Tutorials for Course on Modeling and System Identification (MSI)
Albert-Ludwigs-Universität Freiburg – Winter Term 2016-2017

Emergency Guide to Matlab:
Concerning the Hunting Behavior of Lizards

Prof. Dr. Moritz Diehl, Robin Verschueren, Tobias Schoels, Rachel Leuthold and Mara Vaihinger

Matlab is a programming language specialized for matrix operations and other numerical computations. We're going to use Matlab to consider the case of a hunting lizard and a delicious beetle, as in Figure 1.

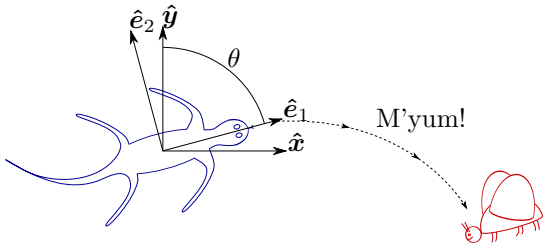

Figure 1: The lizard, the beetle, and two coordinate frames.

The Matlab Environment

Let's begin by opening Matlab, so that we see a screen like that shown in Figure 2. In addition to the toolbar above, there are four main windows: the Current Folder window, the Workspace window, the Editor window, and the Command Window.

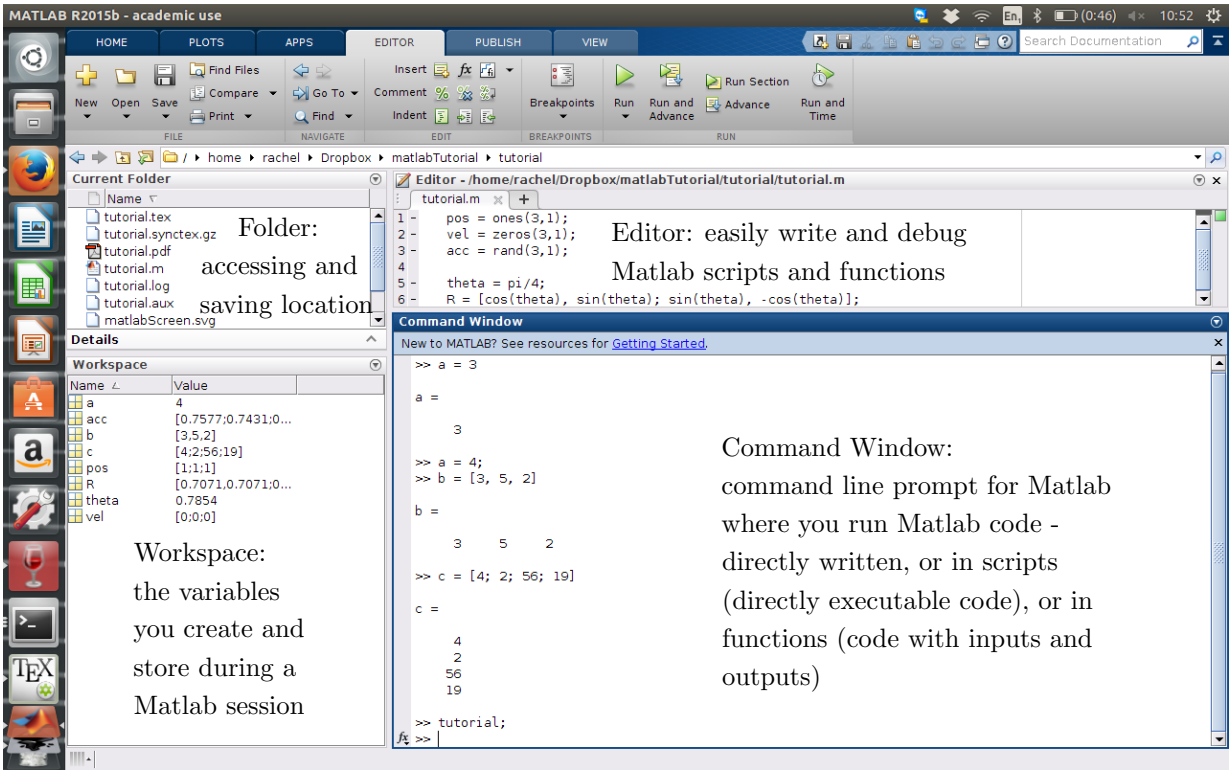

Figure 2: A Matlab screenshot to show the four windows.

There are three ways to code in Matlab. The first option is to write the code sequentially in the command window after the command prompt >>. This is not handy for larger programs. The other two ways are scripts and functions.

The difference between scripts and functions lies in their interaction with the workspace. Scripts can access the workspace; functions cannot. Any variable that is defined in a script will be saved in the workspace. Further, the

script can reference values that are recorded in the workspace. In a function, a defined variable is not automatically saved into the workspace. Further, the function cannot use variables directly from the workspace. Instead, functions have to be defined for given inputs and outputs.

That means that functions are always writen as `function_name` : `input_args` → `output_args`:

```
function [ output_args ] = function_name( input_args )
%FUNCTION_NAME Summary of this function goes here
% Detailed explanation goes here

% Code to get output_args goes here
end
```

Further, the fact that scripts will access the workspace creates the possibility that variables might be accidentally over-written or mis-initialized for long codes. To prevent that, you can use one or more of the following resetting functions:

`clear all`	clears everything; *this is typically unnecessary, use instead:*
`clear variables`	clears all of the variables in the workspace
`close all`	closes all open sub-windows, like figures
`clc`	visually clears the command window

So, we can start thinking about our lizard and beetle code by chosing 'New' (upper left) → 'Script'. Then, saving this untitled script as *tutorial.m*, where the first line is:

```
clear variables;
close all;
clc;
```

In Matlab, the semi-colon is the operator that supresses output at the end of a line. If there is no semi-colon at the end of a line, whatever output that line generates will be printed. This means that multiple lines can be connected just by connecting them with a semi-colon. (If multiple lines are written on one line, but the output-display is still desired, the line-connection can be done with a comma.) That is, this opening statement - which does not have an associated output -could be re-written equivalently as:

```
clear variables; close all; clc;
```

or

```
clear variables
close all
clc
```

However, when variables are defined, this output suppression can be useful. Consider when two comparable calculator phrases are enterred into the command window:

```
>> k = 3 + 2
k =
     5
>> l = 4 / 3;
```

Notice that we can request the output to be printed in a 'long' format that displays 15 digits after the decimal, or a 'short' format that displays 4 digits after the decimal. The format used does not change the value that is actually stored - only printed:

```
>> format long; l
l =
   1.333333333333333
>> format short; l
l =
    1.3333
```

Especially when coming from other scientific programming languages, notice that values will always be treated as floats, even if no decimal point is given during definition.

Variable Definition

Let's return to *tutorial.m*, and define some coordinate system information, in variables of various matrix size.

```
noDimensions = 2
xhat = [1; 0]
yhat = [0; 1];
I = [1, 0; 0, 1]
```

At this point, let's save *tutorial.m*, and run it. The script can be run either by pressing the green-arrow 'Run' button on the toolbar, or by calling `tutorial` in the command line. Notice that Matlab is an interpreted language, which means that we do not need to compile the code separately; instead the Matlab IDE will compile the code for us.

```
>> tutorial
```

This is only going to work if the working directory shown in the Folder window is the same as the directory that holds the file *tutorial.m*.

Then we get the output:

```
noDimensions =
     2
xhat =
     1
     0
I =
     1     0
     0     1
```

So, while defining a matrix with given inputs, commas separate the values within a row, and semi-colons separate the rows. Matlab stores matrices in memory by column, so it is faster to define one dimensional matrices (and access other matrices) as column-vectors rather than row-vectors.

Let's see the dimensions of these variables with the `size` function:

```
>> size(noDimensions)
ans =
     1     1
>> size(xhat)
ans =
     2     1
>> size(I)
ans =
     2     2
```

So, we can see that the dimensions of these matrices - respectively 1×1, 2×1 and 2×2 matrices, correspond to the standard linear algebra 'column $\times$ row' definition.

If we know that a matrix is a vector, we can use the `length` function to find its one dimensional size:

```
>> length(xhat)
ans =
     2
```

Just for background information, an equivalent way to define the identity matrix is:

```
I = eye(noDimensions);
```

Notice, also that indexing begins with 1, and indices are given in paranthesis, so that:

```
xhat(1)
```

will give the output:

```
ans =
     1
```

Operators for Matrices and Scalars

Arithmetic between scalars and matrices behaves as we know it from math class. We can see this by defining a beetle location with a linear combination of the $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$ vectors within our earth-fixed coordinate frame:

```
beetlePosition = 3*xhat + yhat/0.5
```

which will add the following line to the output when *tutorial.m* is run:

```
beetlePosition =
     3
     2
```

However, arithmetic operators between matrices and other matrices are always matrix-arithmetic operators and not scalar-arithmetic (or element-wise) operators. If we want the operator to be an element-wise operator, we have to preface the operator with a dot. Consequently, the following lines:

```
beetlePosition(1)
sum(beetlePosition .* xhat)
transpose(beetlePosition) * xhat
dot(beetlePosition, xhat)
```

all give the equivalent output for the x-component of the beetle position:

```
ans =
     3
```

whereas - due to mismatched matrix dimensions - the following line:

```
beetlePosition * xhat
```

gives an error:

```
Error using  *
Inner matrix dimensions must agree.


Error in tutorial (line 16)
beetlePosition * xhat
```

Matlab has an additional 'elementary-operator' defined for the complex-conjugate transpose, the apostrophe operator. When a matrix has only real values, this apostrophe operator is equivalent to the verbal `transpose` function. The element-wise version of the apostrophe operator - the dot apostrophe operator - is always equivalent to the `transpose` function. Then, these lines:

```
transpose(beetlePosition)
beetlePosition'
beetlePosition.'
```

give the same output:

```
ans =
     3     2
```

Whereas, if we define a complex vector, we can see that the apostrophe operator and the dot apostrophe operator are no longer equivalent:

```
complexVector = [beetlePosition(1) + i * beetlePosition(2); 4 + 6i]
transposeCV = transpose(complexVector)
dotApostropheCV = complexVector.'
apostropheCV = complexVector'
```

will have the output:

```
complexVector =
   3.0000 + 2.0000i
   4.0000 + 6.0000i
transposeCV =
   3.0000 + 2.0000i   4.0000 + 6.0000i
dotApostropheCV =
   3.0000 + 2.0000i   4.0000 + 6.0000i
apostropheCV =
   3.0000 - 2.0000i   4.0000 - 6.0000i
```

This means, that if we use an operator as a short-cut to the transpose, it is good practice to always use the dot apostrophe operator.

Now, let's define some initial conditions for the path of our lizard.

```
lizardInitPos = zeros(noDimensions,1);
lizardInitVel = ones(noDimensions,1);
```

This shows us two useful functions for defining uniform zero or one matrices. Here, the initial position is a zero column-vector and the initial velocity is a one column-vector. There would be an equivalent function `rand(column size, row size)`, which would give a matrix of random numbers.

The lizard, will be more aggressive when he's able to see the beetle, and is likely to pounce. But, he will be less aggressive when he's farther away from the beetle, and is unsure whether the beetle will move much. So, it would make sense that his acceleration is consistently given by:

$$\ddot{\boldsymbol{p}}_{\text{lizard}} = 4\frac{\boldsymbol{p}_{\text{beetle}} - \boldsymbol{p}_{\text{lizard}}}{||\boldsymbol{p}_{\text{beetle}} - \boldsymbol{p}_{\text{lizard}}||_2^2 + 0.1}$$

We can add this to *tutorial.m* as:

```
lizardInitAcc = 4 * (beetlePosition - lizardInitPos)/(norm(beetlePosition - lizardInitPos)^2 + 0.1);
```

Notice the usage of parenthesis. Elemenary-school arithmetic, in general, behaves the way it was taught in elementary-school. That is, operations are performed in the following order:

1. paranthesis from inside to outside
2. exponentials from left to right
3. multiplication and division from left to right
4. addition and subtraction from left to right

Supporting Features

Structures

Matlab is typically written as a Functional language, but there is an option to use a more Object Oriented perspective. This object oriented perspective uses 'objects' that are called 'structures.' These structures are created automatically when their first component is defined. For example, we can create a structure 'lizard' with certain characteristics:

```
lizard.NoseToTailLength = 12.0e-2; %[m], 10 cm long
lizard.LegLength = 4.0e-2; %[m], 4 cm from toe to toe on one "axel".
lizard.position = lizardInitPos;
lizard.velocity = lizardInitVel;
lizard.acceleration = lizardInitAcc;
```

If we then query our lizard in the Command Window, we get:

```
>> lizard
lizard =
    NoseToTailLength: 0.1200
           LegLength: 0.0400
            position: [2x1 double]
            velocity: [2x1 double]
        acceleration: [2x1 double]
```

Help and Documentation

Next, we would like to find a lizard-fixed coordinate axis $\hat{\boldsymbol{e}}_1$ and $\hat{\boldsymbol{e}}_2$, which would be distinct from the earth-fixed coordinates $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$ used so far. These are pictured in Figure 1.

Let's assume that the lizard is always oriented in the direction of velocity, so that it's nose always points where it is instantaneously moving. Then, we need to find the angle θ by which the lizard-fixed frame needs to be rotated to find the earth-fixed frame. We know that this θ can be found with the arctangent of the velocity components:

$$\theta = \arctan\left(\frac{\dot{\boldsymbol{p}}_x}{\dot{\boldsymbol{p}}_y}\right)$$

But, what do we do if we don't know which Matlab function can be used for the arctangent? Particularly: we want the four-quadrant designating arctangent, and not the simple arctangent.

The first tool is the Mathworks website which includes an easily searchable documentation. In this case, we can search some phrase containing 'quadrant' or 'four' and 'tangent', as in Figure 3, and quickly get the link `http://de.mathworks.com/help/matlab/ref/atan2.html`.

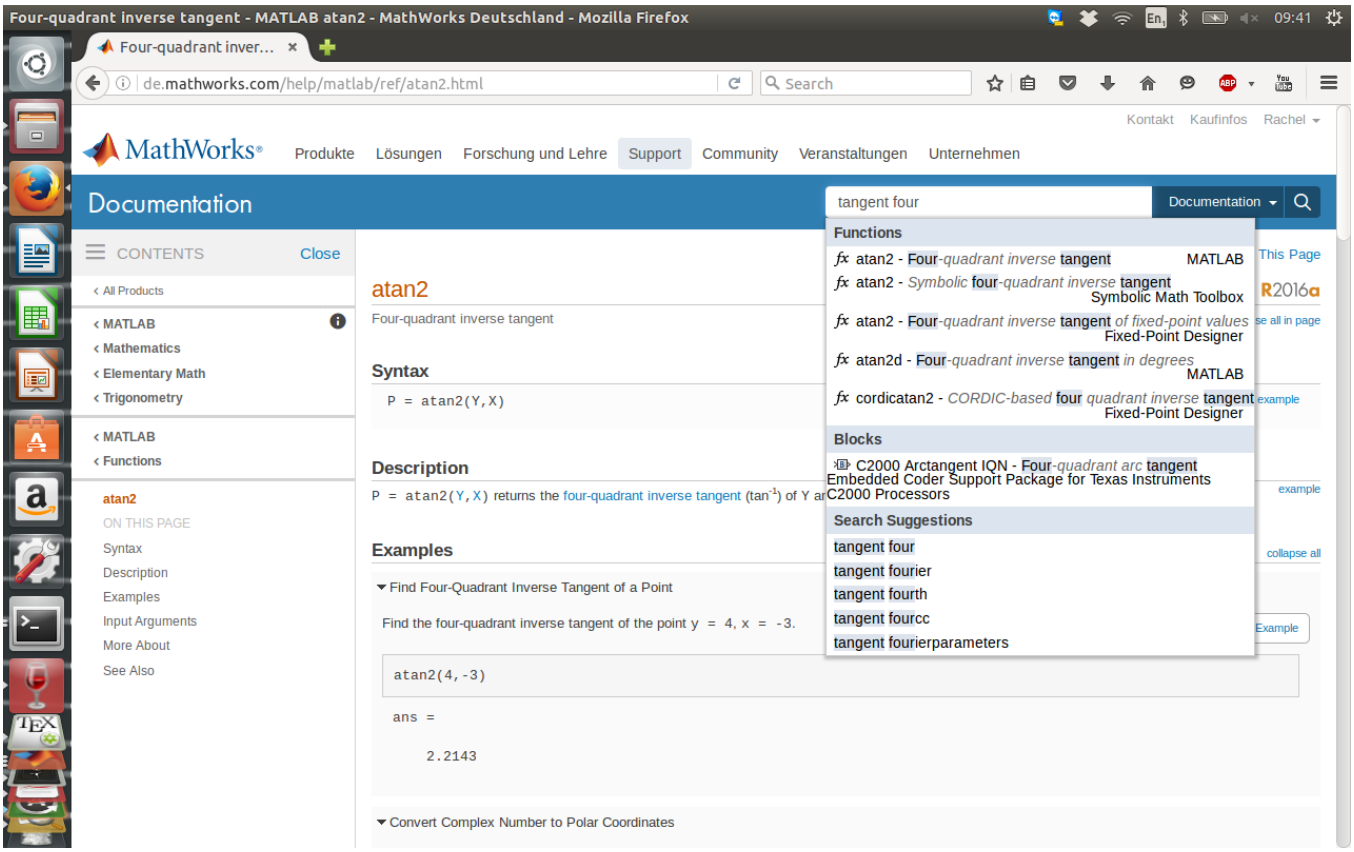

Figure 3: Using the Mathworks documentation to find a Matlab function.

We can get the equivalent page offline as in the online documentation by using the function `doc` in the Command Window, with or without the searched-function name.

```
>> doc % gives the index page for the documentation
>> doc atan2 % gives the documentation for the atan2 function
```

When we know about which function we want to use, but we're not sure exactly how to use it, we can use the `help` function in Matlab itself:

```
>> help atan2
 atan2  Four quadrant inverse tangent.
    atan2(Y,X) is the four quadrant arctangent of the elements
    of X and Y. -pi <= atan2(Y,X) <= pi.

    See also atan, atan2d.

    Reference page for atan2
    Other functions named atan2
```

Now we can use this `atan2` function to generate the rotation matrix R which transforms the lizard-fixed frame to the earth-fixed frame:

```
theta = atan2(lizard.velocity(1), lizard.velocity(2));
R = [sin(theta), -cos(theta); cos(theta), sin(theta)];
disp(R)
```

The function `disp` is one of Matlab's print functions, specialized for numerical values. Then, we have the output:

```
    0.7071   -0.7071
    0.7071    0.7071
```

Let's quickly double-check some behavior of multiplication with the rotation matrix R, namely:

```
R*I

ans =
    0.7071   -0.7071
    0.7071    0.7071
```

```
R.*I

ans =
    0.7071         0
         0    0.7071
```

```
R.' * R

ans =
     1     0
     0     1
```

Now our two lizard-fixed coordinate bases $\hat{\boldsymbol{e}}_1$ and $\hat{\boldsymbol{e}}_2$ are found with the first and second column of our rotation matrix:

```
ehat1 = R(:,1);
ehat2 = R(:,2);
```

Debugging

Let's define some geometric points for our lizard:

```
nose = R * [lizard.NoseToTailLength/2; 0] + lizard.position;
tail = R * [-lizard.NoseToTailLength/2; 0] + lizard.position;

leftFrontLeg = R * [0; -lizard.LegLength/2] + lizard.position + 0.35*lizard.NoseToTailLength*ehat1;
rightFrontLeg = R * [0; lizard.LegLength/2] + lizard.position + 0.35*lizard.NoseToTailLength*ehat1;

leftBackLeg = R * [0; -lizard.LegLength/2] + lizard.position - 0.25*lizard.NoseToTailLength*ehat1;
rightBackLeg = R * [0; lizard.LegLength/2] + lizard.position - 0.25*lizard.NoseToTailLength*ehat1;
```

Say now that we're unsure about what we've just coded, and we want to debug. There are two options.

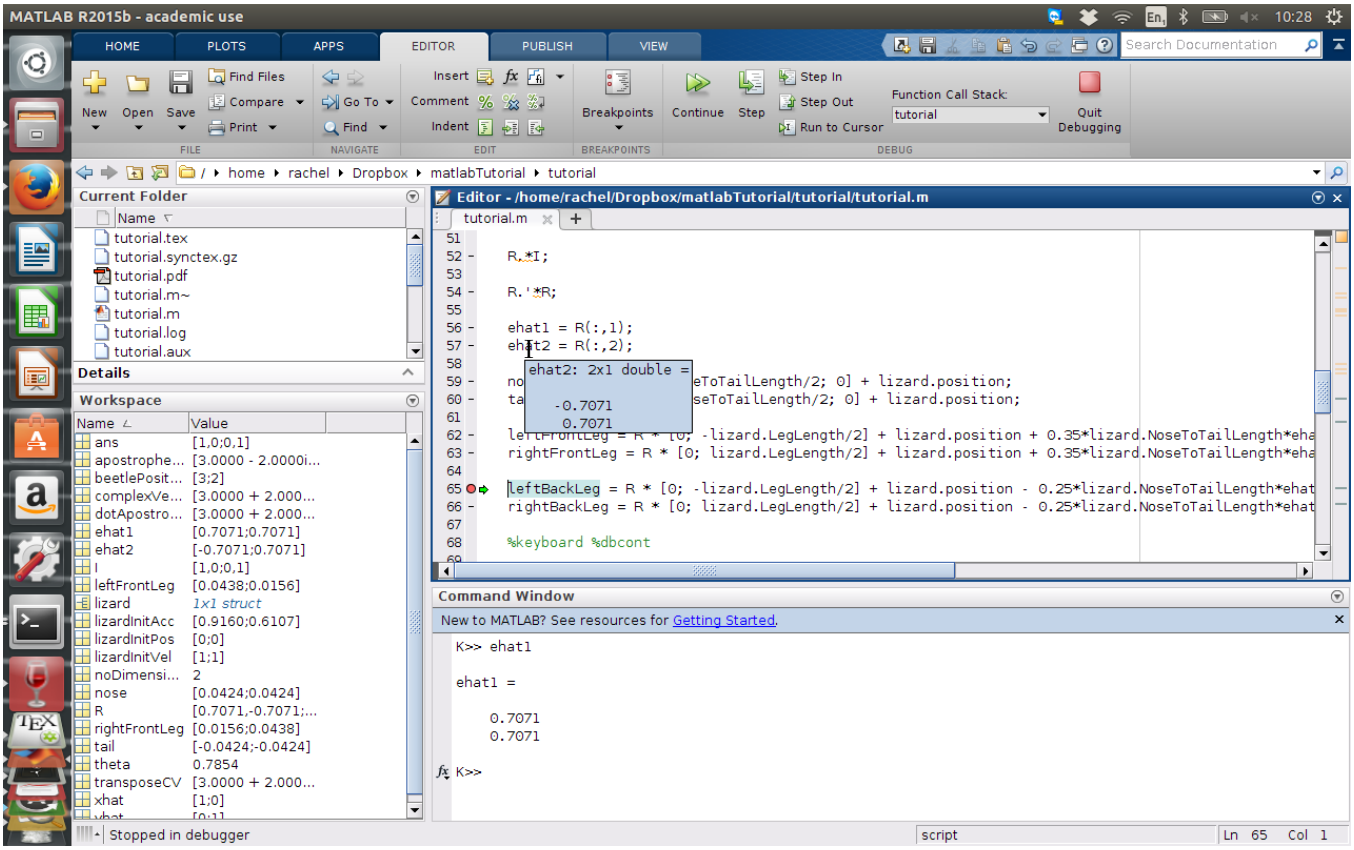

Figure 4: Debugging in Matlab.

The first debugging option is to add the function `keyboard` to *tutorial.m*. The `keyboard` function pauses the execution of the code, and allows us to access the current state from keyboard commands in the Command Window. While we're in keyboard mode, the command prompt symbol is `K>>`. So, we can - for example - query the nose location in the middle of running:

```
K>> nose
```

```
nose =
    0.0424
    0.0424
K>>
```

Further, while in this debugging mode, we can hover the mouse over a variable in the already evaluated code, and a little pop-up window will display the value of the variable.

While we're in keyboard mode, the bottom left of the Matlab frame will say 'Waiting for input.' To leave this keyboard mode, we can either press the green double arrow 'Continue' button in the toolbar, or enter the function `dbcont` in the Command Window.

The second debugging option is the Matlab Debugger. To use the Debugger, set a breakpoint in the code by clicking on the little dash next to the line-number in the Editor window. The little dash will become a red circle. When we run the file, Matlab will again stop at the indicated line. We are again in keyboard mode, but the bottom left of the Matlab frame will say 'Stopped in debugger.' So, we can query from the Command window, and use the hover evaluation. We can again leave keyboard mode and resume program execution with the 'Continue' button or the `dbcont`.

Using the Matlab Debugger and keyboard mode are shown in Figure 4.

Visualization

Now let's make the plot in Figure 5, showing the lizard and the beetle:

```
xFrontLeg = [leftFrontLeg(1), rightFrontLeg(1)];
yFrontLeg = [leftFrontLeg(2), rightFrontLeg(2)];

BackLeg = [leftBackLeg(1), rightBackLeg(1); leftBackLeg(2), rightBackLeg(2)];

figgy = 1; figure(figgy);
hold on;
plot(beetlePosition(1), beetlePosition(2), 'ro');
plot([tail(1), nose(1)], [tail(2), nose(2)], 'b');
plot(xFrontLeg, yFrontLeg, 'b');
plot(BackLeg(1,:), BackLeg(2,:), 'b');
legend('Beetle', 'Lizard', 'Location','southeast');
title('The lizard and the beetle');
xlabel('x [m]');
ylabel('y [m]');
axis equal
hold off;
print('lizardAndBeetle','-depsc')
```

With the `plot` function, the x-components of the data are the first input, then the y-components of the data, and then (optionally) a marker descriptor with a specified color and shape. The `hold` option prevents overwriting, `axis equal` will scale both axes equally, and `print('lizardAndBeetle','-depsc')` will save the figure as *lizardAndBeetle.eps*. More information can be found from the links from the page `http://de.mathworks.com/help/matlab/graphics.html`.

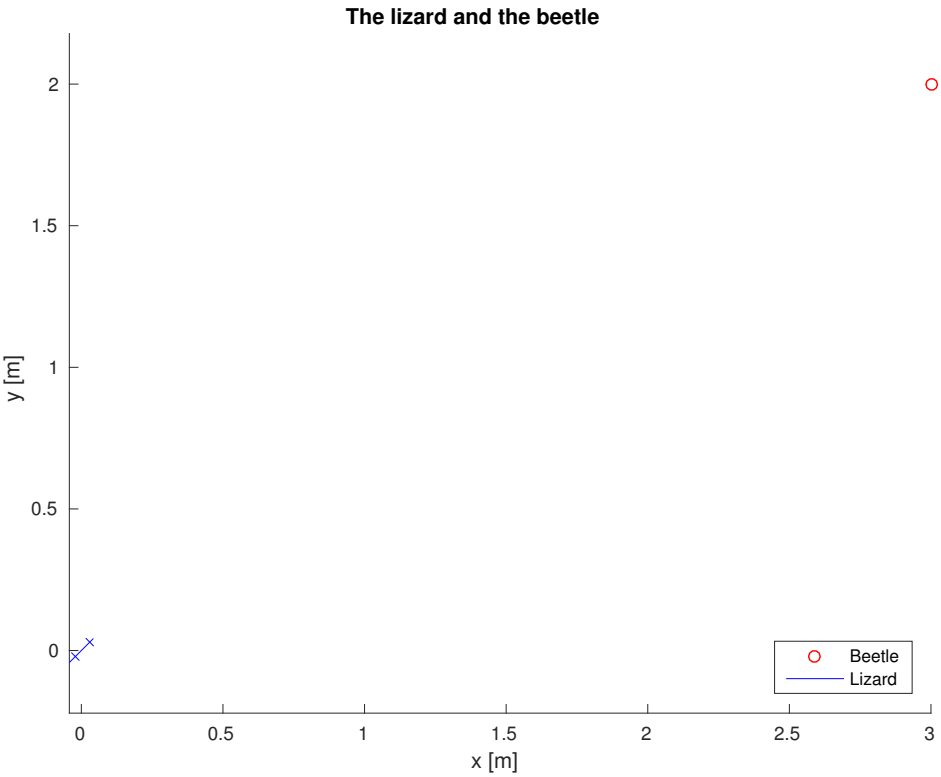

Figure 5: A Matlab Plot.

Functions

Now, let's write a function that will help our lizard to take a step forwards. In *tutorial.m*, this will look like:

```
dt = 0.01; %[s]
[ newPosition, newVelocity ] = takeStep(lizard.position, lizard.velocity, lizard.acceleration, dt);
lizard.position = newPosition;
lizard.velocity = newVelocity;
```

And we need to create this function `takeStep` in a new file *takeStep.m*. It's important for evaluation for the function name and the file name to be the same. Similarly, it's important - and Matlab won't explicitly return an error! - that the function names within a folder be unique.

Then, let's create this `takeStep` function:

```
function [ posEnd, velEnd ] = takeStep(pos, vel, acc, dt)
%takeStep: For constant acceleration and a known time step, determine the
% integrated position and velocity result.

posEnd = pos + vel * dt + 1/2 * acc.^2 * dt;
velEnd = vel + acc * dt;

end
```

Notice that the function ends with the `end` statement, and that the returned values have to have the same names in the function body.

Loops

Now, let's put the stepping functions into a loop where we store the position of the lizard, and plot the path, so that we can watch our lizard catch his dessert!

Matlab has `for` and `while` loops, as well as general `if-elseif` conditionals. Here we're going to use a `while` loop. More information on the others can be found at
`http://de.mathworks.com/help/matlab/matlab_prog/loop-control-statements.html`.

```
% initialize the index counter and the history storage matrix
idx = 1;
posHistory = lizard.position;

% two conditions for the loop, joined by an "and" &&  ("or" would be || )
% 1) idx < 200: a maximum iteration count
% 2) norm(beetlePosition - lizard.position) > 0.01: the lizard has not yet caught the beetle
while(idx < 200 && norm(beetlePosition - lizard.position) > 0.01)

    % take our step
    [ newPosition, newVelocity ] = takeStep(lizard.position, lizard.velocity, lizard.acceleration, dt);
    lizard.position = newPosition;
    lizard.velocity = newVelocity;
    lizard.acceleration = 4 * (beetlePosition - lizard.position)...
        /(norm(beetlePosition - lizard.position)^2 + 0.1);

    % save the lizard's position in a new column after the last column
    posHistory(:,end+1) = lizard.position;

    pause(1.0e-5); % useful so that you can Ctrl-C out of an unsuccessful (or infinite) loop.

    idx = idx+1; % update the counter
end


theta = atan2(lizard.velocity(1), lizard.velocity(2));
R = [sin(theta), -cos(theta); cos(theta), sin(theta)];
ehat1 = R(:,1);
ehat2 = R(:,2);

nose = R * [lizard.NoseToTailLength/2; 0] + lizard.position;
tail = R * [-lizard.NoseToTailLength/2; 0] + lizard.position;

leftFrontLeg = R * [0; -lizard.LegLength/2] + lizard.position + 0.35*lizard.NoseToTailLength*ehat1;
rightFrontLeg = R * [0; lizard.LegLength/2] + lizard.position + 0.35*lizard.NoseToTailLength*ehat1;

leftBackLeg = R * [0; -lizard.LegLength/2] + lizard.position - lizard.NoseToTailLength/4*ehat1;
rightBackLeg = R * [0; lizard.LegLength/2] + lizard.position - lizard.NoseToTailLength/4*ehat1;

figgy = 2; figure(figgy);
hold on;
plot(posHistory(1,:), posHistory(2,:), 'b--');
plot([tail(1), nose(1)], [tail(2), nose(2)], 'b');
plot([leftFrontLeg(1), rightFrontLeg(1)], [leftFrontLeg(2), rightFrontLeg(2)], 'b');
plot([leftBackLeg(1), rightBackLeg(1)], [leftBackLeg(2), rightBackLeg(2)], 'b');
plot(beetlePosition(1), beetlePosition(2), 'ro');
title('The path to the beetle');
xlabel('x [m]');
ylabel('y [m]');
axis equal
hold off;
print('lizardPath','-depsc')
```

Snap! Yes, the beetle is delicious...

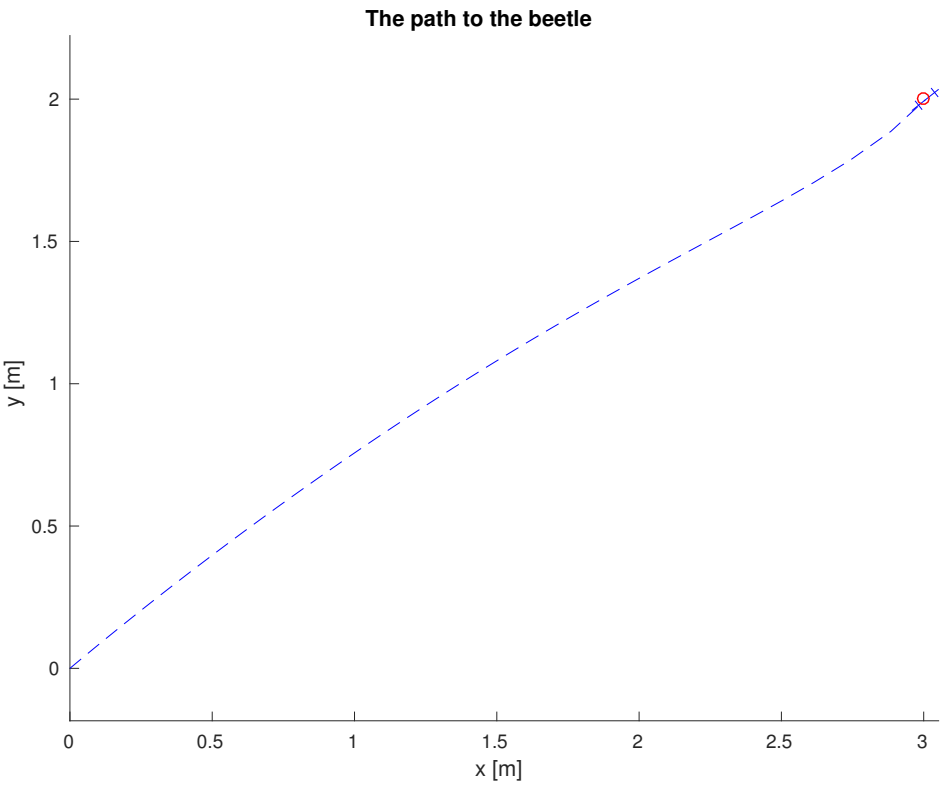

Figure 6: The lizard eats dessert.