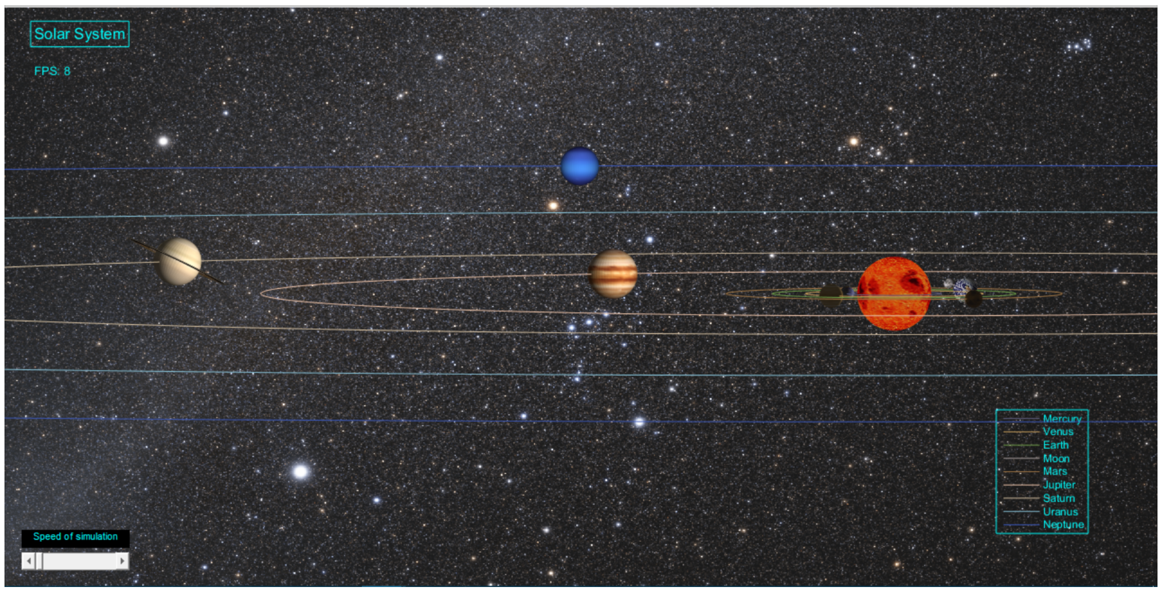


Solar System Simulation

Simulation of our solar system

Applications of MatLab/Octave

MatLab and *Octave* are very versatile programs. To explore the possibilities, we created a simulation of our solar system using *MatLab*. bla bla bla realism, bla bla bla to scale and some shit about not elipsies but circles

Studiengang:	Micro- and Medicaltechnology
Autoren:	Eric Lochmatter, Lukas Studer
Datum:	06. 01. 2016

Inhaltsverzeichnis

1 Introduction **1**

2 Drawing the Planets and Orbits **2**
2.1 Planets . . . 2
2.2 Orbits . . . 3
2.3 Key Values for the Modle . . . 4

3 Moving the Planets **5**
3.1 Earth's Spin . . . 5
3.2 Increased Speed . . . 5

4 UI Elements **6**
4.1 FPS Display . . . 6
4.2 Speed Slider . . . 6
4.3 Orbit Legend . . . 6

1 Introduction

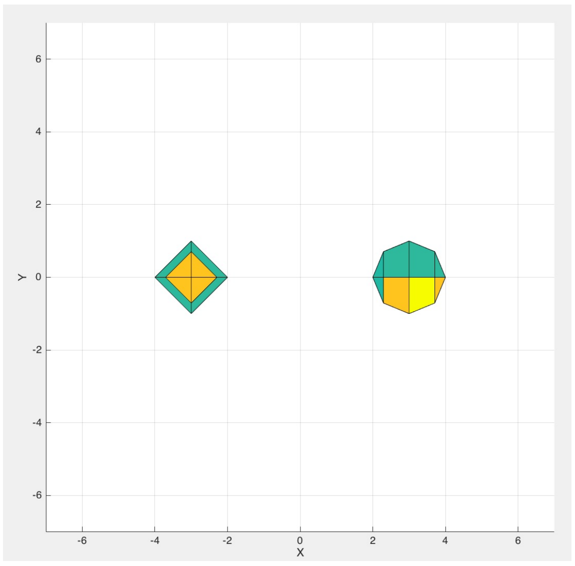
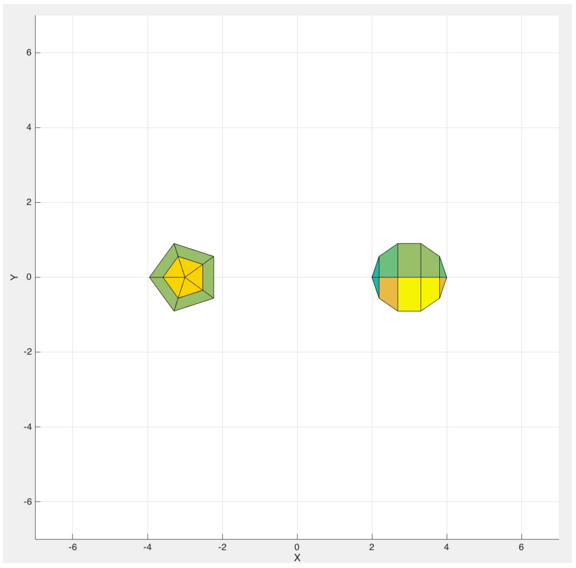
This project aims at creating a simple modle of our solar system using *MatLab.* While putting emphasis on the visuals, we kept the mathematical aspect simple.

Trajectories are not calculated and the orbits are perfect circles, due to the simple fact that the difference would be hardly noticeable. The sizes and distances are mostly to scale. We use a logarithmic scale for the sizes and a linear, though a very small scale, for the distances.

2 Drawing the Planets and Orbits

2.1 Planets

Drawing the planets is generally very simple: at first we create a sphere using the *sphere* function which gives us the coordinates *x*, *y* and *z* of each corner of the sphere while taking the resolution as the argument. The resolution defines how smooth the surface of the sphere is. For example: a resolution of 50 generates a spheres composed of 50 equally wide rings with 50 equally big surfaces. Following images shows how the spheres react to different resolutions:

Left: a sphere with resolution 4 (top and side view); Right: a sphere with resolution 5 (top and side view)

We then manipulate the position values to scale and move the sphere as needed. We do this by multiplying the position matrices with a factor and then adding a constant to move it to the correct position.

Next we load the textures from an image using various *MatLab* functions: first we read the image of a map of the planet using *imread*. Then we convert the data to doubles using *im2double*. Next we use *imresize* to adjust the image the size of the planet. We then only have to flip the data upside down to correctly display the texture. This is done using the *flipud* function. Further details for these function can be found in the help pages of each function.

Last but no least we create a surface object, or the 'planet', using the *surf* function by passing the position matrices and the textures. We can then use this object to move and rotated the planet.
Following *MatLab* code draws a simple sphere with a texture as described above:

```
[x,y,z] = sphere(resolution);
x = x*scale(1) + pos(1); y = y*scale(2) + pos(2); z = z*scale(3) + pos(3);

texture = flipud(imresize(im2double(imread(imgFile)),size(x)));
planet = surf(x, y, z, texture,'EdgeColor', 'none');
```

pos and *scale* are vectors containing the coordinates of the center of the planet and factors for stretching the planet in *x*, *y* and *z* directions. *imgFile* is a string containing the path to the textures of the planet and *resolution* is an integer for the resolution of the sphere as described above.

2.1.1 Rings

Some planets, in our case only saturn, have rings of water ice and rocky material circling them. To respect the thickness of such rings and for simplicity we created them using a torus and squished it to make it thiner. A torus is described as follows:

$$
\begin{aligned}
x(\theta,\varphi) &= (R + r * \cos(\theta) * \cos(\varphi) \\
y(\theta,\varphi) &= (R + r * \cos(\theta) * \sin(\varphi) \\
x(\theta,\varphi) &= r * \sin(\varphi)
\end{aligned}
$$

θ, φ are angles which make a full circle, so that their values start and end at the same point,
R is the distance from the center of the tube to the center of the torus,
r is the radius of the tube.
In *MatLab* this looks some what like this:

```
phi = linspace(0,2*pi,resolution)';
alpha = linspace(0,2*pi,resolution*10)';

tmp = [R + r*cos(phi), r*sin(phi)];
x = cos(alpha)*tmp(:,1)';
y = sin(alpha)*tmp(:,1)';
z = tmp(:,2)';
z = z(ones(1,length(alpha)),:);
```

The rest for drawing the ring is the same as for drawing the planets: we move and scale the ring using vectors containing the position of the center and factors for each direction and then create a surface object with the textures of the ring. Only the textures are slightly different compared to the planets. While the planets use a map, the ring has to be only a portion of the ring mirrored along the inner *'edge'* of the ring.

2.2 Orbits

We decided to use circular orbits for our modle, instead of simulating them and calculating the trajectory. Thus drawing the trajectory of each planet was made a lot easier. The function *getPlanetOrbit* draws a simple circle in a 3 dimensional room. The function has the following attributes:

- Center: stands for the middle of the circle
- Normal: stands for direction that the circle is tilted
- Radius: radius of the circle
- Color: color of the circle ...

```
theta = 0:0.01:(2*pi+0.01);
v = null(normal);
points = repmat(center',1,size(theta,2))+radius*(v(:,1)*cos(theta)+v(:,2)*sin(theta));
orbit = plot3(points(1,:),points(2,:),points(3,:),'Color',color);
```

2.3 Key Values for the Modle

As said before, we tryed to use realistic values and scales for the modle. Listed in the following two chapters are the values we ended up using. We gathered the data from: http://space-facts.com/

2.3.1 Distances

Planet	Distance [km]	Distance [AU]	Scale Factor
Mercury	57'909'227	0.39	0.39
Venus	108'209'475	0.73	0.73
Earth	149'598'262	1	1
Moon	384'400	0.0025	0.0025
Mars	227'943'824	1.38	1.38
Jupiter	778'340'821	5.20	5.20
Saturn	1'426'666'422	9.58	9.58
Uranus	2'870'658'186	19.22	19.22
Neptune	4'498'396'441	30.10	30.10

The distances are all the average distance from the sun.
Except for the moon, whose distance is the average distance from earth.

2.3.2 Sizes

Planet	Diameter [km]	Scale Factor
Sun	1'392'684	109.18
Mercury	4'879	0.38
Venus	12'104	0.95
Earth	12'756	1
Moon	3'475	0.27
Mars	6'805	0.53
Jupiter	142'984	11.21
Saturn	120'536	9.45
Uranus	51'118	4.01
Neptune	49'528	3.88

The diameters used are the equatorial diameters of the planets.
The scale factor used is log10 of the value in this table.

2.3.3 Speed

Planet	Earth days / year	Scale Factor
Mercury	87.97	4.1521
Venus	224.7	1.6255
Earth	365.26	1
Moon	27.3	13.3795
Mars	686.98	0.5317
Jupiter	4332.82	0.0843
Saturn	10755.7	0.034
Uranus	30687.15	0.0119
Neptune	60190.03	0.0061

The scale factor is used for the orbit speed.
Therefor less days mean faster orbit speed.

3 Moving the Planets

The decision to use circular orbits also influenced how we moved the planets in out modle. We were able to use the *rotate* function from *MatLab*. This function allows us to rotate the objects by a number of degrees around a vector at a given position. Due to the flexibility, we can use this function for all the movement in the modle. This includes the movement along the orbit, the spin of planets and keeping the rotation axis of planets aligned.

To animate the modle, we move all objects using the *rotate* function and then refresh the graph. We defined this to be one tick. This way of animating the modle may be very simple, but proved to have some unpleasant side effects as described below.

3.1 Earth's Spin

We defined the *'normal'* speed of the modle to be so that the earth rotates one degree around the sun per tick. This means that it takes earth 360 ticks to complete one year's worth of rotation. This on the other hand means that, rounding the number for this explanation, earth has to rotate once around itself per tick. This would result in, again rounding the numbers, 365 earth days per earth year which, as we all know, would be correct. Rotating earth 360 degrees around itself per tick results in no visible spin of earth. Using the real numbers would still result in only a very slow spin, due to the fact that it rotates a bit more than 360 degrees per tick

We *solved* this problem by giving earth an arbitrary speed of rotation around itself.

3.2 Increased Speed

Basically the same problem occurs when increasing the speed of the animation using the slider. As described in the chapter *UI Elements*, the slider only increases the angle of rotation. Setting the speed to higher values leads to visual effects similar to those of aliasing. The basic principle of aliasing is that the sample rate of the measuring device is to slow for what it is measuring, which leads to distorted results. In our case, we generate to few steps to create a smooth animation. This makes planets jump, rather than smoothly move, and rotate clockwise rather than counter clockwise. As a simple example, if the rotation angle is set to 90 degrees earth would jump three times before returning to its starting position. This is drastically different to the 360 steps earth makes at normal speed.

As already the one above, we weren't able to fix the problem and could only restrict the maximum speed, which works for the planets in the outer solar system but not so much for the planets closer to the sun.

4 UI Elements

The modle provides three UI elements for the user, besides the common controls from *MatLab* itself. A display for the number of frames rendered pre second, a slider to control the speed at which the modle runs and a legend for the different orbits of the planets and the moon.

4.1 FPS Display

While programming the modle we soon realized that the certain changes had a massive impact on the number of frames *MatLab* could render per second. The number of frames per second, also known as *FPS* directly dictates how smooth the modle runs. Through time analysis of the modle we found out that the resolution of the planets, sun, and moon had the most noticeable impact on the *FPS*. We then added the display to find the optimal resolution at which the graphics of the planets didn't suffer to much but still allowed for a smooth animation. We settled for a resolution of 50. This keeps the *FPS* at around 20 which is only slightly lower than the number images the human eye can capture per second. This makes the animation appear smooth.

With the help of the command tic() and toc() we measure the time need for one while loop or one tick. If we take the inverse of the measured time we get the frames rendered per second. We then round the value to 2 decimals places and finally convert the numerical value into a string which is then ploted to the graphical window. Following code does the trick:

```
fps = num2str(round((1/toc())*10)/10);
tic();
fps_text.String = ['FPS: ', fps];
```

4.2 Speed Slider

Because Uranus and Neptune are so far away from the sun, it takes them quite a while to complete an entire turn around the sun. That is why we added a slider on the bottom left which allows users to speed up the animation. Additionally the slider can also stop the animation by setting the speed to zero which allows for closer inspection of the planets. The slider basically controls the angle of rotation per tick. Faster speed results in a grater angle. Setting the speed to higher values leads to distortion of the animation. More on this can be found in the chapter *Moving the Planets*. Alternatives like controlling a delay for rendering the next frame have been tested, but didn't work out due to the *FPS* plummeting or the animation quickly reaching a maximum speed and therefore not having the desired effect.

4.3 Orbit Legend

The third UI element we add is a simple legend for the orbits: it associates a name to the orbit color used. This is especially useful when viewing the entire modle.