphical output to include animations. It will also be advocated that having the outputs of simulations represented as animations rather than band graphics or the static scenes, actually helps to understand the underlying physics and debug the user codes.

The paper is organized as follows. First, Section 2 reminds about the main properties of handle graphics, the system of graphics objects that MATLAB uses to implement graphing and visualization functions. Next, Section 3 introduces the means available in MATLAB to create animations including those built upon handle graphics. Then, Section 4 provides with several examples of animating 2D and 3D graphics and scenes, supported by actual MATLAB M-scripts in Appendices A-D. Finally, Section 5 describes two more examples, where animation occurred to be a very valuable tool for checking the correctness of numerical algorithms and debugging the programs. The paper ends with conclusions.

2 Accessing object properties via its handle

Promoting the 'open source' paradigm, MATLAB allows changing any property of any graphics object, including Figure and its children objects (as shown on Fig. 1), to accommodate user's preferences.

There are three key ideas about the graphics objects a MATLAB user should know to be able to proficiently manipulate with them programmatically:

- graphic objects obey a certain hierarchy (Fig. 1), so that, for instance, Axis happens to be one of Figure's 'children' and simultaneously a 'parent' for a lot of other graphics objects including Line;
- each object has its own handle (even if it was not defined by the user explicitly); and
- knowing object's handle allows accessing (changing) any of its properties.

Consider an example given in the introduction. As mentioned there, the command

```
h_line= plot(sin(0:pi/20:pi));
```

ISBN 978-3-901608-32-2

not only plots the sinusoid (creating a new Figure window and some default Axis automatically), but also creates a handle to this sinusoid, `h_line`. Using this handle we can access all the properties of this Line object. To see what these properties are, one should type `get(h_line)` to obtain

```
                Color: [0 0 1]
            EraseMode: 'normal'
            LineStyle: '-'
            LineWidth: 0.5000
               Marker: 'none'
           MarkerSize: 6
      MarkerEdgeColor: 'auto'
      MarkerFaceColor: 'none'
```

etc. (34 properties in total).

If you are only interested in some specific property you may address it directly, for instance typing `get(h_line,'Color')` results in

```
ans =
     0     0     1
```

(defining red, green, and blue color components of the Line object, so that our sinusoid happens to be blue).

How about accessing the properties of the Axis object? Although we did not created it (it was created automatically for us) and therefore we have no handle to it, we can recall that the Axis object is a parent with respect to the Line object and therefore the command

```
h_axis=get(h_line,'parent')
```

retrieves the Axis handle for us.

Now all Axis properties (133 of them just for the single 2D plot) can be seen by issuing `get(h_axis)` command. As a matter of fact, the `gca` function (stands for 'get current axes') stores the Axis handle automatically, so that the `get(h_axis)` command is equivalent to `get(gca)`.

Similarly, the command

```
h_figure=get(h_axis,'Parent')
```

retrieves the Figure handle. Alternatively, it can be done by issuing any of the following three commands:

```
h_figure=get(gca,'Parent')
```

or

```
h_figure=get(0,'Children')
```

or

```
h_figure=get(0,'CurrentFigure')
```

In addition, the MATLAB `gcf` ('get current figure') function automatically stores the Figure handle too. Therefore, neither of the above four commands is actually needed. However, these simple exercises help to understand how knowing the graphics objects hierarchy (Fig. 1) allows accessing Line properties even without assigning the handle to it explicitly. For instance, once we plotted the sinusoid with

```
plot(sin(0:pi/20:pi))
```

we could use one of the following two commands to retrieve its handle:

```
h_line=get(gca,'children')
```

or

```
h_line=get(get(gcf,'children'),'children')
```

For more efficiency though, for simultaneous manipulations with multiple figures/axes/lines the handle to each object, the properties of which are needed to be changed dynamically, should be assign directly.

Now, using the `set` function we can change any specific property of any object using (referring to) its handle `H`. The general syntax of the `set` function is

```
set(H,'PropertyName',PropertyValue)
```

For example,

```
set(h_line,'Color','r','LineWidth',2.5)
```

changes the sinusoid color to red ([1 0 0]) and increases the line width to 2.5 points (1 point = 1/72 inch).

Knowing these basic features of handle graphics we can now proceed with animations.

3 MATLAB tools for creating animations

MATLAB offers two ways of generating moving, animated graphics. They are so called [1]:

- 'Creating Movies' approach that saves a number of different pictures and then plays them back as a movie; and
- 'Erase Mode' method, which continually erases and then redraws some of the objects on the screen, making incremental changes with each redraw.

Let us briefly consider both of them based on simple examples taken from [6].

3.1 'Creating Movies' approach

We start from the most obvious, old-fashion way of generating animated graphics, which is the 'Creating Movies' approach. The idea here is to simply create each movie frame in advance and then combine them together. The two key MATLAB functions here are:

- `getframe`, capturing movie frame, and
- `movie`, playing recorded movie frames.

The self-explanatory script below provides with an example of using this approach to create animations:

```
% Defining a membrane
 r = [0:0.05:1];     % Radius vector
 phi = 0:pi/20:2*pi;  % Phi angle vector
 x = r*cos(phi);  % x-coordinates of a grid
 y = r*sin(phi);  % y-coordinates of a grid
 z = besselj(1,3.8316*r)*cos(phi);
% Plotting the membrane
 mesh(x,y,z)
 xlabel('x-axis'), ylabel('y-axis')
 zlabel('z-axis'), axis tight
```



```
 set(gca,'nextplot','replacechildren');
% Creating movie frames
 for j = 1:20
    mesh(x,y,sin(2*pi*j/20)*z,z);
    F(j) = getframe;
 end
% Starting the movie
 k=questdlg('Ready to watch the movie?',...
    'Start the Movie', 'Yes', 'No', 'Yes');
% Playing the movie two times
 if char(k(1))=='Y'
    movie(F,2)
 end
```

(hereinafter the key commands responsible for animation are highlighted). This script animates the membrane defined in the first part of it. The MATLAB `mesh` function is used to originally visualize the membrane. The `'nextplot'-'replacechildren'` pair sets current axes to keep their scale, i.e. it removes all child objects, but do not reset axes properties while redrawing the membrane. The animation is done by rescaling the values of z (the z-coordinate on each frame is simply premultiplied by the sinusoidal scale factor that changes smoothly from 0 to 1 and then back to 0). The function `questdlg` is used to create and display the question dialog box). Two of the 20 generated frames are shown on Fig. 2 to give an idea of what happens on the screen.

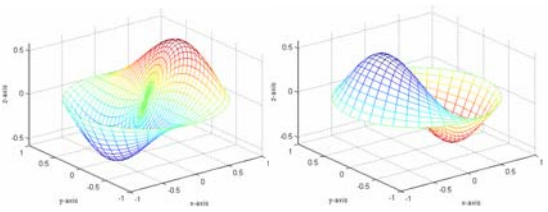

Fig. 2 Frames 5 and 15 of the membrane animation

One of the pitfalls when using the 'Creating Movies' approach is that a movie is not rendered in real-time; it is simply a playback of previously rendered frames. From the other hand, the original drawing time is not important during playback, which is just a matter of blitting the frame to the screen. Therefore, this approach might be better suited to situations where each frame is fairly complex and cannot be redrawn rapidly. Otherwise the 'Erase Mode' approach addressed in the next subsection can be used as well.

3.2 'Erase Mode' method

Another way of creating animations in more elegant way, changing some rather than all graphics objects programmatically, is to use the 'Erase Mode' method.

`EraseMode` is one of the line specifications (offered by the handle graphics) and is very useful and powerful in animation. This property controls the technique MATLAB uses to draw and erase line objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

The default `EraseMode` is `normal` allowing redrawing the affected region of the display, performing the 3D analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other three modes (`background`, `xor` and `none`) are faster, but do not perform a complete redraw and are therefore less accurate, for instance, `none` mode do not erase the line when it is moved or destroyed al all.

For example, let us consider the following set of commands that uses 'Erase Mode' method to slowly convert a sinusoid to cosinusoid:

```
% Plotting a sinusoid
 x=0:0.2:2*pi;   % Defines the x scale
 y=sin(x);       % Computes sin(x)
 z=cos(x);       % Computes cos(x)
 plot(x,y)       % Plots sin(x) curve
 set(gcf,'DoubleBuffer','on');
 set(gca,'xlim',[0 2*pi],'ylim',[-1 1]);
 set(gca,'XTick',[0:pi:2*pi])
 set(gca,'XTickLabel',{'0';'pi';'2pi'})
 xlabel('x'), ylabel('y=f(x)')
% Getting a handle to the line
 h_line=get(gca,'children');
% Changing line properties
 for i=1:1000
    pause(0.005)
% Setting the weighting coefficient w
    w=i/1000;
% Blending sin(x) and cos(x) using w
    d=(1-w)*y+w*z;
% Changing ydata for the line
set(h_line,'ydata',d,'EraseMode','normal');
end
```

In the first four lines of the code we compute data for two dependences, $y = \sin(x)$ and $z = \cos(x)$, and plot the first one (sinusoid). Then, we turn the double buffering on, which helps to produce flash-free rendering for simple animations (such as those involving lines, as opposed to objects containing large numbers of polygons). (Double buffering is the process of drawing to an off-screen pixel buffer and then blitting the buffer contents to the screen once the drawing is complete.). The next line sets the *x* and *y* axes limits. In the two following lines we are also accessing some of the properties of the Axis object. After adding axes' labels we are getting a handle to the line.

Now, what we want to do by the remaining commands is to change the *y*-data for the line, keeping the rest of the properties untouched. Every time we change `ydata`, the previous line is erased (the default value for the `EraseMode` property is set to `normal` anyway, so we just added this property-value pair here to emphasize it). As a result, we will see a smooth conversion of sinusoid to cosinusoid.

Figure 3 shows what you could eventually see on the figure if you would run the above fragment with the `EraseMode` property set to `none`.

The two other options are:

ISBN 978-3-901608-32-2

- `xor`, which draws and erases the image by performing an exclusive OR (XOR) with the color of the screen beneath it (although this mode does not damage the color of the objects beneath the image, the image's color itself depends on the color of whatever is beneath it on the display); and
- `background` that erases the image by drawing it in the axes background color (property `Color` of current `Axis`) or the figure background color (property `Color` of the `Figure`) if the `Axis` color is set to `none` (as opposed to `xor` mode, this does damage objects that are behind the erased image, but images keep their color unchanged).

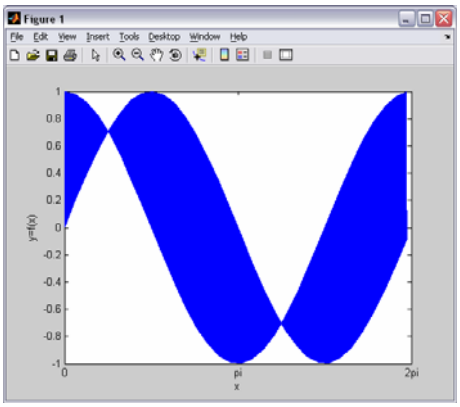

Fig. 3 Snap-shot of the current window after applying `EraseMode none` mode

Note that while all line's transitions are still visible on the screen, you cannot copy this figure or print it, because MATLAB stores no information about its former location. The way we did it on Fig. 3 was taking a snap-shot of the current window with <Alt>+<PrtScrn> keyboard keys. You can also use the MATLAB `getframe` command (discussed in the previous subsection) or any other screen capture applications to create an image of a figure containing the non-normal mode objects.

As mentioned, the 'Erase Mode' method offers programmatic way of animating the results of simulations and therefore is very attractive and powerful. All you have to do is to create a 2D or even 3D scene composed of some objects, get the handles to these objects, and then change their properties (*x*, *y* (*z*) data, color, transparency, etc.). As a matter of fact two of the built-in MATLAB functions use the 'Erase Mode' method to animate graphs by default. They are `comet` and `comet3`. Basically, these two 'dynamic' functions are simply the substitutes for the 'static' `plot` and `plot3` functions, respectively. The comet graphs are animated graphs in which a circle (the comet head) traces the data points on the screen. The comet body is a trailing segment that follows the head. The tail is a solid line that traces the entire function. The script below provides an example of using these two functions (see Fig. 4):

```
subplot(2,1,1)
t = 0:.0005:2*pi;
x = cos(3*t).*(cos(t).^2);
y = sin(3*t).*(sin(t).^2);
comet(x,y);
subplot(2,1,2)
t = -10*pi:pi/1000:10*pi;
comet3((cos(t).^2).*sin(t),...
       (sin(t).^2).*cos(t),t);
```

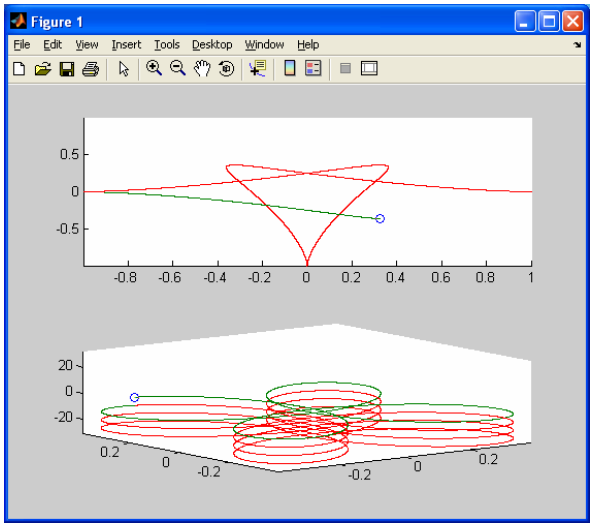

Fig. 4 Snap-shots of the current window when using `comet` and `comet3` functions

The only problem with `comet` and `comet3` functions is that the trace left by a comet is created by using `EraseMode` of `none`, which once again means that you cannot print the graph (you get only the comet head) and it disappears if you cause a redraw (e.g., by zooming, panning, rotating or resizing the window). (Fig. 4 was created by combining two snap-shops when the first and then the second plots were drawn.)

3.3 Creating standalone movies

Obviously, when using any of the aforementioned two methods, 'Creating Movies' or 'Erase Mode', the animation plays within Mathworks' development environment. However, MATLAB offers a way to convert this animation into the standalone movie, specifically into the AVI (Audio Video Interleave) file.

(Despite its limitations and the availability of more modern multimedia formats, like MPEG4, the AVI multimedia format remains popular among file-sharing communities. This is probably due to its high compatibility with existing video editing and playback software like Windows Media Player. Besides, if necessary, the AVI file can be converted later to say MPEG4 format using appropriate codices, for example Xvid or DivX.)

To create and store your animation as a standalone AVI-file, playable outside MATLAB, several functions can be employed. The two key ones are:

- `movie2avi`, creating an AVI movie from MATLAB movie, and

- `addframe`, adding a frame to the current (opened) AVI file.

For instance, if we just add a single line, which is

```
movie2avi(F,'membrane.avi',...
'compression','none','quality',100)
```

after the M-script shown in subsection 3.1 (for the 'Creating Movies' approach) the *membrane.avi* file will be created. Similarly, we can add the line

```
F(i) = getframe;
```

before the end of the `for` loop in the M-script shown in subsection 3.2 (for the 'Erasing Mode' method), and then use for example a command

```
movie2avi(F,'sin2cos.avi',...
'compression','Cinepak','fps',30)
```

afterwards to create the *sin2cos.avi* movie.

An alternative way of creating AVI-file is using the `addframe` function (you will find an example of how to do it in Appendix C). The command

```
aviobj = addframe(aviobj,frame)
```

appends the data in frame to the AVI-file identified by `aviobj`, which must be created beforehand by using `avifile` function. To this end,

```
aviobj = avifile(filename)
```

creates an AVI-file, giving it the name specified in `filename` and using default values for all other AVI-file object properties. If `filename` does not include an extension, `avifile` appends *.avi* to it automatically. The function `avifile` returns a handle to an AVI-file object `aviobj`, which can be used in `addframe` function. Note that once the movie is created, `aviobj` should be closed:

```
aviobj = close(aviobj)
```

You can always retrieve information about your AVI-file using `aviinfo` function

```
aviinfo('filename'),
```

and read the AVI-movie filename back into MATLAB development environment movie structure `mov` using the `aviread` function:

```
mov = aviread(filename)
```

Then you can use the `movie` function again to view the movie `mov`.

Be aware that when you create a standalone movie, the frame height and width will be padded to be a multiple of four as required by majority of codices (MATLAB uses one of the following codices: 'Indeo3', 'Indeo5', 'Cinepak', 'MSVC, 'RLE' or 'None', with 'Indeo5' being the default one). Another warning is that a user should be careful when creating AVI-file on one computer to be played on another one (that another computer might have no codec you created your movie with).

4 Practical examples

This section demonstrates several practical examples of animating the results of simulations. The first example (Fig. 5), deals with analyzing the quality (consistency) of some data associated with image processing, specifically, two frame coordinates of some moving object, *x* and *y* (the problem formulation and the developed algorithms are addressed in [7]). This example employs the 'Erase Mode' method to produce animation.

The *n*-point Welch window runs through the *x* and *y* coordinates of some object in the sequence of frames. The two elongated rectangles on the upper plots (Fig. 5) move simultaneously from left to right through static data (not changing on the graphs). Two bottom plots demonstrate the instantaneous power spectral densities (PSD), so that the spectra diagrams change all the time. If the Welch PSD approaches or drops below some threshold (10^{-5}) (marked with the wide green strip in the bottom portion of the bottom plots), it indicates that something is wrong with the data and those suspicious points should be eliminated (the wide strip itself turns yellow or red, respectively).

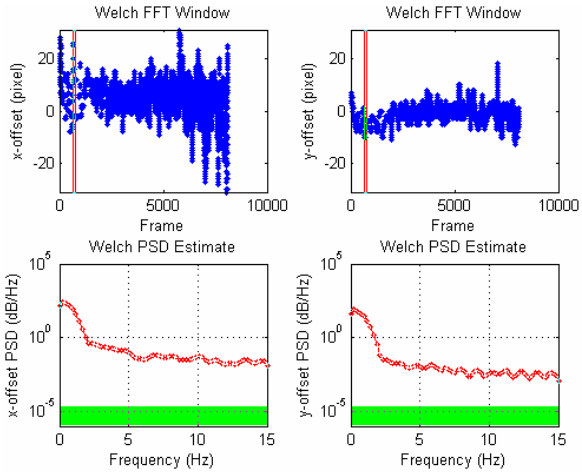

Fig. 5 Checking the consistency of the data

The script in Appendix A shows how it was programmed in MATLAB. The original lines on the first frame were plotted using the `plot` and `semilogy` functions. Then, using their handles some of their properties are continuously altered.

The second example, created using the 'Erase Mode' approach as well, helps to analyze the relative attitude of the ballistic missile and interceptor during impact (Fig. 6) (the problem formulation can be found in [8]). To represent each of two missiles the M-script given in Appendix B uses the `fill` function. Using the simulation results the program rotates the missile silhouettes for each instant of time by changing just a few of their properties accessed via the missile silhouettes' handles. The orientation and magnitude of the speed vectors as well as the text are changing dynamically too (using the handles to corresponding `plot` and `text` commands).

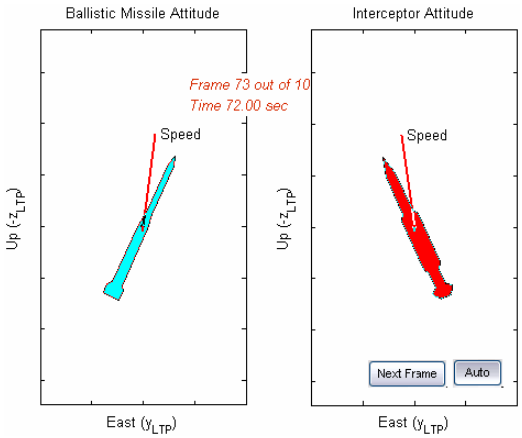

Fig. 6 The intercept geometry animation

Figures 7 and 8 show examples of animation of rendezvous of three robots (see details in [9]). Appendix C contains the complete code, which accepts the parameters of motion of all three robots, along with their thrusters, cameras and lidars orientation, and animates 2D and 3D scenes. Obviously, the code if very bulky, especially for the 3D scene generation (the `patch` function was used to describe each of the robot's side). The proper reordering of the multiple objects to assure correct overlapping is also needed. That is why the 'Creating Movies' approach was used. (For the bird-eye view it could be also done using the 'Erase Mode' approach, i.e. creating the handles to the all moving objects generated with the `fill` function.)

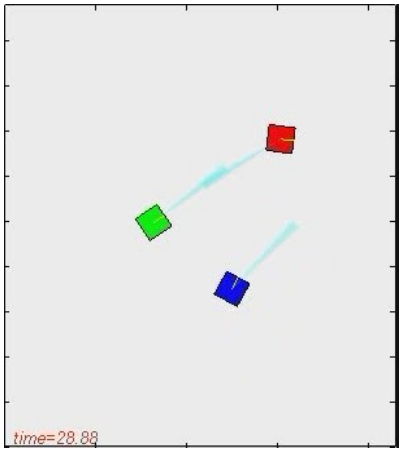

Fig. 7 Bird-eye representation of three robots on a floor with camera beams

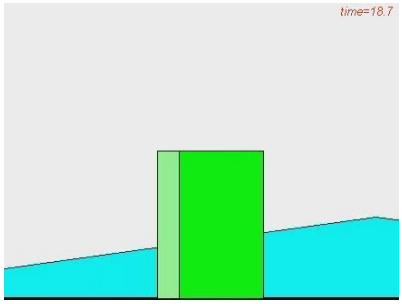

Fig. 8 The 3D visualization of a virtual scene

Finally, Appendix D related to the case considered in the next section contains one more M-script showing an example of how to animate 3D scene using the `patch` function and 'Erase Mode' approach.

5 Cause for animations

Now the question to ask is why we need animations at all? Is it that we simply want to have fun? Well, quite often there is a reason beyond fun. Exploiting powerful and relatively easy-to-use tools provided by MATLAB allows to better understand the underlying physics and sometime effectively debug a user code.

As an example, Figs. 9 and 10 represent two user-created GUIs (graphical user interface) allowing to understand the physics beyond the pose (position and attitude) estimation problem for the descending payload of the aerodynamic delivery system (the details can be found in [10]). (By the way, these GUIs were created using MATLAB GUIDE tool.)

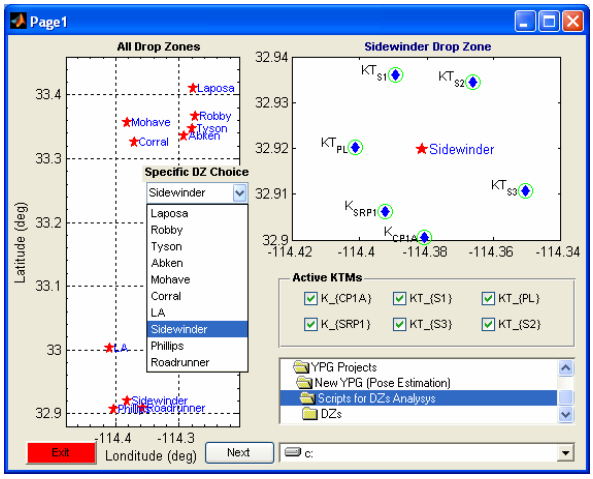

Fig. 9 Interactive graphical user interface

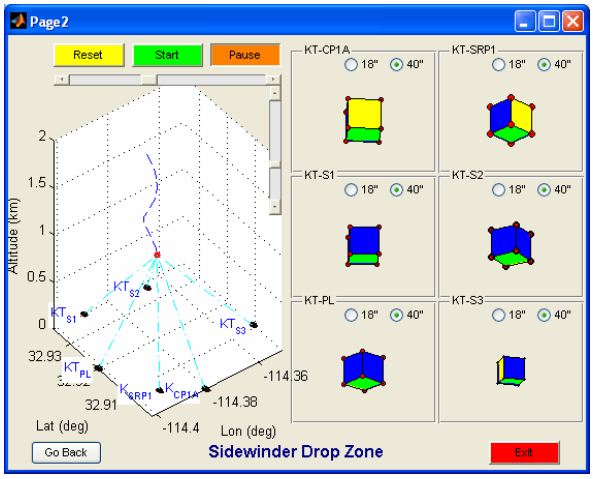

Fig. 10 Aerodynamic delivery system descent analysis

Several cameras (that can be chosen interactively using the first GUI shown on Fig. 9) observe the descending payload and emulate what they would see (Fig. 10). On this second GUI everything is changing dynamically demonstrating what kind of information

could be extracted if the real system were implemented (which would cost tens and hundreds of thousands of dollars to do). Therefore, the developed tool producing realistic animations happens to be very useful. It allows to thoroughly analyze the overall geometry of the experimental setup, choose the most efficient constellation of cameras to use and challenge (test) pose estimation algorithms without costly real drops ([7,10]).

The complete M-script are too long to show them here, however Appendix D demonstrates a small self-contained piece of it exhibiting how to animate the rotating payload. Of course, as shown on Fig. 10, there might be up to six cameras involved, so in order to manage the properties of each 3D view in the most efficient way the original script used arrays of handles.

Finally, Figs. 11 and 12 present an example, where animation allowed finding a small but unfortunate error, which was quite difficult to capture without dynamic 3D representation.

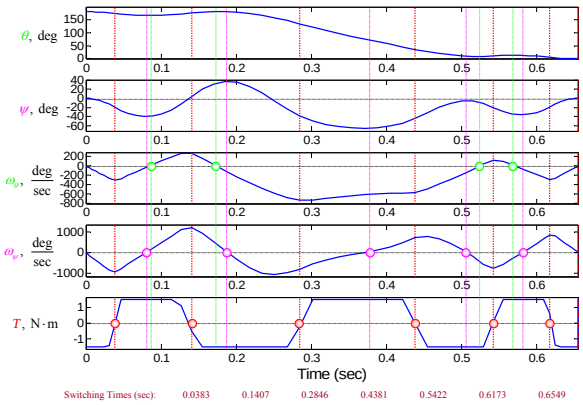

Fig. 11 Example of a multiple band graphic

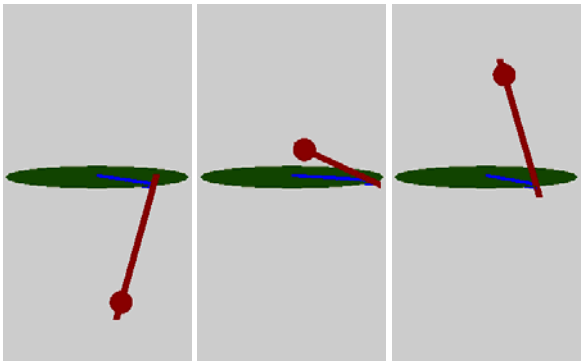

Fig. 12 Control of the inverted pendulum

The parameters of the controlled inverted pendulum shown on Fig. 11 as the usual band plots were considered quite reasonable until the proper 3D animation (using the 'Creating Movies' method) was created (Fig.12). This animation led to the instantaneous conclusion that something was not right. It further allowed to debug the program by simply changing the sign of one state. Not to mention that the time spent on debugging the original program was much greater then the time spent on writing the code for animation.

6 Conclusions

The popular and growing Mathworks' software offers a lot of advanced features allowing to animate the results of simulations and moreover to do it programmatically. The paper shows the main tools allowing to do this and presents some examples. It is thought that using these advanced capabilities not only improves the 'readability' of the results, but might also help in better understanding the underlying physics and debugging a user code. The scripts presented in the paper serve as examples of practical application of some of MATLAB's tools and are supposed to teach some useful techniques and programming tricks. While it does not take much time to explore the advanced capability of MATLAB addressed in this paper, the usefulness of using it will definitely surpass all expectations.

7 References

[1] http://www.mathworks.com.

[2] C. Moler. *Numerical Computing with Matlab*, Society for Industrial Mathematics, 2004.

[3] D. Hanselman and B. Littlefield. *Mastering MATLAB 7*, Prentice Hall, 2004.

[4] S. Chapman. *MATLAB programming for engineers*, 3rd edition, Thomson, 2005.

[5] R. Colgren. *Basic MATLAB, Simulink, and Stateflow*, AIAA Education Series, 2007.

[6] O. Yakimenko. *Introduction to digital computation, AE2440 course notes*, Naval Postgraduate School, Winter, 2006.

[7] O. Yakimenko, V. Dobrokhodov, and I. Kaminer. Autonomous video scoring and dynamic attitude measurement, *18th AIAA Aerodynamic Decelerator Systems Technology conference and seminar*, Munich, Germany, May 23-26, 2005.

[8] J. Lukacs and O. Yakimenko. Trajectory-shape-varying Missile guidance for interception of ballistic missiles during the boost phase, *AIAA Guidance, Navigation and Control conference and exhibit*, Hilton Head, SC, August 20-23, 2007.

[9] B. Eikenberry, O. Yakimenko, and M. Romano. A Vision based navigation among multiple flocking robots: Modeling and simulation, *AIAA Modeling and Simulation Technologies conference*, Keystone, CO, August 21-24, 2006.

[10] O. Yakimenko, R. Berlind, and C. Albright. Status on video data reduction and air delivery payload pose estimation, *19th AIAA Aerodynamic Decelerator Systems Technology conference and seminar*, Williamsburg, VA, May 21-24, 2007.

[11] O. Yakimenko. Direct method for real-time prototyping of optimal control, *International conference Control-2006*, Glasgow, Scotland, August 30 - September 11, 2006.

Appendix A. Example of animating 2D graphics using handles ('Erase Mode' method)

```
function badIndex=fftScan(xData,yData,window,overlap)
% This function accepts two vectors of input data (xData and yData),
% size of the segment for the fft analysis (window), and overlap
% It outputs the indices of the input data where PSD falls below 10e-5
%       To see how it works you can try to run the following two lines:
%          XData=randn(1000,1); XData(30:400)=0*XData(30:400); YData=randn(1000,1);
%          fftScan(XData,YData,128,2)
PSDthreshold=10e-5;
%% Setting-up a two-plot figure window
ysc=max([abs(xData)' abs(yData)']);
figure('Name','Animation of Spectral Analysis for Camera')
i=1;
xOffset=xData(i:i+window);
yOffset=yData(i:i+window);
[Pxx,xww]=pwelch(xOffset,[],[],window,30);
[Pyy,yww]=pwelch(yOffset,[],[],window,30);
    subplot(2,2,1)
    plot(xData,'.b'), hold on
    xwin=plot([i i i+window i+window i],-ysc*[-1 1 1 -1 -1],'r');
    set(xwin,'EraseMode','xor')
    ylim(ysc*[-1 1])
    xlabel('Frame'), ylabel('x-offset (pixel)')
       title('Welch FFT Window');
subplot(2,2,2)
plot(yData,'.b')
hold on
ywin=plot([i i i+window i+window i],-ysc*[-1 1 1 -1 -1],'r');
set(ywin,'EraseMode','xor')
ylim(ysc*[-1 1])
ylabel('y-offset (pixel)');
xlabel('Frame');
title('Welch FFT Window');
    subplot(2,2,3)
    hx=semilogy(xww,Pxx,'.-r');
    set(hx,'EraseMode','xor')
    axis([0 15 10e-7 10e4]), hold on
    warnbarx=semilogy([0.1 15],5e-6*[1 1],'Color','g','LineWidth',12);
    set(warnbarx,'EraseMode','xor')
    grid on;
    xlabel('Frequency (Hz)'), ylabel('x-offset PSD (dB/Hz)')
    title('Welch PSD Estimate');
subplot(2,2,4)
hy=semilogy(yww,Pyy,'.-r');
set(hy,'EraseMode','xor')
hold on
warnbary=semilogy([0.1 15],5e-6*[1 1],'Color','g','LineWidth',12);
set(warnbary,'EraseMode','xor')
axis([0 15 10e-7 10e4]), grid on;
xlabel('Frequency (Hz)'), ylabel('y-offset PSD (dB/Hz)')
title('Welch PSD Estimate');

%% Animating Welch PSD estimates
size=length(xData);
badIndex(1)=0;
while i<(size-window)
    set(xwin,'XData',[i i i+window i+window i],'YData',-ysc*[-1 1 1 -1 -1]);
    set(ywin,'XData',[i i i+window i+window i],'YData',-ysc*[-1 1 1 -1 -1]);
xOffset=xData(i:i+window);
yOffset=yData(i:i+window);
    [Pxx,xww]=pwelch(xOffset,[],[],window,30);
    [Pyy,yww]=pwelch(yOffset,[],[],window,30);
set(hx,'XData',xww,'YData',Pxx);
set(hy,'XData',yww,'YData',Pyy);
      set(warnbarx,'Color','g');      set(warnbary,'Color','g');
    if min(Pxx)<5*PSDthreshold || min(Pyy)<5*PSDthreshold
      set(warnbarx,'Color','y');      set(warnbary,'Color','y');
    end                          % if end
    if min(Pxx)<PSDthreshold || min(Pyy)<PSDthreshold
      set(warnbarx,'Color','r');      set(warnbary,'Color','r');
      badIndex=[badIndex; i];   % There is a failure in the target tracking
    end                          % if end
i=i+overlap; pause(0.001)
end
badIndex(1)=[];
return
```


Appendix B. Example of animating 2D scene using handles (‘Erase Mode’ method)

```
function x=Animation(speed, psi, theta, phi, alpha, beta,...
                     speedI,psiI,thetaI,phiI,alphaI,betaI)
% This script 'animates' the data received during ballistic missile intercept
% simulation. It requires six input vectors of parameters for both missile and
% interceptor:
%          speed -  defining missile's full speed time history,
%  psi,theta,phi -  defining Euler angles histories (orientation of {b}
%                   wrt {n}),
%     alpha,beta -  defining angle of attack and sideslip angle
%                   histories.
%       To see how it works you can try to run the following six lines:
%          N=100; time=0:N;
%          psi=pi/2*ones(N,1); theta=pi/2*cos(pi/N/3*time); phi=zeros(N,1);
%          alpha=0.3*sin(2*pi/N*time); beta=zeros(N,1);
%          speed=2000*sin(pi/N/2*time);
%          Animation(speed,psi,theta,phi,alpha,beta,...
%                    speed,pi+psi,theta,phi,alpha,beta);
%
NumbofFrs=length(speed)-1;
time=0:NumbofFrs;
%% Define Ballistic Missile's geometry (3 stages)
% The geometry is defined for a half of the missile
xBMs{1}=[0  0    2    3  16  19   30   32];
yBMs{1}=[0 1.85 1.85 1.1 1.1 0.65 0.65  0];
xBMs{2}=[0  0    1    2   14   16];
yBMs{2}=[0 1.31 1.31 0.65 0.65  0];
xBMs{3}=[0  0    2];
yBMs{3}=[0 0.65  0];
% Define geometry for the second half
for iMS=1:3
nP=length(xBMs{iMS});
for i=1:nP-1
xBMs{iMS}=[xBMs{iMS}  xBMs{iMS}(nP-i)];
yBMs{iMS}=[yBMs{iMS} -yBMs{iMS}(nP-i)];
end
% Place the missile to the center of the image and scale it so that it
% occupies 2/3 of the screen
sca=max(xBMs{iMS})-min(xBMs{iMS});
xbm=3*(xBMs{iMS}-sca/2)/sca;  % Missile geometry is defined in NED
ybm=3*yBMs{iMS}/sca;          % frame {b} (x-axis is pointe North)
% The final (full, centered and scaled) geomery
Missile{iMS}(:,1)=xbm'; Missile{iMS}(:,2)=ones(length(xbm),1);
Missile{iMS}(:,3)=ybm';
end
%% Define Interceptor's geometry (2 stages)
% The geometry is defined for a half of the missile
xIMs{1}=[0 0    1.6   1.7  1.74 1.83 2  2.1 2.6  2.8 4.5 4.7  6  6.63];
yIMs{1}=[0 0.265 0.265 0.17 0.17 0.29 0.33 0.17 0.17 0.3 0.3 0.17 ...
         0.17 0];
xIMs{2}=[0 0    0.04 0.13 0.3  0.4  0.9  1.1 2.8 3    4.3  4.93];
yIMs{2}=[0 0.17 0.17 0.29 0.33 0.17 0.17 0.3 0.3 0.17 0.17 0];
% Define geometry for the second half
for iIS=1:2
nP=length(xIMs{iIS});
for i=1:nP-1
xIMs{iIS}=[xIMs{iIS}  xIMs{iIS}(nP-i)];
yIMs{iIS}=[yIMs{iIS} -yIMs{iIS}(nP-i)];
end
% Place the missile to the center of the image and scale it so that it
% occupies 2/3 of the screeng
sca=max(xIMs{iIS})-min(xIMs{iIS});
xim=3*(xIMs{iIS}-sca/2)/sca;  % Missile geometry is defined in NED
yim=3*yIMs{iIS}/sca;          % frame {b} (x-axis is pointed North)
% The final (full, centered and scaled) geomery
Interceptor{iIS}(:,1)=xim'; Interceptor{iIS}(:,2)=ones(length(xim),1);
Interceptor{iIS}(:,3)=yim';
end
%% Define the initial frame for Missile
figure('Name','Side-View Animation')
subplot(1,2,1)
R_psi   = [ cos(psi(1))   sin(psi(1))   0;
           -sin(psi(1))   cos(psi(1))   0;
                0             0         1];
R_theta = [cos(theta(1))  0  -sin(theta(1))
              0           1       0;
```



```
           sin(theta(1))  0   cos(theta(1))];
R_phi   = [1       0            0;
           0   cos(phi(1))   sin(phi(1));
           0  -sin(phi(1))   cos(phi(1))];
Rm_n2b = R_phi*R_theta*R_psi;        % Rotation from {n} to {b}
imMis=Rm_n2b'*Missile{1}';           % Missile's coordinates in {n}
Mis=fill(imMis(2,:),-imMis(3,:),'c');% Projection onto y-z plane of {n}
set(Mis,'EraseMode','xor')
axis(2*[-1 1 -1 1]), axis equal, hold on
plot(0,0,'r.')                                % Center of the figure
plot([0 0],[0 -.1],'Color','k','LineWidth',2)   % Gravity vector
R_beta   = [ cos(beta(1))   sin(beta(1))   0;
            -sin(beta(1))   cos(beta(1))   0;
                 0              0          1];
R_alpha = [ cos(-alpha(1))  0  -sin(-alpha(1))
                 0          1         0;
            sin(-alpha(1))  0   cos(-alpha(1))];
Rm_n2v = R_alpha*Rm_n2b;              % Rotation from {n} wrt {v}
Speed=[speed(1); 0; 0]/1000;         % Speed magnitude in {v}
imSpd=Rm_n2v'*Speed;                 % Projection onto y-z plane of {n}
SpM=plot([0 imSpd(2)],[0 -imSpd(3)],'Color','r','LineWidth',2);
set(SpM,'EraseMode','xor');
textSpeed=text(imSpd(2)+.1,-imSpd(3),'Speed');
set(textSpeed,'EraseMode','xor');
xlabel('East (y_{LTP})'), ylabel('Up (-z_{LTP})')
set(gca,'XTickLabel',{}), set(gca,'YTickLabel',{})
title('Ballistic Missile Attitude')
%% Display initial frame and time
textFrame=text('Color',[0.8471 0.1608 0],'FontAngle','italic',...
       'Position',[0.9 2.75],'String',['Frame ' num2str(1) ' out of '...
       num2str(NumbofFrs)],'BackgroundColor',[1 1 1]);
textTime=text('Color',[0.8471 0.1608 0],'FontAngle','italic',...
       'Position',[0.9 2.35],'String',['Time ' num2str(time(1),'%.2f')...
       ' sec'],'BackgroundColor',[1 1 1]);
%% Define the initial frame for Interceptor
subplot(1,2,2)
R_psi   = [ cos(psiI(1))   sin(psiI(1))   0;
           -sin(psiI(1))   cos(psiI(1))   0;
                0             0         1];
R_theta = [cos(thetaI(1))  0  -sin(thetaI(1))
               0          1       0;
           sin(thetaI(1))  0   cos(thetaI(1))];
R_phi   = [1       0            0;
           0   cos(phiI(1))   sin(phiI(1));
           0  -sin(phiI(1))   cos(phiI(1))];
Ri_n2b = R_phi*R_theta*R_psi;        % Rotation from {n} to {b}
imInt=Ri_n2b'*Interceptor{1}';       % Interceptor's coordinates in {n}
Int=fill(imInt(2,:),-imInt(3,:),'r');% Projection onto y-z plane of {n}
set(Int,'EraseMode','xor');
axis(2*[-1 1 -1 1]), axis equal, hold on
plot(0,0,'r.')                                % Center of the figure
plot([0 0],[0 -.1],'Color','k','LineWidth',2)   % Gravity vector
R_beta   = [ cos(betaI(1))   sin(betaI(1))   0;
            -sin(betaI(1))   cos(betaI(1))   0;
                 0              0          1];
R_alpha = [ cos(-alphaI(1))  0  -sin(-alphaI(1))
                 0          1         0;
            sin(-alphaI(1))  0   cos(-alphaI(1))];
Ri_n2v = R_alpha*Ri_n2b;             % Rotation from {n} wrt {v}
Speed=[speedI(1); 0; 0]/1000;        % Speed magnitude in {v}
imSpd=Ri_n2v'*Speed;                 % Projection onto y-z plane of {n}
SpI=plot([0 imSpd(2)],[0 -imSpd(3)],'Color','r','LineWidth',2);
set(SpI,'EraseMode','xor');
textSpeedI=text(imSpd(2)+.1,-imSpd(3),'Speed');
set(textSpeedI,'EraseMode','xor');
xlabel('East (y_{LTP})'), ylabel('Up (-z_{LTP})')
set(gca,'XTickLabel',{}), set(gca,'YTickLabel',{})
title('Interceptor Attitude')
%% Add the 'Next Frame' and 'Auto' buttons
uicontrol('string','Next Frame','units','normalized','pos',[.66,.15,.13,.06],...
          'callback','set(gcf,''userdata'',1)');
auto = uicontrol('string','Auto','units','normalized','pos',[.8 .15,.08,.06],...
                 'style','togglebutton','callback','set(gcf,''userdata'',1)');
set(gcf,'userdata',0); goFlag=0;
%% Start animation
for j = 2:NumbofFrs
%% i) Define current stage for Missile and Interceptor
```



```
if time(j)<6                            % Interceptor booster burns out
    iMS=1; iIS=1;
elseif time(j)<130                      % Missile 1st stage burns out
    iMS=1; iIS=2;
elseif time(j)<240                      % Missile 2nd stage burns out
    iMS=2; iIS=2;
else
    iMS=3; iIS=2;
end
%% ii) Define rotation matrix from {n} to {b} and rotate the missile
R_psi   = [ cos(psi(j))   sin(psi(j))   0;
           -sin(psi(j))   cos(psi(j))   0;
                0             0         1];
R_theta = [cos(theta(j))  0  -sin(theta(j))
                0         1       0;
           sin(theta(j))  0   cos(theta(j))];
R_phi   = [1       0            0;
           0   cos(phi(j))   sin(phi(j));
           0  -sin(phi(j))   cos(phi(j))];
Rm_n2b = R_phi*R_theta*R_psi;           % Rotation from {n} to {b}
imMis=Rm_n2b'*Missile{iMS}';            % Missile's coordinates in {n}
set(Mis,'XData',imMis(2,:),'YData',-imMis(3,:)); % y-z plane projection
%% iii) Define rotation matrix for the Missile's speed vector
%% and rotate it
R_beta   = [ cos(beta(j))   sin(beta(j))   0;
            -sin(beta(j))   cos(beta(j))   0;
                0             0          1];
R_alpha = [ cos(-alpha(j))  0  -sin(-alpha(j))
                0           1         0;
            sin(-alpha(j))  0   cos(-alpha(j))];
Rm_n2v = R_alpha*Rm_n2b;                % Rotation from {n} wrt {v}
Speed=[speed(j); 0; 0]/1000;            % Speed magnitude in {v}
imSpd=Rm_n2v'*Speed;                    % Projection onto y-z plane of {n}
set(SpM,'XData',[0 imSpd(2)],'YData',[0 -imSpd(3)]);
set(textSpeed,'Position',[imSpd(2)+.1 -imSpd(3) 0]);
%% iv) Define rotation matrix from {n} to {b} and rotate the interceptor
R_psi   = [ cos(psiI(j))   sin(psiI(j))   0;
           -sin(psiI(j))   cos(psiI(j))   0;
                0             0         1];
R_theta = [cos(thetaI(j))  0  -sin(thetaI(j))
                0         1       0;
           sin(thetaI(j))  0   cos(thetaI(j))];
R_phi   = [1       0            0;
           0   cos(phiI(j))   sin(phiI(j));
           0  -sin(phiI(j))   cos(phiI(j))];
Ri_n2b = R_phi*R_theta*R_psi;           % Rotation from {n} to {b}
imInt=Ri_n2b'*Interceptor{iIS}';        % Interceptor coordinates in {n}
set(Int,'XData',imInt(2,:),'YData',-imInt(3,:)); % y-z plane projection
%% v) Define rotation matrix for the Interceptor's speed vector
%% and rotate it
R_beta   = [ cos(betaI(j))   sin(betaI(j))   0;
            -sin(betaI(j))   cos(betaI(j))   0;
                0             0          1];
R_alpha = [ cos(-alphaI(j))  0  -sin(-alphaI(j))
                0           1         0;
            sin(-alphaI(j))  0   cos(-alphaI(j))];
Ri_n2v = R_alpha*Ri_n2b;                % Rotation from {n} wrt {v}
Speed=[speedI(j); 0; 0]/1000;           % Speed magnitude in {v}
imSpd=Ri_n2v'*Speed;                    % Projection onto y-z plane of {n}
set(SpI,'XData',[0 imSpd(2)],'YData',[0 -imSpd(3)]);
set(textSpeedI,'Position',[imSpd(2)+.1 -imSpd(3) 0]);
%% vi) Count frames
set(textFrame,'String',['Frame ' num2str(j) ' out of ' ...
    num2str(NumbofFrs)]);
set(textTime,'String',['Time ' num2str(time(j),'%.2f') ' sec']);
%% vii) Wait for any control button to be pushed
while goFlag==0
     if get(auto,'value')==1
         goFlag=1;
     elseif get(gcf,'userdata')==1
         goFlag=1; set(gcf,'userdata',0)
     else
         pause(0.25)
     end
 end
 goFlag=0; pause(0.1)
end
```


Appendix C. Example of animating 2D and 3D scenes using 'Creating Movie' approach

The **anim_floor** function below is the main function. It sequentially calls **draw_floor**, **draw_robot**, and **draw_dev** functions (appearing after **anim_floor** below) to produce the bird-eye view of the rendezvous geometry and **draw_3D** function to visualize what would robot's camera see. The input signals are:

- vector `time(:)` holding the time stamps of the simulation;
- three-dimensional array `states(1:3,1:3,:)`, composed of time histories of robots' *x-y* position and yaw angle;
- two-dimensional array `thrusters(:,1:3)`, describing time histories of thrusters' orientation; and
- two-dimensional array `camera(:,1:6)`, containing time histories of cameras and lidars orientation.

The geometry of the floor and three robots used in almost every function is defined not as a set of global variables but rather in the **global_props** function shown the last.

```
function anim_floor(time,states,thrusters,camera)
% This function animates the bird-eye and 3D view of the rendezvous animation
% For the bird-eye view it sequentially calls another functions:
%                   draw_floor, draw_robot, and draw_dev
% For the 3D view it calls draw_foto functions
%
[robot_props, floor_props] = global_props;
mov = avifile('robotmov.avi','quality',100,'Compression','Indeo3','fps',5);
 [m,n] = size(time);
for i = 1:ceil(m/100):m
    subplot(1,2,1);
    draw_floor(time(i));
    for j = 1 : 3
        pos=states(1:3,j,i);
        draw_robot(pos,robot_props(j));
        switch j
            case 1
% draw camera field of view
            draw_dev(j, pos, 'Cam', camera(i,1));
% draw 360 vectored variable thruster
            draw_dev(j, pos, 'Thruster', thrusters(i,1:3));
% draw lidar beam
            draw_dev(j, pos, 'Lidar', camera(i,[1:2,4:5]));
            case {2,3}
        end
    end
% draw camera's snap-shot
    subplot(1,2,2);
    data=states(1:3,1:3,i)';
    alf= camera(i,1)+states(3,1,i);
    draw_3D(1, data, alf,0,0)
mov = addframe(mov,getframe(gcf));
end
mov = close(mov);

function draw_floor(t)
% This function plots the rectangular floor
[robot_props,floor_props]=global_props;
hold off
BlueFloor=floor_props.dim;
fill([BlueFloor(2,3) BlueFloor(3,2)], [BlueFloor(1,1) BlueFloor(1,2)],'w'), hold on
axis equal, axis([BlueFloor(2,2) BlueFloor(3,2) BlueFloor(1,1) BlueFloor(2,1)]);
title('Bird''s Eye View');
xlabel('y-axis (East) (m)'), ylabel('x-axis (North) (m)')
text('Color',[0.8471 0.1608 0],'FontAngle','italic',...
    'Position',[.1 .1],...
    'String',['time=' num2str(round(100*t)/100)])

function draw_robot(pos,robot)
% This function plots the robot (top view)
x=pos(1); y=pos(2); t=pos(3);   % postion/orientation of robot
% Converting robot's corners from {b} to {n}
r2n  = [cos(t) -sin(t)  0;
        sin(t)  cos(t)  0;
            0        0  1];
RobCrns=r2n*robot.crns'+[x;y;0]*ones(1,8);
fill(RobCrns(2,1:4,1),RobCrns(1,1:4,1),robot.dc)
radius=abs(robot.crns(1));
line([y y+radius*sin(t)],[x x+radius*cos(t)], 'Color', 'y')
```



```
function draw_dev(me,pos,type,u)
% This function plots diverging rays corresponding to thrusters and lidar
robot_props=global_props;
x=pos(1); y=pos(2); t=pos(3);
switch type
case 'Cam'
 a=u+t;
 clr= robot_props(me).lc;
 sFoV= robot_props(me).sfov;
 rx=40/x; ry=40/y;
 patch(y*[1 1+ry*sin(a-sFoV) 1+ry*sin(a+sFoV) 1],...
 x*[1 1+rx*cos(a-sFoV) 1+rx*cos(a+sFoV) 1],...
 clr, 'LineStyle', 'none', 'FaceAlpha', .25);
case 'Thruster'
 mag=u(1)*10;
 a=u(2)+t+pi;
 clr='c';
 sFoV=.05;
 rx=mag/x; ry=mag/y;
 patch(y*[1 1+ry*sin(a-sFoV) 1+ry*sin(a+sFoV) 1],...
 x*[1 1+rx*cos(a-sFoV) 1+rx*cos(a+sFoV) 1],...
 clr, 'LineStyle', 'none', 'FaceAlpha', .25);
case 'fThruster'
 mag=u(1)*10;
 d=- robot_props(me).crns(1);
 a=u(2)+t;
 clr='c';
 len=.3;
 sFoV=.05;
 rx=mag; ry=mag;
 patch([y+d*sin(t) y+ry*sin(a-sFoV)+d*sin(t) y+ry*sin(a+sFoV)+d*sin(t) y+d*sin(t)],...
 [x+d*cos(t) x+rx*cos(a-sFoV)+d*cos(t) x+rx*cos(a+sFoV)+d*cos(t) x+d*cos(t)],...
 clr, 'LineStyle', 'none', 'FaceAlpha', .25);
 plot([y+d*sin(t), y+d*sin(t)+len*sin(a)], [x+d*cos(t), x+d*cos(t)+len*cos(a)], 'm');
case 'bThruster'
 mag=u(1)*10;
 d= robot_props(me).crns(1);
 a=u(2)+t+pi;
 clr='c';
 len=.3;
 sFoV=.05;
 rx=mag; ry=mag;
 patch([y+d*sin(t) y+ry*sin(a-sFoV)+d*sin(t) y+ry*sin(a+sFoV)+d*sin(t) y+d*sin(t) ],...
 [x+d*cos(t) x+rx*cos(a-sFoV)+d*cos(t) x+rx*cos(a+sFoV)+d*cos(t) x+d*cos(t)],...
 clr, 'LineStyle', 'none', 'FaceAlpha', .25);
 plot([y+d*sin(t), y+d*sin(t)+len*sin(a)], [x+d*cos(t), x+d*cos(t)+len*cos(a)], 'm');
case 'Lidar'
 r12 = u(1); b12 = u(2); r13 = u(3); b13 = u(4);
 x2 = x+r12*cos(t+b12); y2 = y+r12*sin(t+b12);
 x3 = x+r13*cos(t+b13); y3 = y+r13*sin(t+b13);
 plot(y2+.1*randn(1,10),x2+.1*randn(1,10),'.')
 plot(y3+.1*randn(1,10),x3+.1*randn(1,10),'.')
end
```

```
function draw_3D(ME,FLR,psi,theta,phi)
% This function produces the 3D view from one of the robots
[robot_props, floor_props]=global_props;
% psi=yaw, theta=pitch, phi=roll
X=FLR(ME,1); Y=FLR(ME,2); T=FLR(ME,3);
%% Defining parameters of the square in {n} (NED)
BlueFloor=floor_props.dim;                % Square's corners starting from the origin
NumbofPts=length(BlueFloor);
%% Defining camera (attached to the robot's top)
hc= robot_props(ME).crns(end);        % Camera's height above the ground
Camera = [X; Y; hc];                  % Camera's position in {n}
sFoV= robot_props(ME).sfov;           % Semi-field of view (horizontal)
AsRatio= robot_props(ME).ar;          % Frame's aspect ratio' (horizontal/vertical)
f= robot_props(ME).f;                 % Focal length (m)
R_phi   = [1   0          0;
           0   cos(phi)   sin(phi);
           0  -sin(phi)   cos(phi)];
R_theta = [cos(theta)  0  -sin(theta)
           0           1   0;
           sin(theta)  0   cos(theta)];
%% Defining two other robots in {b} (NED) attached to the robot's bottom
if ME==1, ROBOT1=2; ROBOT2=3; end
```



```
if ME==2, ROBOT1=1; ROBOT2=3; end
if ME==3, ROBOT1=1; ROBOT2=2; end
%% Defining light green and dark green colors
RobPos=[FLR(ROBOT1,1),FLR(ROBOT1,2),0;     % Origin of the robot's {b} in {n}
        FLR(ROBOT2,1),FLR(ROBOT2,2),0];
RobOr= [FLR(ROBOT1,3);FLR(ROBOT2,3)];      % Orientation of {b} wrt to {n}
L(1,:)=robot_props(ROBOT1).lc;
D(1,:)=robot_props(ROBOT1).dc;
L(2,:)=robot_props(ROBOT2).lc;
D(2,:)=robot_props(ROBOT2).dc;
CRNS{1}=robot_props(ROBOT1).crns;
CRNS{2}=robot_props(ROBOT2).crns;
NumbofRbts=length(RobOr);
for u=1:NumbofRbts
R_r2n(:,:,u)  = [cos(RobOr(u))  -sin(RobOr(u))  0;
                 sin(RobOr(u))   cos(RobOr(u))  0;
                      0              0         1];
end
%% Defining a camera frame
Uscale=f*tan(sFoV);
Vscale=Uscale/AsRatio;
%% i) Converting the square to the camera frame
R_psi = [cos(psi) sin(psi) 0;
        -sin(psi) cos(psi) 0;
                0        0 1];
R_n2c = R_phi*R_theta*R_psi;                     % Rotation from {n} wrt {c}
imrs=R_n2c*(BlueFloor'-Camera*ones(1,NumbofPts));% Coordinates in {c}
azimuth=atan2(imrs(2,:),imrs(1,:));
u0 = f*imrs(2,:)./imrs(1,:);                     % x-coordinate in {f} (right)
v0 =-f*imrs(3,:)./imrs(1,:);                     % y-coordinate in {f} (right)
%% ii) Counting the number and indices of Visible and Invisible points
indVis=find(imrs(1,:)>0); indInv=find(imrs(1,:)<=0);
nVis=length(indVis); nInv=NumbofPts-nVis;
%% iii) Reodering the points
if (nVis~=1) & (min(indInv)>1 & max(indInv)<NumbofPts)
    fict=indVis;
    indVis=(max(indInv)+1):NumbofPts;
    indVis=[indVis 1:(min(indInv)-1)];
end
%% iv) Assigning fictituous points as substitutes for invisible points
u(2:nVis+1)=u0(indVis);
v(2:nVis+1)=v0(indVis);
inleft=indVis(1)-1;     if inleft<1,          inleft=inleft+NumbofPts;   end
inright=indVis(nVis)+1; if inright>NumbofPts, inright=inright-NumbofPts; end
tau1=abs((-sFoV-azimuth(indVis(1)))/(azimuth(inleft)-azimuth(indVis(1))));
tau2=abs((sFoV-azimuth(indVis(nVis)))/(azimuth(inright)-azimuth(indVis(nVis))));
imrLeft=imrs(:,inleft)*tau1+imrs(:,indVis(1))*(1-tau1);
imrRight=imrs(:,inright)*tau2+imrs(:,indVis(nVis))*(1-tau2);
ul = f*imrLeft(2)/imrLeft(1);       % Coordinates of fictituous points in {f}
vl =-f*imrLeft(3)/imrLeft(1);
ur = f*imrRight(2)/imrRight(1);
vr =-f*imrRight(3)/imrRight(1);
u(1)=(-Vscale-vl)*(u(2)-ul)/(v(2)-vl)+ul; v(1)=-Vscale;
u(nVis+2)=(-Vscale-vr)*(u(nVis+1)-ur)/(v(nVis+1)-vr)+ur; v(nVis+2)=-Vscale;
%% v) Converting robots centers from {n} to {c}
imRts=R_n2c*(RobPos'-Camera*ones(1,NumbofRbts)); % Robots coordinates in {c}
distRts=[norm(imRts(:,1),2) norm(imRts(:,2),2)]; % Distance from the origin of {c}
azimuthRts=atan2(imRts(2,:),imRts(1,:));         % Robots azimuths in {c}
for jr=1:NumbofRbts
    % vi) Converting robot's corners from {b} to {n}
    Robot=CRNS{jr};
    cyl=sqrt(Robot(1)^2+Robot(2)^2);          % Cylinder around the robot
    RobCrns(:,:,jr)=R_r2n(:,:,jr)*Robot'+RobPos(jr,:)'*ones(1,8);
    % vii) Default zeroing of left and right planes' coordinates (in {f})
    uR(:,2*jr-1) = zeros(4,1);  vR(:,2*jr-1) = zeros(4,1);
    uR(:,2*jr)   = zeros(4,1);  vR(:,2*jr)   = zeros(4,1);
    uRcolor(:,:,jr)=[L(jr,:);D(jr,:)];
    if abs(azimuthRts(jr))-sFoV<pi/2 && distRts(jr)*sin(abs(azimuthRts(jr))-sFoV)<cyl
        %% viii) Converting visible robot's corners from {n} to {c}
        imRtsCrns(:,:,jr)=R_n2c*(RobCrns(:,:,jr)-Camera*ones(1,8));   % Coordinates in {c}
        %% ix) Determining the closest edge and two adjacent panels (left and right)
        [dv,in]=min([norm(imRtsCrns(1:3,1,jr)),norm(imRtsCrns(1:3,2,jr)),...
            norm(imRtsCrns(1:3,3,jr)),norm(imRtsCrns(1:3,4,jr))]);
        inL=in+1;       if inL>4,   inL=inL-4;    end
        inR=in-1;       if inR<1,   inR=inR+4;    end
        Panel(:,:,2*jr-1)=[imRtsCrns(:,in,jr),imRtsCrns(:,inL,jr),...   % Left panel
                           imRtsCrns(:,inL+4,jr),imRtsCrns(:,in+4,jr)];
```



```
        Panel(:,:,2*jr)  =[imRtsCrns(:,in,jr),imRtsCrns(:,inR,jr),...   % Right panel
                          imRtsCrns(:,inR+4,jr),imRtsCrns(:,in+4,jr)];
        %% x) Determining more distant panel (left or right) to be shown first
        dl=norm(mean(Panel(:,:,2*jr-1),2));
        dr=norm(mean(Panel(:,:,2*jr),2));
        tt=[0,1];
        if dl<dr, tt=[1,0]; uRcolor(:,:,jr)=[D(jr,:);L(jr,:)]; end
        %% xi) Computing {f}-coordinates of the farther and closer panels
        for jt=1:2
            uR(:,2*jr-1+tt(jt))= f*Panel(2,:,2*jr-2+jt)./Panel(1,:,2*jr-2+jt);
            vR(:,2*jr-1+tt(jt))=-f*Panel(3,:,2*jr-2+jt)./Panel(1,:,2*jr-2+jt);
        end
    end             % The end of the 'if' structure
end                 % The end of the 'for' loop
ord=[2,1]; if distRts(2)<distRts(1), ord=[1,2]; end
%     u(5)=u(1); v(5)=v(1);
%     uR(5,:)=uR(1,:); vR(5,:)=vR(1,:);
fill([-1 1 1 -1], [-1 -1 1 1], 'w','FaceAlpha', 1)
patch(u,v,'c','FaceAlpha', 1);
patch(uR(:,2*ord(1)-1),vR(:,2*ord(1)-1),uRcolor(1,:,ord(1)), 'FaceAlpha', 1);
patch(uR(:,2*ord(1)),  vR(:,2*ord(1)),  uRcolor(2,:,ord(1)), 'FaceAlpha', 1);
patch(uR(:,2*ord(2)-1),vR(:,2*ord(2)-1),uRcolor(1,:,ord(2)), 'FaceAlpha', 1);
patch(uR(:,2*ord(2)),  vR(:,2*ord(2)),  uRcolor(2,:,ord(2)), 'FaceAlpha', 1);
axis equal, axis([-Uscale Uscale -Vscale Vscale]);
title (['SimImage from ' robot_props(ME).name])

function [robot,floor]=global_props();
%% Defining the first robot (as a structure)
d2r=pi/180;
robot(1).name='Blue';
robot(1).sfov=23*d2r;
robot(1).f=.1;
robot(1).ar=4/3;
robot(1).lc=[0.6 .6 1];
robot(1).dc=[0 0 1];       % Define light green and dark colors
a=.288/2; b=.288/2; h=.6;
robot(1).crns=[...
   -a, -b, 0;              % Robot's corners starting from the left-bottom-floor
    a, -b, 0;              % and going clockwise (1-2-3-4);
    a,  b, 0;
   -a,  b, 0;
   -a, -b, -h;             % The same pattern is repeated at the above-floor
    a, -b, -h;             % level (5-6-7-8), i.e. 5 is located above 1, etc.
    a,  b, -h;
   -a,  b, -h];
clear a b h d2r
%% Defining two other robots
robot(2)=robot(1); robot(2).name='Red';
robot(2).lc=[1 .6 .6]; robot(2).dc=[1 0 0];
robot(3)=robot(1); robot(3).name='Green';
robot(3).lc=[0.6 1 .6]; robot(3).dc=[0 1 0];
%% Defining the floor (as a structure)
floor.dim=[  0,   0, 0;    % Square's corners starting from the origin
           4.9,   0, 0;    % (left bottom) and going clockwise
           4.9, 4.3, 0;
             0, 4.3, 0];
```

Appendix D. Example of animating 3D scene using handles (‘Erase Mode’ method)

```
%% Defining payload and its 'trajectory'
xb = [ 1  1 -1 -1  1  1 -1 -1];
yb = [ 1 -1 -1  1  1 -1 -1  1];
zb = [-1 -1 -1 -1  1  1  1  1];
Npoints = 600;
t     = linspace(0,Npoints,Npoints);
phi   = 0.4*sin(t*4*pi/Npoints);
theta = 0.4*cos(t*4*pi/Npoints);
psi   = t*1*pi/Npoints;
el = linspace(-20,0,Npoints);
si = linspace(6,2,Npoints);
%% Setting the initial view
fig = figure(1)
set(fig,'DoubleBuffer','on');
axis equal
set(gca,'xlim',si(1)*[-1 1],'ylim',si(1)*[-1 1],'zlim',si(1)*[-1 1],...
                                        'View',[-14,el(1)],'Visible','off')
R_phi   = [      1            0            0;
                 0        cos(phi(1))   sin(phi(1));
                 0       -sin(phi(1))   cos(phi(1))];
R_theta = [cos(theta(1))      0        -sin(theta(1))
                 0            1            0;
            sin(theta(1))     0         cos(theta(1))];
R_psi   = [ cos(psi(1))   sin(psi(1))       0;
           -sin(psi(1))   cos(psi(1))       0;
                 0            0            1];
R_n2c = R_phi*R_theta*R_psi;                    % Rotation from {n} to {c}
for k=1:8
zz=R_n2c*[xb(k); yb(k); zb(k)];
x(k)=zz(1); y(k)=zz(2); z(k)=zz(3);
end
MS=24/si(1);
h_top   = patch(x([1:4]),y([1:4]),z([1:4]),'b',...
                         'Marker','o','MarkerSize',MS,'MarkerFaceColor','r');
hold
h_bottom= patch(x([5:8]),y([5:8]),z([5:8]),'b',...
                         'Marker','o','MarkerSize',MS,'MarkerFaceColor','r');
h_front = patch(x([1,2,6,5]),y([1,2,6,5]),z([1,2,6,5]),'b',...
                         'Marker','o','MarkerSize',MS,'MarkerFaceColor','r');
h_back  = patch(x([2,3,7,6]),y([2,3,7,6]),z([2,3,7,6]),'b',...
                         'Marker','o','MarkerSize',MS,'MarkerFaceColor','r');
h_left  = patch(x([3,4,8,7]),y([3,4,8,7]),z([3,4,8,7]),'b',...
                         'Marker','o','MarkerSize',MS,'MarkerFaceColor','r');
h_right = patch(x([4,1,5,8]),y([4,1,5,8]),z([4,1,5,8]),'b',...
                         'Marker','o','MarkerSize',MS,'MarkerFaceColor','r');
%% Starting animation
for j=2:Npoints
pause(0.005);
R_phi   = [      1            0            0;
                 0        cos(phi(j))   sin(phi(j));
                 0       -sin(phi(j))   cos(phi(j))];
R_theta = [cos(theta(j))      0        -sin(theta(j))
                 0            1            0;
            sin(theta(j))     0         cos(theta(j))];
R_psi   = [ cos(psi(j))   sin(psi(j))       0;
           -sin(psi(j))   cos(psi(j))       0;
                 0            0            1];
R_n2c = R_phi*R_theta*R_psi;
for k=1:8
zz=R_n2c*[xb(k); yb(k); zb(k)];
x(k)=zz(1); y(k)=zz(2); z(k)=zz(3);
end
MS=24/si(j);
set(h_top,   'XData',x([1:4]),    'YData',y([1:4]),    'ZData',z([1:4]),'MarkerSize',MS);
set(h_bottom,'XData',x([5:8]),    'YData',y([5:8]),    'ZData',z([5:8]),'MarkerSize',MS);
set(h_front, 'XData',x([1,2,6,5]),'YData',y([1,2,6,5]),'ZData',z([1,2,6,5]),'MarkerSize',MS);
set(h_back,  'XData',x([2,3,7,6]),'YData',y([2,3,7,6]),'ZData',z([2,3,7,6]),'MarkerSize',MS);
set(h_left,  'XData',x([3,4,8,7]),'YData',y([3,4,8,7]),'ZData',z([3,4,8,7]),'MarkerSize',MS);
set(h_right, 'XData',x([4,1,5,8]),'YData',y([4,1,5,8]),'ZData',z([4,1,5,8]),'MarkerSize',MS);
set(gca,'xlim',si(j)*[-1 1],'ylim',si(j)*[-1 1],'zlim',si(j)*[-1 1],'View',[-14,el(j)])
end
```

