

# **ELECTRONIC COMPASS TILT COMPENSATION**

## **Version N**

This document accompanies an Octave tutorial. It contains the text and the graphical output of the tutorial. It gives a first idea what all this is about. Still, it's much more fun to actually play with the equations. And if you are interested in compass development, you will appreciate the simulation environment. So please setup Octave and run the tutorial scripts.

The files part1.m to part4.m are both, the text of a tutorial as well as the tutorials Octave scripts. You may just read through them though it is quite handy to see the math actually executed. Walk your way through the parts. Part 1 starts with the Octave setup.

Arnold Neugebauer

# Part 1

## Motivation

This is a tutorial dealing with the math behind tilt compensation in an electronic compass.

I have seen several datasheets from sensor manufacturers stating inaccurate equations and/or missing the underlying conventions (axes, angles, direction of rotations). That is why I will explain the topic starting from the basics to the final equations.

The less abstract Euler angles (or Euler's angles or Eulerian angles) are used, choosing one of several possible conventions for them. Note that another approach uses quaternions instead.

## Setup Octave

Octave is a program for performing numerical computations. The homepage is [www.gnu.org/software/octave](http://www.gnu.org/software/octave). Download and execute the installation package. For Windows this is currently Octave-3.0.5\_i686-pc-mingw32\_gcc-4.3.0\_setup.exe.

You may want to alter some settings. The script

<install dir>\share\octave\3.0.5\m\startup\octaverc

is executed on every startup of Octave. You can for example shorten the prompt with PS1( "> " ).

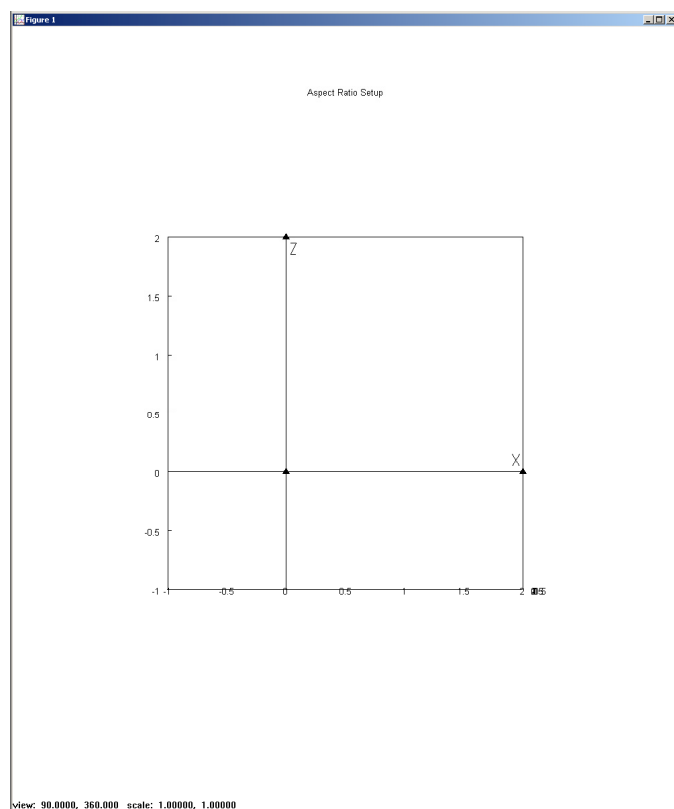
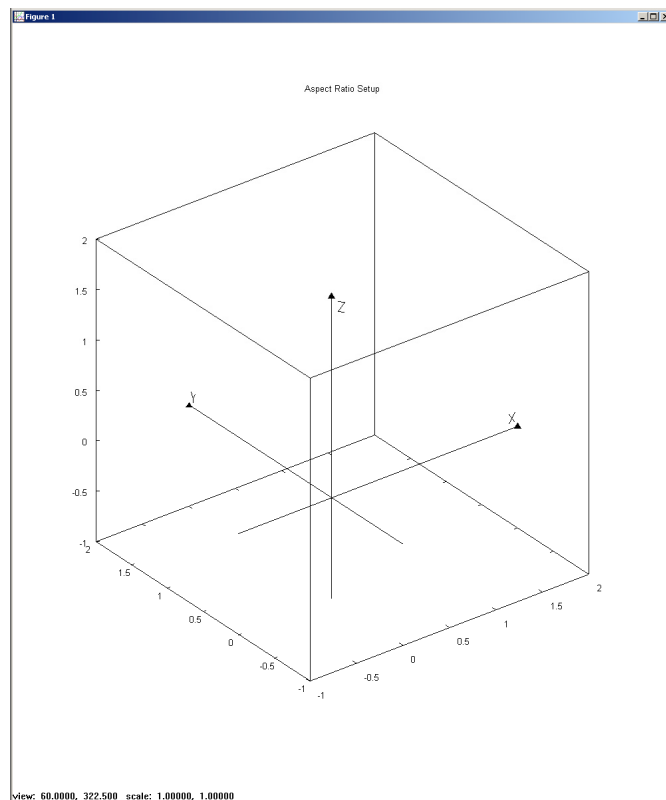
To set the startup path to the directory of your own scripts simply alter the properties of the Octave icon. Right click on it, click "Properties" and then edit the "Start In:" folder.

For the graphical output of this tutorial an aspect ratio of 1:1 is best to see the drawings undistorted. I have'nt found a way to control the aspect ratio within Octave. A workaround is to set the Gnuplot window to a fixed size. The GraphSize is set in Documents and Settings\<user>\Application Data\wgnuplot.ini. (If the file does not exist generate it by selecting Options->Update wgnuplot.ini in the menu of a plot.)

The following should create an undistorted output. You can rotate the drawing with the left mouse button pressed. Rotate it until you get a 2D view on one plane (for example the x-y-plane). Alter GraphSize until the following creates a graphical output in which the axes in vertical and horizontal direction are displayed with the same length.

```
clear                %undefines all variables
hold( "off" )        %next plot command will delete former output
rot = eye( 3 ); %see next part for details
plotaxes( rot, 3, 0.05, 0, "k" )
title( "Aspect Ratio Setup" )
```

A script, for example the part1.m you are currently reading, is simply executed by entering its name "part1" on the command line of Octave.

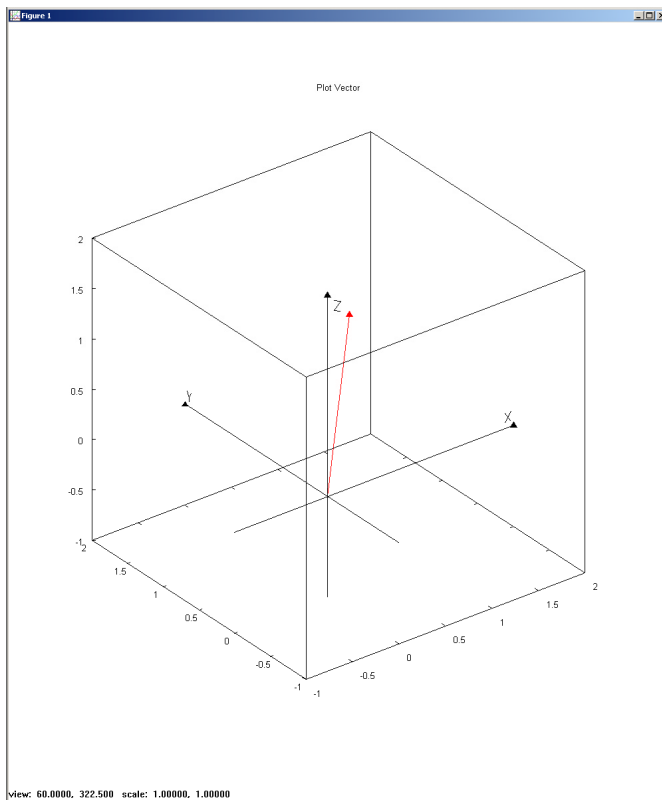


## Part 2

### Basics

To display results we use vectors a lot. A vector in this tutorial is represented by its start and its end point. It is organized as [xstart ystart zstart; xend yend zend]. To plot a vector in this form, the function `plotvector()` has been implemented. To first plot our coordinate system `plotaxes()` can be used, which in turn calls `plotvector()` three times. The arguments of `plotaxes` are explained later on.

```
clear
hold( "off" )
rot = eye( 3 );           %see below for details
plotaxes( rot, 3, 0.05, 0, "k" ) %k=black
hold( "on" )              %following plot commands will add to the
former output
vector = [ 0, 0, 0; 1 1 1 ];
plotvector( vector, "r" )  %r=red
title( "Plot Vector" );
kbhit();                  %press return to continue the script
```



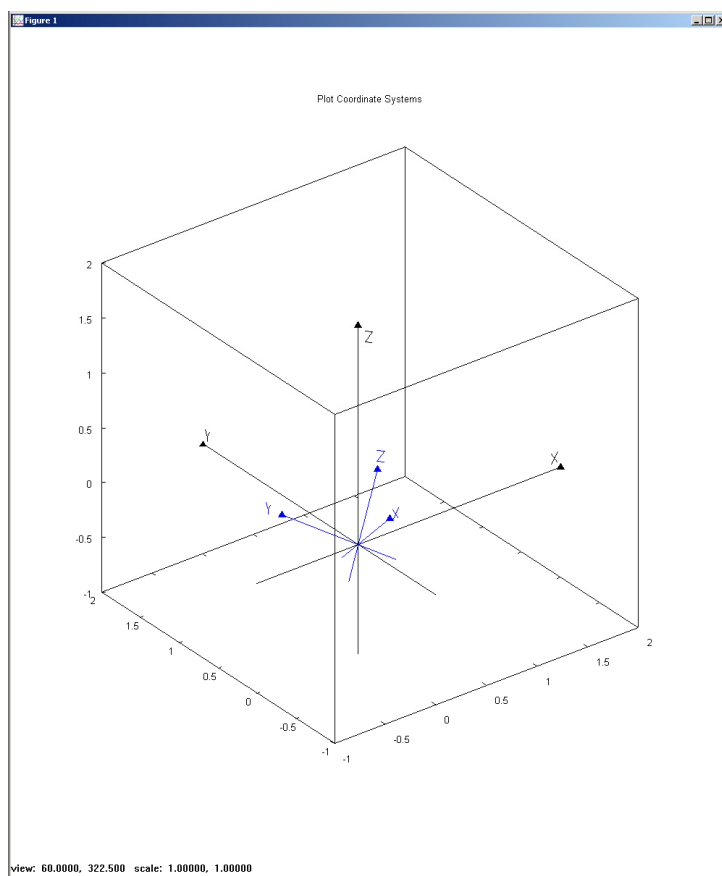
Tilt compensation of a compass is dealing heavily with rotations. A rotation in 3D can be described with a 3x3 rotation matrix. Every rotation can be split up into the three rotations about the axes of the coordinate system. To easily define a rotation about the x-, y- and z-axis respectively the three functions `rotx()`, `roty()` and `rotz()` have been implemented. The total rotation is then simply the multiplication of all three.

The function `plotaxes()` lets us define the attitude of the coordinate system to draw. The first parameter is the rotation, the second the length of each axis. The next 3

parameters define the size/position of the XYZ characters drawn at the axes and the color. Type 'help plotaxes' for details.

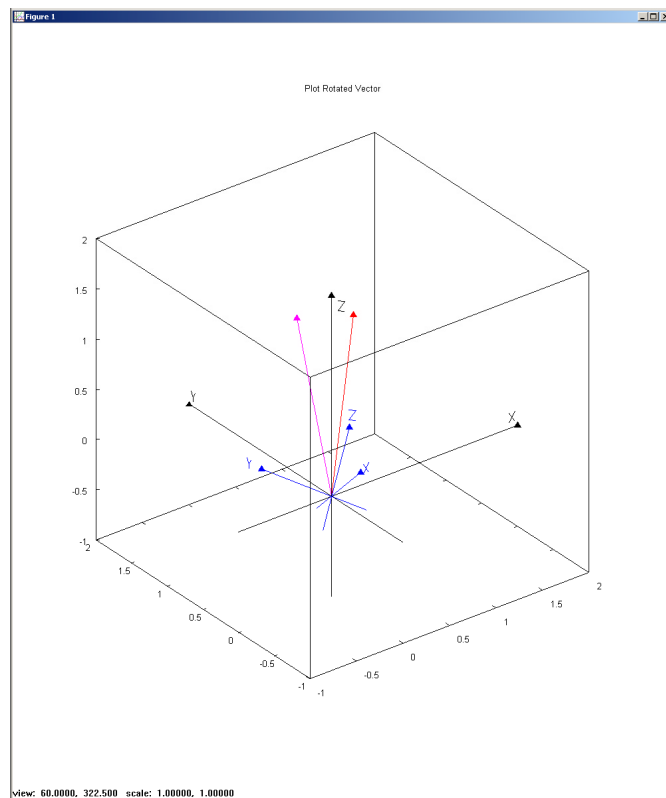
Lets again draw our black reference system. The rotation is 0, so we call plotaxes() with the unity matrix eye(3) as rotation. Then lets draw another blue coordinate system with some arbitrary attitude. Don't be bothered by the angles, direction of rotations etc. We'll get to that later.

```
clear
hold( "off" )
rot = eye( 3 );
plotaxes( rot, 3, 0.05, 0, "k" )
hold( "on" )
rot = rotx( 10 ) * roty( 10 ) * rotz( 30 );
plotaxes( rot, 1, 0.05, 0, "b" )
title( "Plot Coordinate Systems" );
kbhit();
```



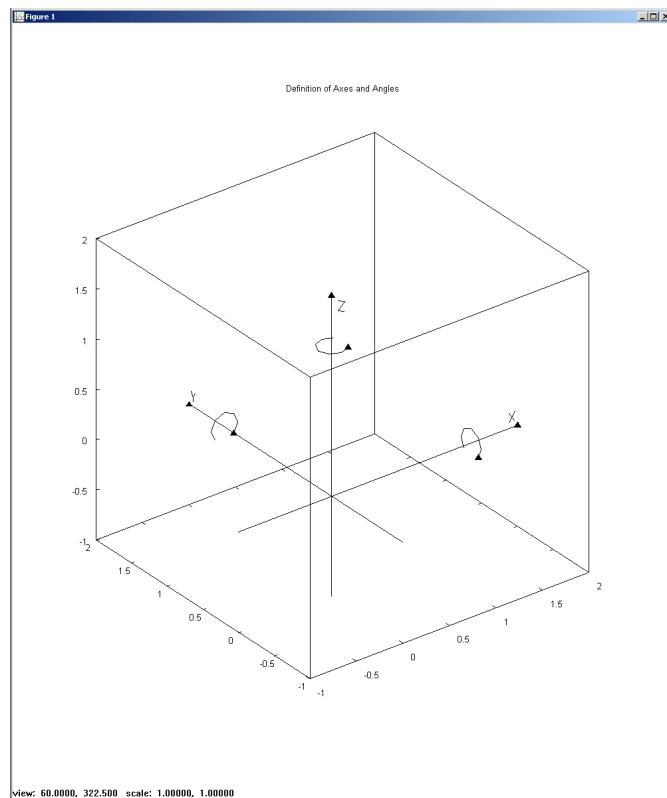
Octave lets us rotate a vector given in the form above simply by multiplying it with the rotation matrix, thus  $\text{vector\_rotated} = \text{vector} * \text{rot}$ . This will rotate both, the start and the end point, at the same time.

```
vector = [ 0, 0, 0; 1 1 1 ];
plotvector( vector, "r" )
vector_rotated = vector * rot;
plotvector( vector_rotated, "m" )
title( "Plot Rotated Vector" );
kbhit();
```



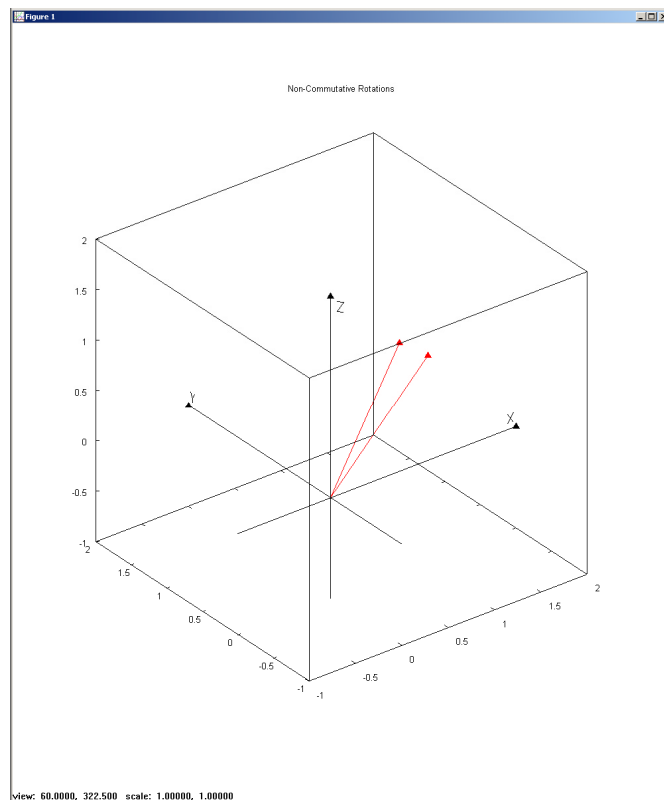
Ok, now let's go into that a bit more precisely. Like the definition of the coordinate system (relative orientation of the axes, where is positive etc.) it is also required to define the direction of rotations. We define the rotations mathematically positive. Seen the xy-plane from above (looking in the negative z direction), the rotation about z is counter-clockwise. The x-axis is 0 degrees, the y-axis 90 degrees. The same for the other axes.

```
draw_definition
title( "Definition of Axes and Angles" );
kbhit();
```



It is essential to understand that in a sequence of rotations the rotations can't be exchanged (non-commutative). In the following the result is not the same:

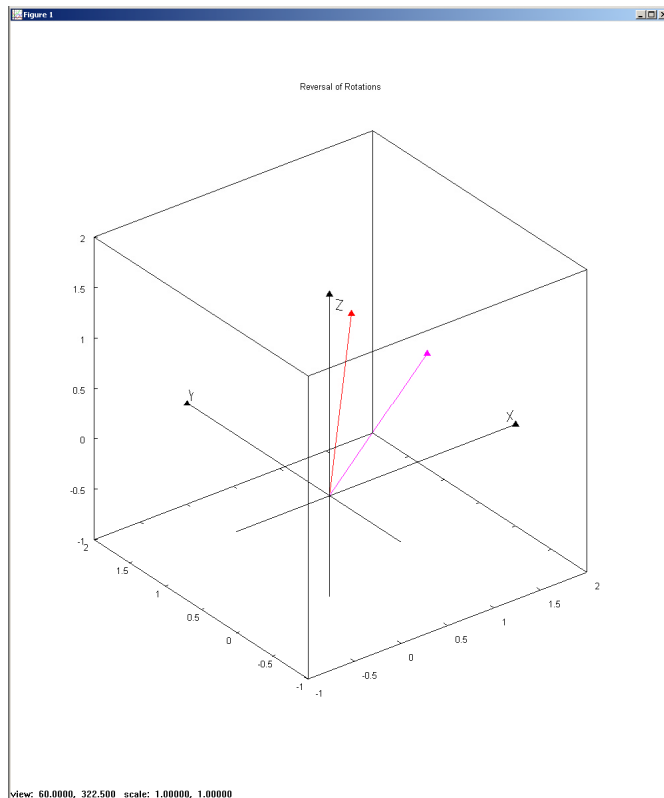
```
clear
hold( "off" )
rot = eye( 3 );
plotaxes( rot, 3, 0.05, 0, "k" )
hold( "on" )
rx = rotx( 20 );
ry = roty( 30 );
vector = [ 0, 0, 0; 1 1 1 ];
plotvector( vector * rx * ry, "r" ) %first rx, then ry
plotvector( vector * ry * rx, "r" ) %the other way round
title( "Non-Commutative Rotations" );
kbhit();
```



Thus we need to define which angle comes first. Intuitively the sequence is x, then y, then z. We will see later in the calculations why this is a clever choice. To reverse a rotation the angle is negated and the sequence is exchanged.

```
clear
hold( "off" )
rot = eye( 3 );
plotaxes( rot, 3, 0.05, 0, "k" )
hold( "on" )
vector = [ 0, 0, 0; 1 1 1 ];
vector_rot = vector * rotx( 20 ) * roty( 30 );
vector_rotback = vector_rot * roty( -30 ) * rotx( -20 )
plotvector( vector, "r" )
plotvector( vector_rot, "m" )
plotvector( vector_rotback, "k" )
title( "Reversal of Rotations" );
```





Furthermore we define that a vehicle or an airplane is heading in the positive x direction (if it is not rotated). The wings of an airplane would be level with the xy-plane, its longitudinal axis on the x-axis. In this initial position all angles are 0. The attitude, the deviation from the initial position, can now be described by three angles:

roll about the x-axis  
pitch about the y-axis  
yaw about the z-axis

It doesn't matter if an angle is given as 0..360 degrees or -180..+180 degrees. For example -90 is equal to +270 degrees. The trigonometric functions of Octave used later on output -180 to +180 degrees. So I prefer to use consistently -180..+180 degrees.

There is still an ambiguity left. The same attitude can be expressed by different combinations of roll, pitch and yaw.

Turning the vehicle upside down can be expressed by either pitch > +90 or roll > +-90 degrees. For example a pitch of 160 degrees is the same as rolling 180 degrees, then applying a pitch of 20 degrees and flipping the vehicle in the other direction. In both cases the attitude and the rotation matrix is the same:

roll = 0		roll = 180
pitch = 160	<=>	pitch = 20
yaw = 0		yaw = 180

When applying a pitch larger than +90 degrees the nose of the vehicle is suddenly pointing backwards. This changes the heading without altering yaw. Also the direction of roll is reversed. An ill-suited behaviour.

Roll, on the other hand, does not alter the heading. Even when turning the vehicle upside down with roll larger than +-90 degrees.

Therefore, by convention, pitch is limited while the full range for roll is admitted:

-180 < roll <= 180

```
-90 <= pitch <= 90
-180 < yaw <= 180
```

(To be exact: The combinations 0/+90/0 and 180/+90/180 describe the same attitude. Nevertheless they are both admitted. Why? +-90 is a special case anyway. Heading is not defined. Roll (and with that all other angles) can't be determined from the sensor outputs. We will see that later. So +-90 degrees has to be excluded from the final heading calculation anyway. Admitting both combinations allows to decide only on pitch during the conversion of combinations.)

By looking at some combinations describing the same attitude we can derive a method to convert between them. Converting an angle combination not conforming to the convention is done by flipping all directions and inverting pitch:

```
clear
disp( "Rotation matrices for equal angle combinations:" )
roll = 0
pitch = 160
yaw = 0
rot = rotx( roll ) * roty( pitch ) * rotz( yaw )
if( (pitch > 90) || (pitch < -90) )
    roll = roll + 180; %flip
    if( roll > 180 ) %pull into standard range
        roll -= 360;
    endif
    pitch = pitch + 180;
    if( pitch > 180 )
        pitch -= 360;
    endif
    pitch = -pitch; %invert pitch
    yaw = yaw + 180;
    if( yaw > 180 )
        yaw -= 360;
    endif
endif
roll
pitch
yaw
rot = rotx( roll ) * roty( pitch ) * rotz( yaw )
```

Later on in the next parts of this tutorial we will do both, calculating sensor outputs from angles given and - the other way round - calculating angles from sensor outputs. You can go into the calculations with any arbitrary angle combination. The angles calculated, however, will conform to the convention above. This is analog to the behaviour of trigonometric functions. The ranges of the inverse functions are subsets of the domains of the original functions.

With all these conventions the attitude, the deviation from the initial position, can now unambiguously be described.

Now that we are all setup we can go to the actual math. Tilt compensation is done in two steps. First the attitude of the vehicle or airplane is determined using the output of an acceleration sensor. This is described in part 3 of this tutorial. The second step is the heading calculation including the tilt compensation based on the attitude. Part 4 will go into details about this.

## Part 3

### Determining the Attitude

In the following we will derive a model. That is we will draw the ground-fixed reference system, the tilted vehicle-fixed system and acceleration vectors.

This is quite handy. We can play with the model and see what happens. Finally, by expanding the calculation with matrices and vectors, we get the equations to implement in the electronic compass.

The following defines an arbitrary attitude and plots the ground-fixed as well as the vehicle-fixed system. Also the positive and negative rotation matrices are calculated beforehand.

```
clear
hold( "off" )
plotaxes( eye(3), 30, 0.5, 0, "k" )
hold( "on" )

roll  = 30; %x
pitch = 10; %y
yaw   = 10; %z

rxp = rotx( roll );
ryp = roty( pitch );
rzp = rotz( yaw );
rxn = rotx( -roll );
ryn = roty( -pitch );
rzn = rotz( -yaw );
rp  = rxp * ryp * rzp; %first rotate about x, then y, then z
rn  = rzn * ryn * rxn; %sequence the other way round!

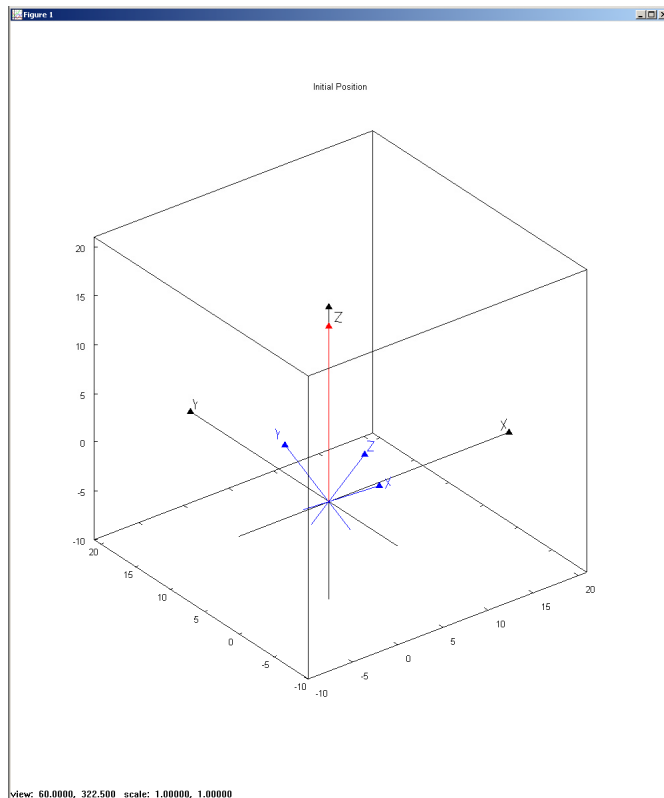
plotaxes( rp, 10, 0.5, 0, "b" )
```

Next the acceleration vector. It points up. We just make everything in the model positive for now. It's easier to understand. Later on we can make it point down just by exchanging signs.

The magnitude is arbitrarily chosen. In a real system the magnitude is the maximum sensor output in the earth field found out by calibration.

The postfix `_g` means that a vector is described based on the ground-fixed coordinate system. Variables with postfix `_v` are described based on the vehicle-fixed coordinate system.

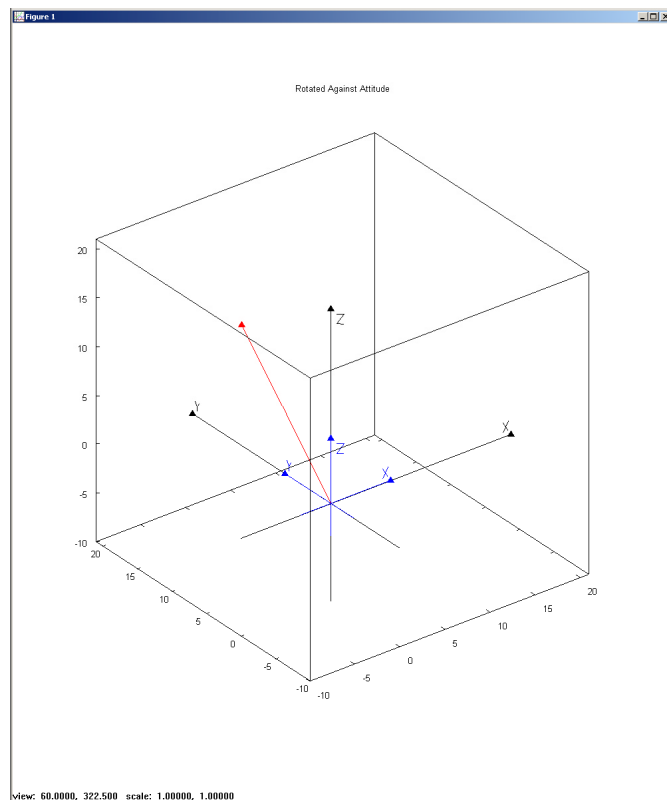
```
acc_strength = 18;
acc_g = [ 0 0 0; 0 0 acc_strength ];
plotvector( acc_g, "r" )
title( "Initial Position" )
kbhit();
```



At this point we have our reference system and a vehicle system somehow tilted in the 3D space.  $\text{acc\_g}$  is an object in the space. For calculations in the vehicle system, the general trick is to transform the tilted vehicle system to the reference system. Calculations are done conveniently there in a standard way. All objects are transformed in the same way, so that their relative position to the vehicle system remains constant. After the calculations have been done the vehicle system and all objects are transformed back / tilted again. The transformations are coordinate transformation or rotations respectively. Calculations here are just the simple split-up of a vector in its three components.

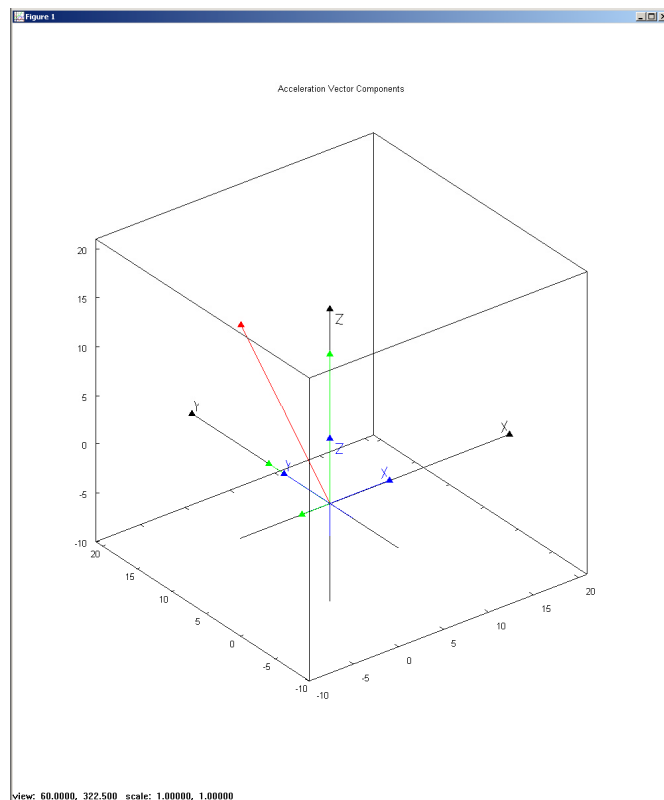
The next Octave commands draw the vehicle-fixed system and the acceleration vector again. This time rotated against the attitude of the vehicle. The vehicle-fixed system becomes the same as the ground-fixed system. The acceleration vector is now slanted but its position relative to the vehicle-fixed system has not changed.

```
hold( "off" )
plotaxes( eye(3), 30, 0.5, 0, "k" )
hold( "on" )
plotaxes( rp * rn, 10, 0.5, 0, "b" ) %same as plotaxes( eye(3), 10, 0, "b" )
acc_v = acc_g * rn;
plotvector( acc_v, "r" )
title( "Rotated Against Attitude" )
kbhit();
```



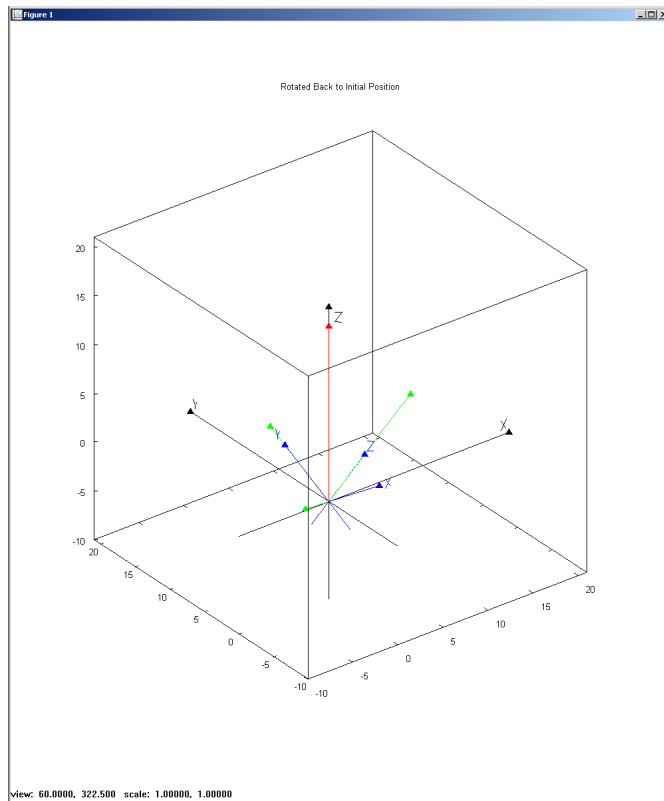
Now the acceleration vector can be split up into its components parallel to the axes. The magnitude of these vectors is the output of the acceleration sensors experienced. The three vectors along the axes are:

```
accx_v = [ 0, 0, 0; acc_v(2, 1), 0, 0 ];
accy_v = [ 0, 0, 0; 0, acc_v(2, 2), 0 ];
accz_v = [ 0, 0, 0; 0, 0, acc_v(2, 3) ];
plotvector( accx_v, "g" )
plotvector( accy_v, "g" )
plotvector( accz_v, "g" )
title( "Acceleration Vector Components" )
kbhit();
```



Rotate components back to their original attitude:

```
hold( "off" )
plotaxes( eye(3), 30, 0.5, 0, "k" )
hold( "on" )
plotaxes( rp, 10, 0.5, 0, "b" )
plotvector( acc_g, "r" )
accx_g = accx_v * rp;
accy_g = accy_v * rp;
accz_g = accz_v * rp;
plotvector( accx_g, "g" )
plotvector( accy_g, "g" )
plotvector( accz_g, "g" )
title( "Rotated Back to Initial Position" )
kbhit();
```

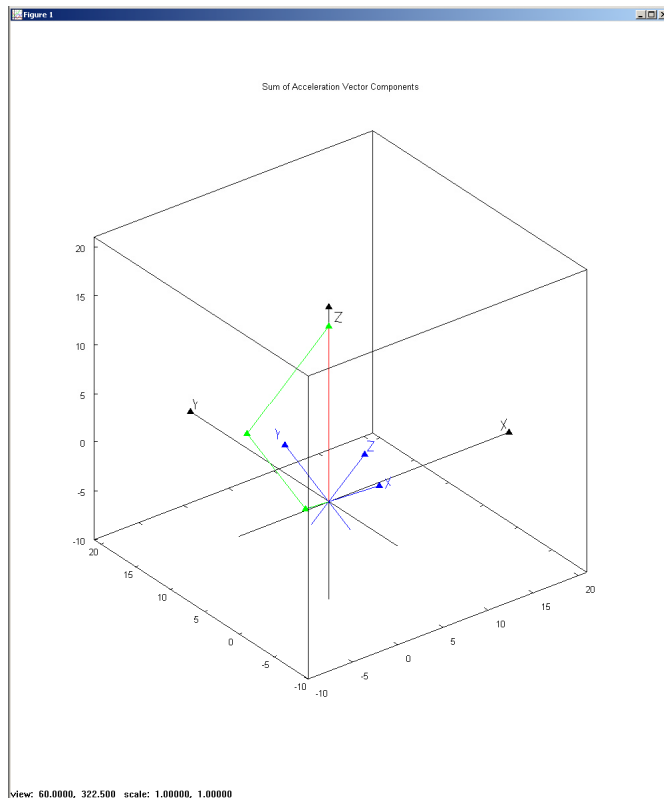


To see more precisely that the green components on the axes are in fact adding up to the red acceleration vector this can be drawn a little bit different.

Let's graphically add up the green vectors of the vehicle-fixed system. `accx_g` starting from zero, `accy_g` vector starting at end of `accx_g`, `accz_g` vector starting from there. The end point of these 3 vectors must be the end point of `acc_g`, every single vector pointing in the direction of the respective vehicle-fixed axis.

```
accy_g_offs = [ accx_g(2,:); accx_g(2,:) + accy_g(2,:) ];
accz_g_offs = [ accx_g(2,:) + accy_g(2,:); accx_g(2,:) + accy_g(2,:) +
accz_g(2,:) ];
hold( "off" )
plotaxes( eye(3), 30, 0.5, 0, "k" )
hold( "on" )
plotaxes( rp, 10, 0.5, 0, "b" )
plotvector( acc_g, "r" )
plotvector( accx_g, "g" )
plotvector( accy_g_offs, "g" )
plotvector( accz_g_offs, "g" )
title( "Sum of Acceleration Vector Components" )
kbhit();
```





What we are finally interested in is the relationship between the acceleration sensor outputs (the magnitude of the components seen on the vehicle-fixed axes) and the attitude of the vehicle.

The sensor outputs are `acc_v(2,:)`, forming the vector `acc_v` in the vehicle-fixed coordinate system. For convenience these abbreviations are used in the following:

```
accx = acc_v(2,1);
accy = acc_v(2,2);
accz = acc_v(2,3);
```

Expanding the equation  $\text{acc\_v} = \text{acc\_g} * \text{rn}$  from above leads to the final equations. Now we see why it was clever to choose the rotation sequence `xyz` for the vehicle. Rotating the other way round is `zyx`. Since `acc_g` is pointing from the origin upwards it is invariant to the first rotation about `z`. So we can leave that away, simplifying the equations enormously.

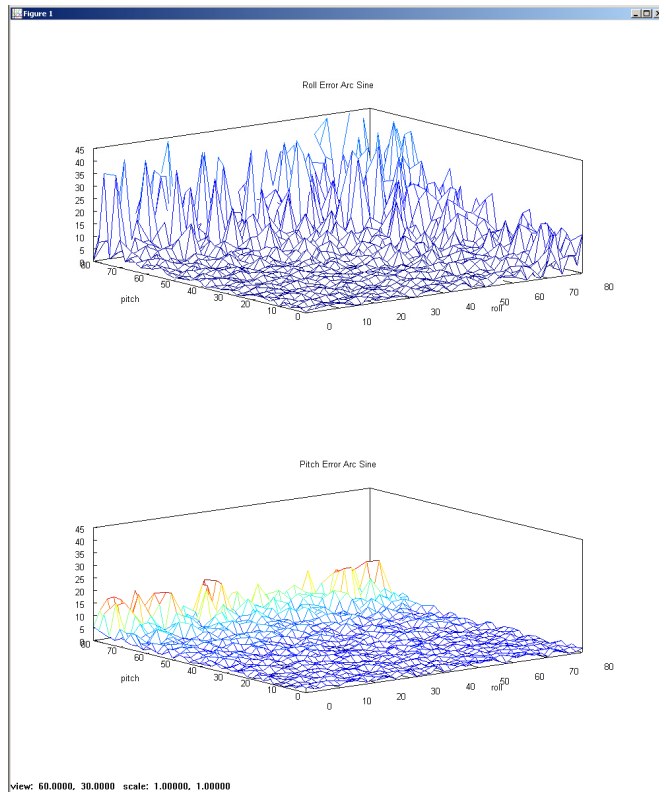
```
acc_v = acc_g * rn
=> [ accx, accy, accz ] = [ 0 0 acc_strength ] * ryn * rxn
=> accx =  acc_strength * sind( pitch )      (1)
    accy = - acc_strength * sind( roll ) * cosd( pitch ) (2)
    accz =  acc_strength * cosd( roll ) * cosd( pitch ) (3)
```

There are more equations than unknown variables what allows to solve in different ways. Simply taking (1) and (2) leads to the equations often found in the literature. The sign must be changed since in `rn` the angles are negative.

```
disp( "Roll and pitch calculated using arc sine:" )
pitch1 = - asind( accx / acc_strength );           %y
roll1   =  asind( accy / acc_strength / cosd(pitch1) ) %x
pitch1
kbhit();
```

However, doing this in the real world is disadvantageous. The slope of the asin function is strongly increasing with higher angles. Thus it gets very sensitive to noise, offset and gain errors. The following shows this by adding 10% noise to the sensor outputs (the high noise level is arbitrarily chosen but seems not to be unusual in a low-quality/low-cost circuit).

```
draw_asin_error()  
kbhit();
```

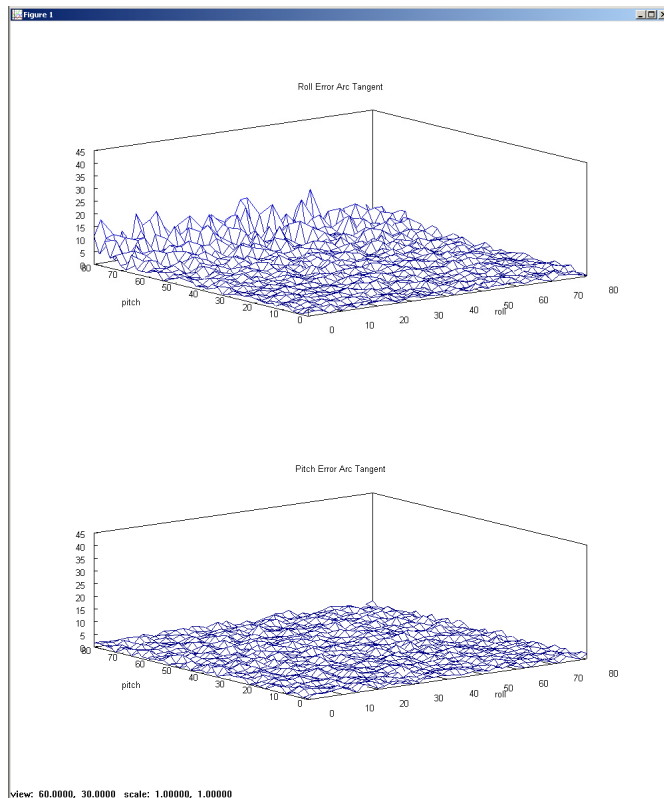


The output can be improved significantly. Dividing equation (1) by (3) and equation (2) by (3) leads to expressions with atan. The slope of the atan function is increasing later than the asin function.

Note that `acc_strength` is canceled down. So we don't need to know the absolute value from now on.

```
disp( "Roll and pitch calculated using arc tangent:" )  
roll1 = atand( accy / accz ) %x  
pitch1 = - atand( accx / accz * cosd(roll1) ) %y  
kbhit();
```

```
draw_atan_error()  
kbhit();
```



Still, the error with pitch near 90 degrees gets very high. Limiting the operation to about  $\pm 80$  degrees is acceptable for vehicles or handheld devices. Otherwise, however, more effort is needed to get around the noise near the poles of the trigonometric functions.

In any case, when dealing with a real system, proper noise filtering/damping of the sensor outputs has to be applied and a plausibility check to cancel out the acceleration of the vehicle itself.

Remember the conventions made for the output ranges in part 2 of this tutorial?

We targeted for  $-180 < \text{roll} \leq 180$  and  $-90 \leq \text{pitch} \leq 90$ . A pitch of exactly  $\pm 90$  degrees, however, leads to  $\text{accy} = \text{accz} = 0$ . Roll (and with that all other angles) can't be determined from the sensor outputs. So a pitch of  $\pm 90$  degrees has to be excluded from the calculations (by discarding measurements with  $\text{accy} = \text{accz} = 0$ ). Heading is not defined anyway in this case.

The last equations for roll and pitch are using atan with an output range of  $-90 < \text{angle} < 90$ . While this is perfect for pitch, we need to use atan2 for roll. atan2 is a variant of the atan function with the expanded output range of  $-180 < \text{angle} \leq 180$ . For details type 'help atan2' on the Octave command line.

Since atan2 is part of the C programming language's math.h standard library, it is often available. Also Octave features the built-in function. atan2d.m shows a possible implementation if it is not available on your system. (Be aware that sometimes implementations reverse the order of the parameters.)

Alter roll, pitch and yaw at the beginning of the script to see the effect:

```
disp( "Roll and pitch calculated using arc tangent 2:" )
roll1 = atan2d( accy, accz )           %x
```

```
pitch1 = - atand( accx / accz * cosd(roll1) ) %y
```

In this part we derived a handy model to play around with and to verify our calculations. It will also be used in the next part. And we derived two equations, enabling us to determine the tilt of the vehicle-fixed system from its acceleration sensor outputs. So let's start with the actual heading calculation in the next part.

## Part 4

### Heading Calculation

This part deals with the heading calculation. We will use the same approach as in the last part. First the whole flow is modeled by manipulating vectors and using Octave's matrix multiplications. Intermediate steps and results are visualized. Remember that you can drag the graphics around with the mouse. And be encouraged to alter the parameters and calculations. Play around with it to get a feeling of what's going on. In a second step, by expanding the calculation with matrices and vectors, we get the equations to implement in the electronic compass.

Again, the following defines an arbitrary attitude and plots the ground-fixed as well as the vehicle-fixed system. Also the positive and negative rotation matrices are calculated beforehand.

```
clear
hold( "off" )
plotaxes( eye(3), 30, 0.5, 0, "k" )
hold( "on" )

roll  = 30; %x
pitch = 10; %y
yaw   = 10; %z

rxp = rotx( roll );
ryp = roty( pitch );
rzp = rotz( yaw );
rxn = rotx( -roll );
ryn = roty( -pitch );
rzn = rotz( -yaw );
rp  = rxp * ryp * rzp; %first rotate about x, then y, then z
rn  = rzn * ryn * rxn; %sequence the other way round!

plotaxes( rp, 10, 0.5, 0, "b" )
```

Now the magnetic field vector. It points up. Like the acceleration vector we just make everything positive for now. (Later, pointing down with an inclination = 70 degrees is more appropriate.)

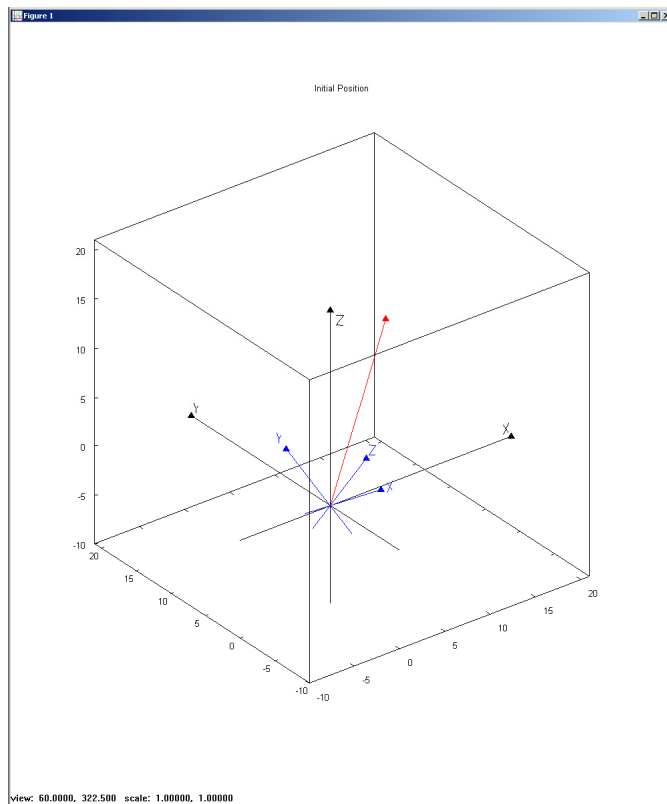
As of our definition from part 2, north is in the direction of the x-axis of the ground-fixed system. Given the parameters strength and inclination the following models the magnetic field vector in the xz-plane. The magnetic strength has been arbitrarily chosen to fit in our coordinate system drawn. In a real system the strength is the maximum sensor output in the earth field found out by calibration. A magnetic field vector with a negative inclination is pointing up (towards +z) per definition.

```
mfield_strength = 18;
mfield_inclination = -70; %northern europe would be about +70
degrees
mfieldx_g = mfield_strength * cosd( mfield_inclination );
mfieldd_g = 0;
mfieldd_g = sqrt( mfield_strength^2 - mfieldx_g^2 ); %derived from vector
length
if( mfield_inclination > 0 )
    mfieldd_g = -mfieldd_g;
```

```

endif
mfield_g = [ 0, 0, 0; mfieldx_g, mfielgy_g, mfieldz_g ];
plotvector( mfield_g, "r" )
title( "Initial Position" )
kbhit();

```



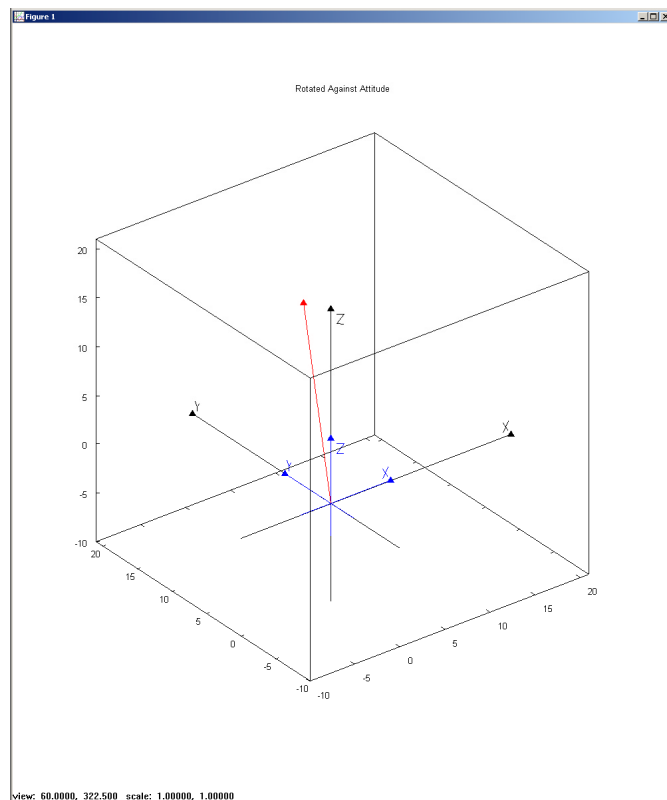
The question is now what sensor outputs we see on the axes of the vehicle-fixed system. The same trick as for the acceleration in part 3 is used. The tilted vehicle system and all objects are transformed to the ground-fixed reference system. The relative position is kept. Then calculations are executed and finally everything is transformed back.

The next commands draw the vehicle-fixed system and the magnetic field vector again. This time rotated in the opposite direction of the vehicle attitude. The vehicle-fixed system becomes the same as the ground-fixed system. The magnetic field vector is not in the xz-plane anymore but its position relative to the vehicle-fixed system has not changed.

```

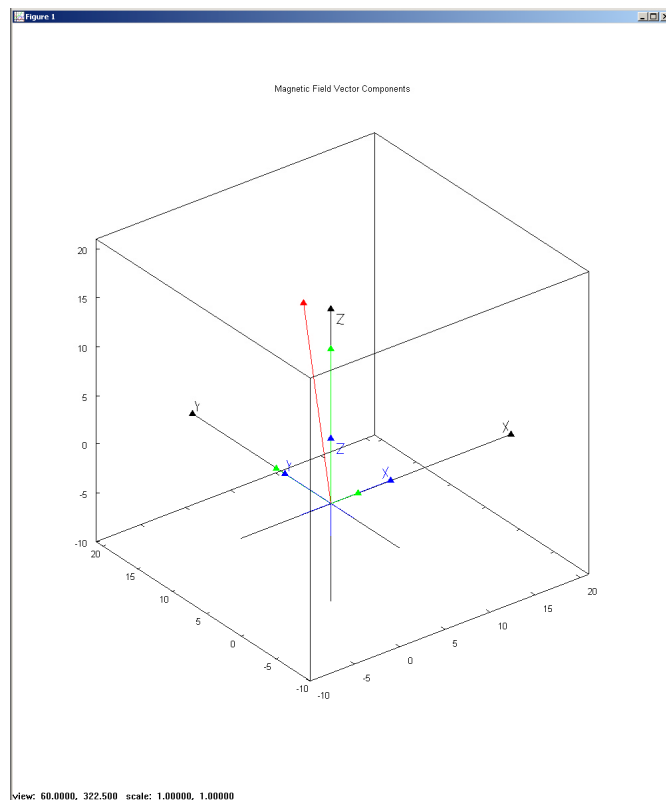
hold( "off" )
plotaxes( eye(3), 30, 0.5, 0, "k" )
hold( "on" )
plotaxes( rp * rn, 10, 0.5, 0, "b" ) %same as plotaxes( eye(3), 10, 0.5, 0,
"b" )
magn_v = mfield_g * rn;
plotvector( magn_v, "r" )
title( "Rotated Against Attitude" )
kbhit();

```



Now the magnetic field vector can be split up into its components parallel to the axes. The magnitude of these vectors is the output of the magnetic field sensors experienced. The three vectors along the axes are:

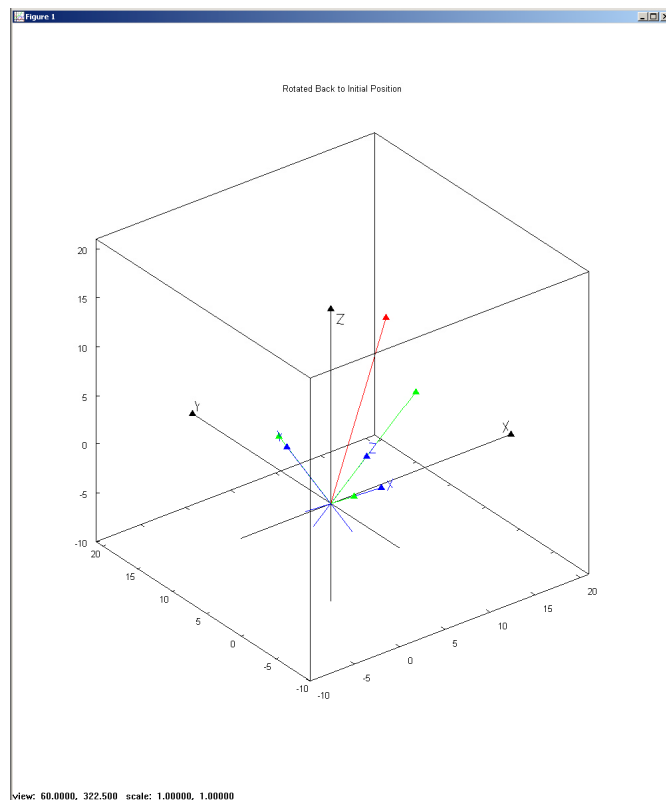
```
magnx_v = [ 0, 0, 0; magn_v(2, 1), 0, 0 ];
magny_v = [ 0, 0, 0; 0, magn_v(2, 2), 0 ];
magnz_v = [ 0, 0, 0; 0, 0, magn_v(2, 3) ];
plotvector( magnx_v, "g" )
plotvector( magny_v, "g" )
plotvector( magnz_v, "g" )
title( "Magnetic Field Vector Components" )
kbhit();
```



Rotate components back to their original attitude:

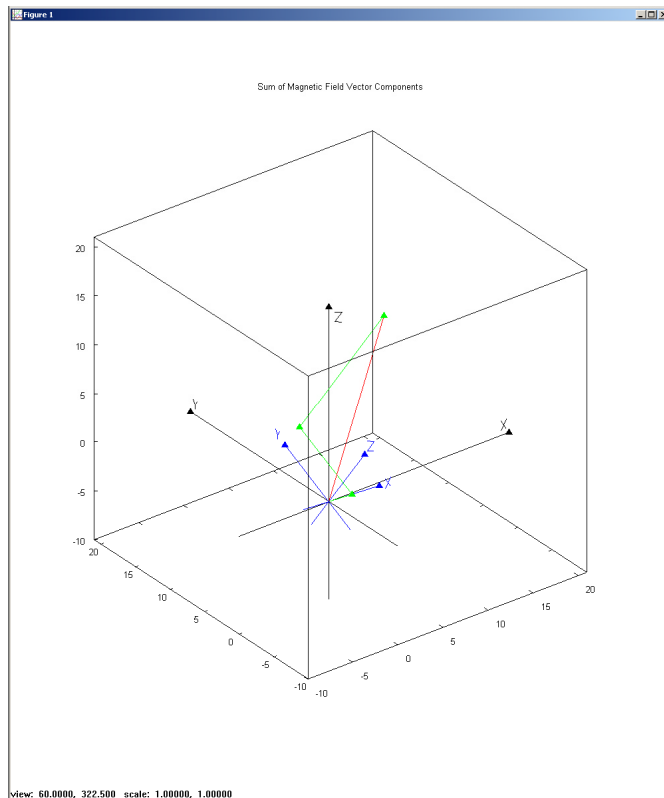
```
hold( "off" )
plotaxes( eye(3), 30, 0.5, 0, "k" )
hold( "on" )
plotaxes( rp, 10, 0.5, 0, "b" )
plotvector( mfield_g, "r" )
magnx_g = magnx_v * rp;
magny_g = magny_v * rp;
magnz_g = magnz_v * rp;
plotvector( magnx_g, "g" )
plotvector( magny_g, "g" )
plotvector( magnz_g, "g" )
title( "Rotated Back to Initial Position" )
kbhit();
```





Graphically add up the green vectors to see more precisely that they are in fact add up to the red magnetic field vector:

```
magny_g_offs = [ magnx_g(2,:); magnx_g(2,:) + magny_g(2,:) ];
magnz_g_offs = [ magnx_g(2,:) + magny_g(2,:); magnx_g(2,:) + magny_g(2,:) +
magnz_g(2,:) ];
hold( "off" )
plotaxes( eye(3), 30, 0.5, 0, "k" )
hold( "on" )
plotaxes( rp, 10, 0.5, 0, "b" )
plotvector( mfield_g, "r" )
plotvector( magnx_g, "g" )
plotvector( magny_g_offs, "g" )
plotvector( magnz_g_offs, "g" )
title( "Sum of Magnetic Field Vector Components" )
kbhit();
```



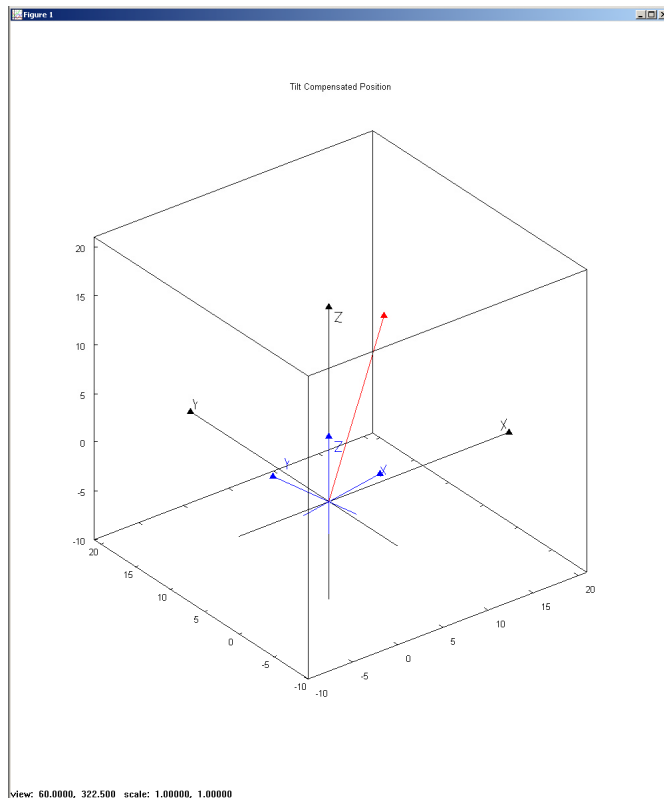
Ok - after having the model up and running we can now try to figure out the equations for the implementation. What we are finally interested in is the relationship between the magnetic field vector components seen in the vehicle-fixed system (the sensor outputs) and the heading.

In theory it would be possible to derive them directly from the model above (like we did for the attitude in part3). We would have to expand  $\text{magn\_v} = \text{mfield\_g} * \text{rn}$ . Unfortunately math with the full rotation  $\text{rn}$  is very hard to do. (For the attitude we were lucky - the acceleration vector only had a z-component and the rotation sequence had been chosen to support further.)

What helps is an intermediate step: First calculate the tilt-compensated sensor outputs and then do the final heading calculation without tilt in the xy-plane. This way the rotation about z is omitted in the first step to make the expansion easier. Then the z-component is left out since it does not contain any heading information, further simplifying things. In the following you will see what I mean.

For a better understanding let's draw the vehicle-fixed system in its tilt-compensated position. It is rotated only about z with  $\text{rzp}$ . The xy-plane is level with the xy-plane of the ground-fixed system. Here again we see why it was clever to choose the rotation sequence xyz for the vehicle. Only with z being the last rotation, the yaw is equal to the heading and the rotations about x and y can simply be left out.

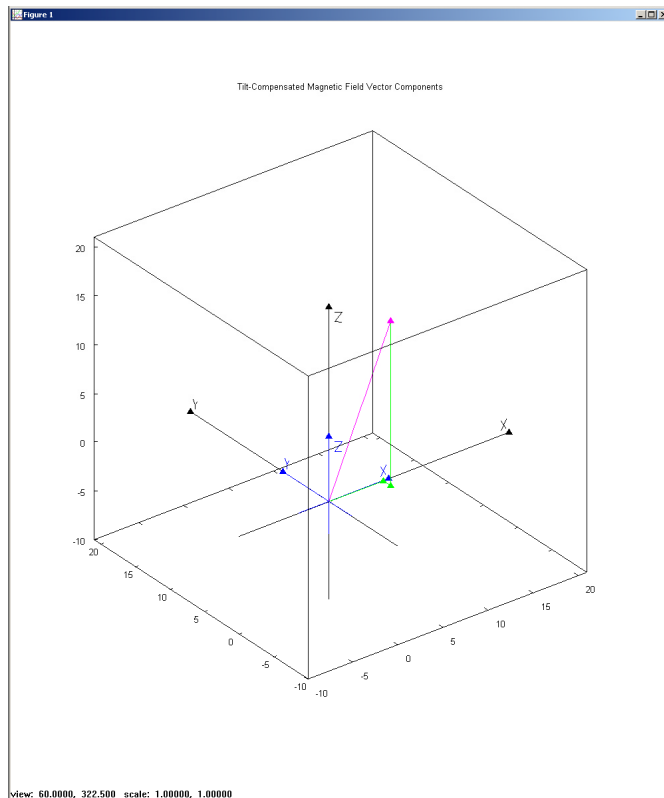
```
hold( "off" )
plotaxes( eye(3), 30, 0.5, 0, "k" )
hold( "on" )
plotaxes( rzp, 10, 0.5, 0, "b" )
plotvector( mfield_g, "r" )
title( "Tilt Compensated Position" )
kbhit();
```



To calculate the magnetic field components we would see on the axes of the tilt-compensated system, the trick is again to do a transformation to the ground-fixed system. This time only  $r_{zn}$  is needed, not the full  $r_n$ .

(I skip rotating back for display from now on, otherwise its the same game as always - rotating, split up, rotating back.)

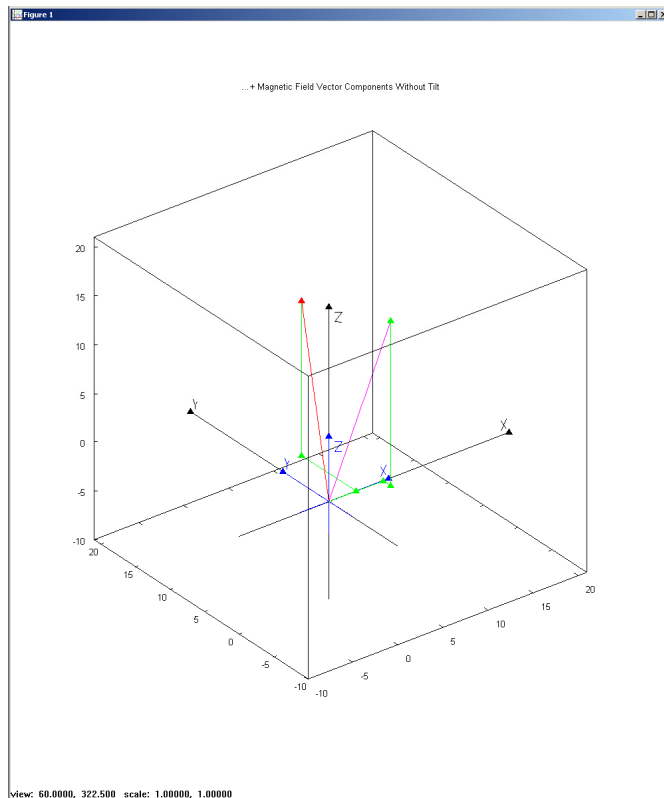
```
hold( "off" )
plotaxes( eye(3), 30, 0.5, 0, "k" )
hold( "on" )
plotaxes( rzp * rzn, 10, 0.5, 0, "b" ) %same as plotaxes( eye(3), 10, 0.5,
0, "b" )
plotvector( mfield_g * rzn, "m" )
magn_c_v = mfield_g * rzn; %equation (1)
magnxc_v = [ 0, 0, 0; magn_c_v(2, 1), 0, 0 ];
magnyc_v = [ 0, 0, 0; 0, magn_c_v(2, 2), 0 ];
magnzc_v = [ 0, 0, 0; 0, 0, magn_c_v(2, 3) ];
magnyc_v_offs = [ magnxc_v(2,:); magnxc_v(2,:) + magnyc_v(2,:) ];
magnzc_v_offs = [ magnxc_v(2,:) + magnyc_v(2,:); magnxc_v(2,:) +
magnyc_v(2,:) + magnzc_v(2,:) ];
plotvector( magnxc_v, "g" )
plotvector( magnyc_v_offs, "g" )
plotvector( magnzc_v_offs, "g" )
title( "Tilt-Compensated Magnetic Field Vector Components" )
kbhit();
```



Note that the variables got an additional 'c' signifying the tilt-compensation. The magnetic field vector `magn_c_v` seen in the tilt-compensated position is drawn in magenta. Keep in mind equation (1). It is needed later on.

In our model we already did the transformation from the vehicle-fixed to the ground-fixed system. For reference the magnetic field vector `magn_v` – the vector without tilt-compensation - is added again to this drawing:

```
%magn_v = mfield_g * rn %equation (2), already calculated above
plotvector( magn_v, "r" )
magnx_v_offs = [ magnx_v(2,:); magnx_v(2,:) + magny_v(2,:) ];
magnz_v_offs = [ magnx_v(2,:) + magny_v(2,:); magnx_v(2,:) + magny_v(2,:) +
magnz_v(2,:) ];
plotvector( magnx_v, "g" )
plotvector( magny_v_offs, "g" )
plotvector( magnz_v_offs, "g" )
title( "...+ Magnetic Field Vector Components Without Tilt" )
kbhit();
```



So, how do we get from the experienced magnetic field (the sensor outputs, red) to the tilt-compensated magnetic field (magenta)?

By eliminating the unknown `mfield_g` from equation (1) and (2) we get the relation between them:

```

    magnc_v = mfield_g * rzn          equation (1) from above
    magn_v   = mfield_g * rn          equation (2) from above with rn = rzn *
ryn * rxn
=> magn_v   = magnc_v * rzn * rxn
<=> magnc_v = magn_v * rxp * ryp    equation (3)

```

The last two equations may have been simply developed from the idea that it is possible to convert back and forth just by leaving out the rotation about `z`.

(Though, I think, it is not obvious that this is also mathematically correct. And in fact it wouldn't be that simple if we hadn't chosen the rotation sequence to be `xyz`.)

By expanding equation (3) we get the following to finally calculate the heading:

```

magnc_x = magn_v(2,1); %abbreviation for convenience
magnc_y = magn_v(2,2);
magnc_z = magn_v(2,3);
magnc_xc = magnc_x * cosd( pitch ) + magnc_y * sind( roll ) * sind( pitch ) +
magnc_z * cosd( roll ) * sind( pitch );
magnc_yc = magnc_y * cosd( roll ) - magnc_z * sind( roll );

```

The `z`-component of `magnc_v` naturally is not contributing to the heading. The heading can finally be determined from the arrangement in the `xy`-plane. To better see this look at the drawing from the top:

```

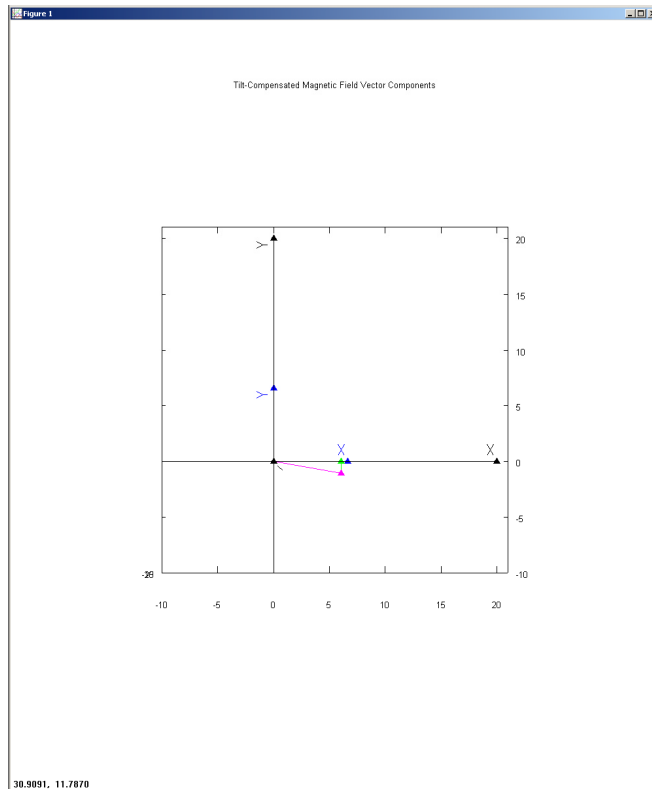
hold( "off" )
plotaxes( eye(3), 30, 0.5, 1, "k" )
hold( "on" )
plotaxes( eye(3), 10, 0.5, 1, "b" )
plotvector( mfield_g * rzn, "m" )

```

```

plotvector( magnxc_v, "g" )
plotvector( magnyc_v_offs, "g" )
plotvector( magnzc_v_offs, "g" )
handle_to_current_axis_object = gca();
set( handle_to_current_axis_object, "view", [0, 90] )
title( "Tilt-Compensated Magnetic Field Vector Components" )
kbhit();

```



From the figure we derive the equation:

```
heading = - atand( magnyc / magnxc )
```

As with roll atan2 has to be used for an expanded output range. To stay well in the range of  $-180 < \text{heading} \leq 180$ , change of sign is done by changing the sign of the first parameter:

```
heading = atan2d( - magnyc, magnxc )
```

We can now check if our equations work for every attitude as expected. The stepsize in the script is currently set to 15 degrees to save time. Reduce it if you like:

```
angle_check
```

I hope you are still with me ;-). At the end the whole stuff boils down to only 5 equations. In recapitulation here is an abstract with some additional implementation hints. The processing sequence how I think it should be implemented is:

```

3 axis acceleration measurement (accx, accy, accz)
3 axis magnetic field measurement (magnx, magny, magnz)
noise filter / damping
offset and gain error compensation
discard measurements with accz = 0
plausibility check, sqrt( accx^2 + accy^2 + accz^2 ) close to 1g
roll  =  atan2d( accy, accz )
pitch = - atand( accx / accz * cosd(roll) )

```

```

(limit operational range to +-80 degrees)
magnxc = magnx * cosd( pitch ) + magny * sind( roll ) * sind( pitch ) +
magnz * cosd( roll ) * sind( pitch )
magnyc = magny * cosd( roll ) - magnz * sind( roll )
heading = atan2d( - magnyc, magnxc )

```

The 6 measurements must be executed in fast succession, virtually at the same time. This makes up a snapshot of the current attitude and magnetic field. All measurements are belonging together.

The error propagation of the equations allows to do the noise filtering either directly on the sensor outputs or later on intermediate results or even on the final heading. The result is about the same.

The noise filtering is typically done on the sensor outputs. The reason is that the sensor outputs are usually sampled quite fast (faster than the change in attitude or at least tracking it very fast) while the display requires only a slow update rate. With the filter on the sensor outputs, the performance critical trigonometric calculations are done only with the display update rate.

The noise filter is primarily intended to cancel out the noise. This could be done by simple averaging. With damping I mean the more tricky part to achieve that the displayed heading nicely follows the compass/vehicle movement (like a liquid damped mechanical compass). Damping can be combined with the noise filter or implemented later, working on the final heading for example.

Offset and gain error compensation is done with the values from the compass calibration. In its simplest form (an offset and a factor for each axis) it makes up for the linear and time-invariant deviations of the hardware.

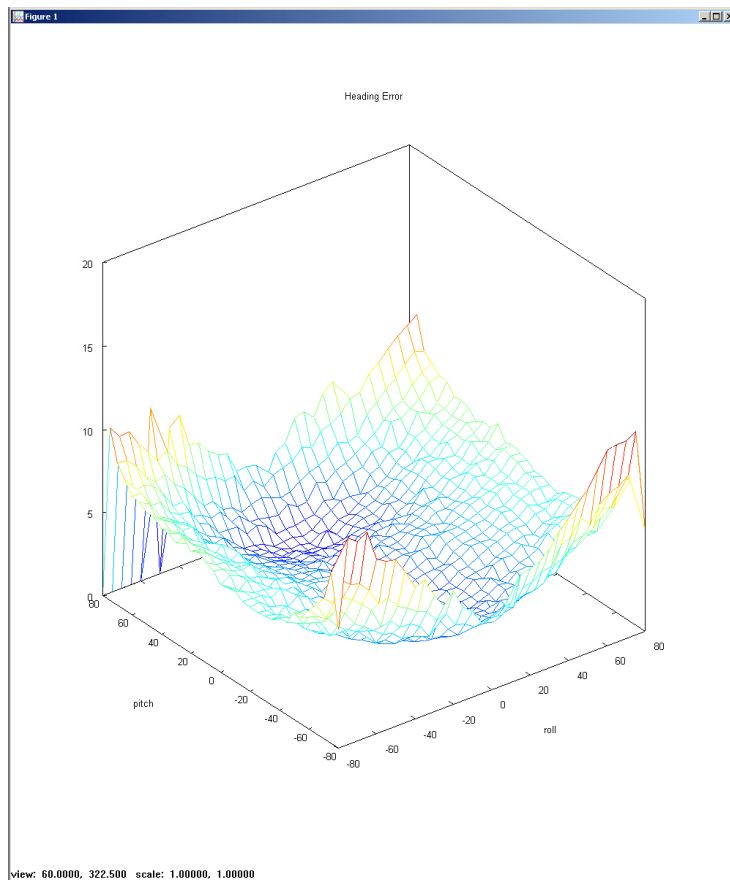
Applying offsets and scaling each axis separately provides also a simple way to calibrate for soft and hard iron effects of the compass environment. This method is sufficient in a lot of environments (environments producing the typical Lissajous-formed deviation in the xy-plane).

To avoid division by 0 measurements with  $\text{accz} = 0$  are simply discarded.  $\text{accz} = 0$  means roll and/or pitch =  $\pm 90$  degrees. This is above the limit we considered to be useful for vehicles/handhelds. pitch =  $\pm 90$  degrees leads to  $\text{accy} = \text{accz} = 0$  and thus roll can't be determined from sensor outputs. On one hand it's pretty useless anyway. In practice getting exactly 0 from the sensors happens quite seldom. So think of it as a measure to get the software bulletproof.

Acceleration of the vehicle itself will lead to a false tilt detection. It is assumed that the vehicle acceleration is small and/or from short duration. The plausibility check avoids false tilt detection. If the magnitude of the acceleration measured is not close to 1g, the current measurement is discarded. It is not entering the noise filter.

As discussed in part 3, noise sensitivity is very high near the poles of the trigonometric functions. Limiting the operation to about  $\pm 80$  degrees is acceptable for vehicles/handhelds. Otherwise, however, more effort is needed to get around the noise near the poles of the trigonometric functions.

All the stuff discussed for a possible implementation taking various error sources into account is simulated by the script implementation



The script calculates and displays the expected heading error over the full range of angles.

I hope this was not too scary. I tried to shed light on every corner of the tilt compensation, and to provide you with simulation models for your own experiments. Hopefully I got you to a point where you are now able to work successfully on your projects.

Good Luck :-)