Dirk Eisenbiegler

Object Oriented Modeling and Simulation with the Physolator

—

Getting Started

This text is an extract from the German book

"Objektorientierte Modellierung und Simulation physikalischer Systeme mit dem Physolator"

1 Getting Started

This chapter explains from scratch, how to build physical simulations using Physolator. The chapter starts with a simple physical system. We will learn how to build a physical model using physical variables and formulae, how to translate these variables and formulae into Java code, how to load the program code to Physolator, how to start the simulation and how to add a graphics component to the physical system.

1.1 Physical Model

The physical system used in this chapter consists of three physical bodies: earth, moon and a satellite moving on an orbit somewhere around moon and earth.[1] In our physical model, all three physical bodies – earth, moon and the satellite – shall be point masses. The mass of the satellite shall be negligibly small compared with the mass of earth and moon.

1 This example originates from a publication from E. Fehlberg und R.R. Newton [1] [2].

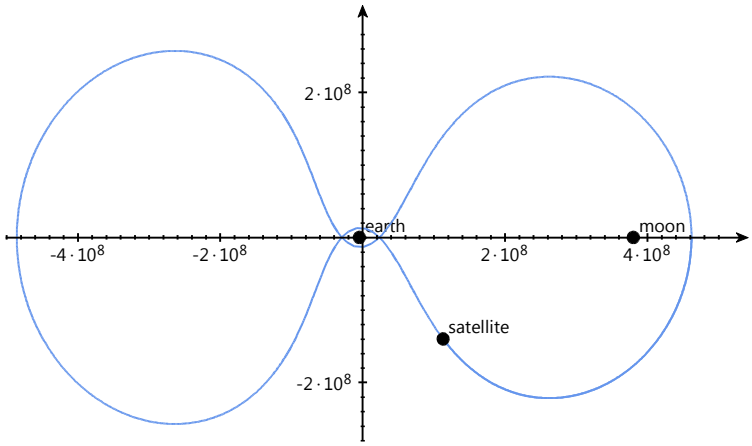

fig. 1

To simplify matters, it is assumed that the distance between earth and moon is to be constant.

In every day life one likes to say, that the moon revolves around the earth. This statement is not absolutely correct. More precisely, one would have to say, that earth and moon have a common center of mass and both earth and moon revolve around this center of mass. The center of mass is located on the connection line between moon and earth. Since the mass of earth is far bigger than the mass of moon, the center of mass is located very close to earth.

For our physical model we will use a special coordinate system. In this coordinate system, the center of mass shall be the origin and both earth and moon shall be positioned on the x-axis (see figure 1). The entire coordinate system rotates counterclockwise around its origin. The angular velocity of the rotation is constant. Figure 1 shows the satellites orbit. In this figure, the x-axis is oriented to the right and the y-axis is oriented upwards. The origin is the center of mass. Note, that the earth is located a little bit to the left of the origin. We will work with a rotating reference system. From the spectators perspective, the x-axis always points to the right, the y-axis always points upwards and moon and earth are on fixed positions. With respect to the coordinate system, earth and moon do not move during the simulation. The satellite is the only body that moves. As a result of our simulation we will determine the orbit of the satellite.

At the beginning of the simulation, the satellite is put to a “suitable” position on the right hand side of the moon and it is given a “suitable” initial speed in the direction of the negative y-axis. The orbit of the satellite depends on the initial position and the initial speed. By using a well chosen initial position and a well chosen initial speed according to figure, one can achieve, that the satellite flies on a closed curve. With the initial values, that we will use, the satellite flies on an orbit according to figure 1 and one circulation takes approximately 27 days. Approximately 27 days after its start, the satellite reaches exactly its

initial position. At this point in time its speed equals exactly the initial speed. This is why the next circulation as well as all further circulations will be exactly on the same orbit.

During its flight, the satellite is affected by several forces. These forces define the movement of the satellite. There three different kinds of forces applying to the satellite:

- gravitational forces of earth and moon
- a centrifugal force
- a Coriolis force

The gravitation forces pull the satellite towards moon and earth. In this example we are using a rotating reference system. This is why centrifugal forces and Coriolis forces act on all masses inside the physical system. The centrifugal force pushes the satellite away from the center of rotation, i.e. the origin of the coordinate system. The Coriolis force applies in a direction that is vertical to the movement of the body.

Figure 2 shows the initial state of the physical system.

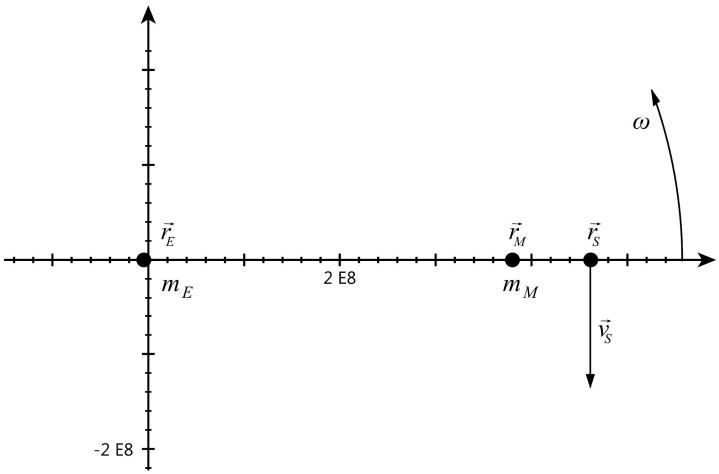

fig. 2: Initial State

1.2 Determining the Constants

In a first step we will assemble all constants used in this physical system. Constants are all physical quantities, whose values do not change during simulation.

The entire coordinate system rotates counterclockwise with a fixed angular velocity ω. Inside this coordinate system earth and moon have both fixed positions $\vec{r}_E$ and $\vec{r}_M$,

respectively. The masses of earth and moon shall be referred to as m_E and m_M, respectively. The constant of gravitation is named G.

$$\omega = 2{,}6616994 \cdot 10^{-6}\ s^{-1}$$

$$\vec{r}_E = \begin{pmatrix} -4.67588761 \cdot 10^{6}\ m \\ 0 \end{pmatrix}$$

$$m_E = 5{,}974 \cdot 10^{24}\ kg$$

$$\vec{r}_M = \begin{pmatrix} 3{,}8010277025 \cdot 10^{8} \\ 0 \end{pmatrix} m$$

$$m_M = 7{,}349 \cdot 10^{22}\ kg$$

$$G = 6{,}67384 \cdot 10^{-11}\ \frac{m^3}{kg\ s^2}$$

fig. 3: Constants

1.3 Determining State Variables

In a next step we will assemble all variables, that define the actual state of the physical system.

In our example, there are two state variables: the actual position of the satellite $\vec{r}_S$ and its actual velocity $\vec{v}_S$. The following figure shows a snapshot during the flight of the satellite.

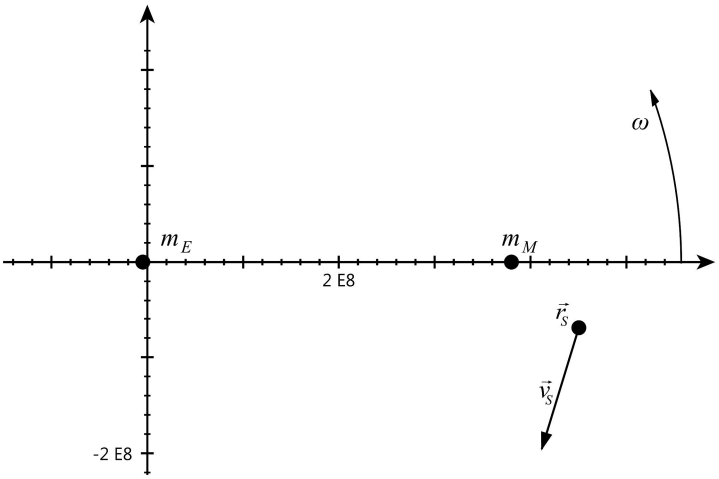

fig. 4: State of the Physical System

1.4 Initial Values for State Variables

Now we have to define the initial state of the physical system. The initial state is the state at the beginning of the simulation, i.e. the state at time zero. The state of the physical system is unambiguously described by its state variables. Therefore, we define initial values for all state variables.

The following figure shows the initial values for state variables $\vec{r}_S$ and $\vec{v}_S$. At time zero the satellite is located on the x-axis on the right hand side of the moon and the speed is directed downwards (see figure 2).

$$\vec{r}_S(0) = \begin{pmatrix} 4{,}617343894 \cdot 10^8 \\ 0 \end{pmatrix} m$$

$$\vec{v}_S(0) = \begin{pmatrix} 0 \\ -1074{,}98157 \end{pmatrix} \frac{m}{s}$$

fig. 5: Initial Values

1.5 Derivations for State Variables

To unambiguously define the behavior of the physical system, we have to define the derivations with respect to time for all state variables.

In our example, there are two state variables $\vec{r}_S$ and $\vec{v}_S$. The derivation of the satellite's postion $\vec{r}_S$ is its velocity $\vec{v}_S$. The first derivative of $\vec{v}_S$ is the acceleration of the satellite. The satellites acceleration shall be referred to as $\vec{a}_S$.

$$\dot{\vec{r}}_S = \vec{v}_S$$

$$\dot{\vec{v}}_S = \vec{a}_S$$

The derivation of $\vec{r}_S$ is well known, because it is also a state variable. We will now determine the acceleration $\vec{a}_S$. The acceleration depends on the actual state, i.e. it depends on the actual position and the actual velocity of the satellite. We will first determine the forces acting on the satellite and then derive the acceleration from the total force.

Let m_S be the mass of the satellite. The gravitation forces from earth and moon acting on the satellite shall be referred to as $\vec{F}_E$ and $\vec{F}_M$, respectively. Let $\vec{F}_Z$ be the centrifugal force and $\vec{F}_C$ the Coriolis force acting on the satellite. The following relationships apply:

$$\vec{F}_E = \frac{G\, m_S\, m_E\, (\vec{r}_E - \vec{r}_S)}{|\vec{r}_E - \vec{r}_S|^3}$$

$$\vec{F}_M = \frac{G\, m_S\, m_M\, (\vec{r}_M - \vec{r}_S)}{|\vec{r}_M - \vec{r}_S|^3}$$

$$\vec{F}_Z = m_S \omega^2 \vec{r}$$

$$\vec{F}_C = 2\, m_S\, \omega \begin{pmatrix} v_{S,y} \\ -v_{S,x} \end{pmatrix}$$

Explanation: In the equation with the Coriolis force $\begin{pmatrix} v_{S,y} \\ -v_{S,x} \end{pmatrix}$ is the normal vector to the satellites velocity $\vec{v} = \begin{pmatrix} v_{S,x} \\ v_{S,y} \end{pmatrix}$. The Coriolis force acts in the direction of the normal vector.

Adding all the forces that apply to the satellite results in a total force. The total force shall be referred to as $\vec{F}_S$.

$$\vec{F}_S = \vec{F}_E + \vec{F}_M + \vec{F}_Z + \vec{F}_C$$

In a next step the acceleration of the satellite can be derived from the total force. The total acceleration of the satellite can be represented by a sum of single accelerations – each linked with its corresponding force.

$$\vec{a}_S = \frac{\vec{F}_S}{m_S} = \frac{\vec{F}_E + \vec{F}_M + \vec{F}_Z + \vec{F}_C}{m_S} = \frac{\vec{F}_E}{m_S} + \frac{\vec{F}_M}{m_S} + \frac{\vec{F}_Z}{m_S} + \frac{\vec{F}_C}{m_S} = \vec{a}_E + \vec{a}_M + \vec{a}_Z + \vec{a}_C$$

The equations for the single accelerations:

$$\vec{a}_E = \frac{G\, m_E\, (\vec{r}_E - \vec{r}_S)}{|\vec{r}_E - \vec{r}_S|^3}$$

$$\vec{a}_M = \frac{G\, m_M\, (\vec{r}_M - \vec{r}_S)}{|\vec{r}_M - \vec{r}_S|^3}$$

$$\vec{a}_Z = \omega^2\, \vec{r}$$

$$\vec{a}_C = 2\, \omega \begin{pmatrix} v_{S,y} \\ -v_{S,x} \end{pmatrix}$$

The total acceleration of the satellite:

$$\vec{a}_S = \frac{G\, m_E\, (\vec{r}_E - \vec{r}_S)}{|\vec{r}_E - \vec{r}_S|^3} + \frac{G\, m_M\, (\vec{r}_M - \vec{r}_S)}{|\vec{r}_M - \vec{r}_S|^3} + \omega^2\, \vec{r} + 2\, \omega \begin{pmatrix} v_{S,y} \\ -v_{S,x} \end{pmatrix}$$

As you can see, the single accelerations as well as the total acceleration do not depend on the mass of the satellite m_S. This is why the mass of the satellite m_S need no longer be taken into consideration.

Figure 13 shows the satellite at a certain point in time during the simulation. In this figure, single accelerations are represented by blue arrows and the total acceleration is represented by a red arrow.

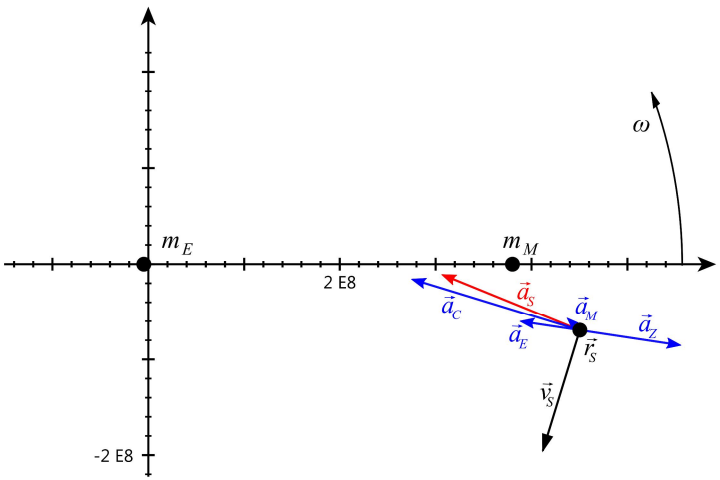

fig. 6: Accelerations Acting on the Satellite

1.6 Converting Vectors to Scalar Variables

So far, two dimensional vectors have been used for representing positions, velocities and accelerations. In a first approach, the Java program code shall be implemented with scalar variables only, i.e. without vectors. In fact, Physolator does support vectors. However, when working with vectors, one has to deal with object oriented concepts. Object oriented concepts will be introduced later on in chapter 6. In chapter 6, we will implement our physical system one more time using vectors and object oriented concepts.

The implementation, that will be presented in this chapter, does not use any object oriented concepts. This is why, in a next step we will describe our physical model in a scalar manner. The equations in the following figures are an equivalent representation of our physical system. The only difference: vectors have been replaced by scalar variables.

$$\omega = 2{,}6616994 \cdot 10^{-6}\ s^{-1}$$

$$r_{E,x} = -4.67588761 \cdot 10^{6}\ m$$

$$m_E = 5{,}974 \cdot 10^{24}\ kg$$

$$r_{M,x} = 3{,}8010277025 \cdot 10^{8}\ m$$

$$m_M = 7{,}349 \cdot 10^{22}\ kg$$

$$G = 6{,}67384 \cdot 10^{-11}\ \frac{m^3}{kg\ s^2}$$

fig. 7: Constants, Scalar Equations

$$r_{S,x}(0) = 4{,}617343894 \cdot 10^{8}\ m$$

$$r_{S,y}(0) = 0\ m$$

$$v_{S,x}(0) = 0\ \frac{m}{s}$$

$$v_{S,y}(0) = -1074{,}98157\ \frac{m}{s}$$

fig. 8: Initial Values, Scalar Equations

$\dot{r}_{S,x} = v_{S,x}$
$\dot{r}_{S,x} = v_{S,x}$
$\dot{v}_{S,x} = a_{S,x}$
$\dot{r}_{S,y} = a_{S,y}$

fig. 9: Derivations, Scalar Equations

$a_{E,x} = \frac{G\, m_E \left(r_{E,x} - r_{S,x}\right)}{\left(\left(r_{E,x} - r_{S,x}\right)^2 + r_{S,y}{}^2\right)^{\frac{3}{2}}}$	$a_{E,y} = -\frac{G\, m_E\, r_{S,y}}{\left(\left(r_{E,x} - r_{S,x}\right)^2 + r_{S,y}{}^2\right)^{\frac{3}{2}}}$
$a_{M,x} = \frac{G\, m_M \left(r_{M,x} - r_{S,x}\right)}{\left(\left(r_{M,x} - r_{S,x}\right)^2 + r_{S,y}{}^2\right)^{\frac{3}{2}}}$	$a_{E,y} = -\frac{G\, m_E\, r_{S,y}}{\left(\left(r_{E,x} - r_{S,x}\right)^2 + r_{S,y}{}^2\right)^{\frac{3}{2}}}$
$a_{Z,x} = \omega^2 r_{S,x}$	$a_{Z,y} = \omega^2 r_{S,y}$
$a_{C,x} = 2 \cdot \omega \cdot v_{S,y}$	$a_{C,y} = -2 \cdot \omega \cdot v_{S,x}$
$a_x = a_{E,x} + a_{M,x} + a_{Z,x} + a_{C,x}$	$a_{S,y} = a_{E,y} + a_{M,y} + a_{Z,y} + a_{C,y}$

fig. 10: Accelerations, Scalar Equations

1.7 Constants, State Variables and Dependent Variables

In a physical system, there are three kinds of physical variables:

- constants
- state variables
- dependent variables

Figure 11 lists all variables from our example. Each variable is assigned to one of the three groups: constants, state variables and dependent variables.

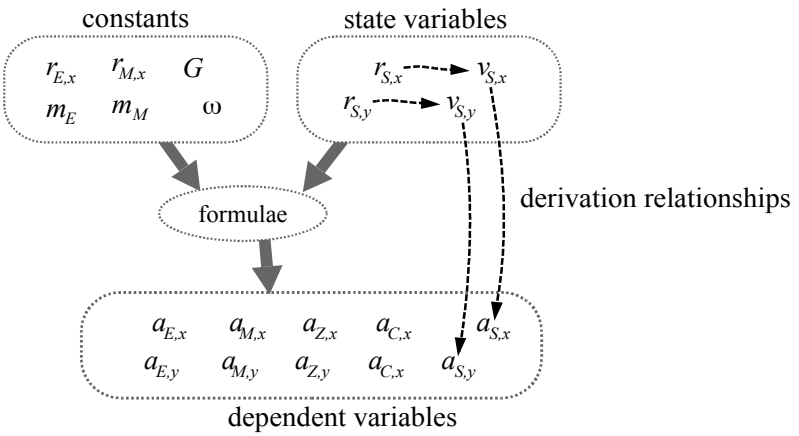

fig. 11

In the context of physical simulation, a constant is a specific physical variable, whose value is assigned at the beginning of the simulation and whose value does not change during simulation.

States variable define the state of the physical system in an unambiguous way. State variables do change their values during the simulation. The initial values of the state variables at time zero have do be well-defined.

Formulae are used to compute further physical values from the constants and the state variables. These further values shall be referred to as dependent variables. For every dependent variable, there must always be a formulae, with which you can directly compute its value – with the constants and state variables as inputs.

Both constants and state variables have well-defined initial values, that are assigned at the beginning of the simulation. During the simulation, state variables usually do change there values from simulation step to simulation step whereas the values of the constants remain unchanged. Unlike constants and state variables, dependent variables do not have initial values. At any point in time, their values can be determined from the actual values of the constants and state variables. This is also true for time zero.

For every state variable, there must be a derivation, that is attached to the state variable. The derivation can be any kind of variable: a dependent variable, a constant or another state variable. In figure 11 the relationships between the state variables ant their corresponding derivation variables are represented by dashed arrows.

1.8 Installing the Physolator

The variables and formulae from the previous section describe our physical system in an appropriate manner. In a next step we will convert the physical model into a Java class and afterwords run the physical system inside Physolator.

Before we continue, we have to make sure, that Physolator is installed on your computer system. This section describes, how to install Physolator and how to setup your development environment. Have a look at the following website for more detailed installation and configuration instructions:

www.physolator.de

On this website there is also the installation package for download. The installation is easy. After starting the installation file, the setup procedure is executed automatically. If not explicitly configured differently, Physolator is installed in the following folder:

C:/Programs/Physolator-PE-1.0.0

Physolator is a Java program. Therefore, make sure Java 8 is installed on your computer. More precisely you need the Java Development Kit (JDK) in a Standard Edition (SE), version 8. The installation package for Java 8 can be found on the following website:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

For programming your Java code any integrated development (IDE) environment can be used. It is recommended to use Eclipse. No matter which IDE you prefer to use: Create an ordinary Java project inside the IDE and link the Physolator library to your class path. The Physolator library is part of the installation package. Inside the installation folder you can find it at the following location:

plugins/Physolator_1.0.0.jar

It depends on the IDE, how a library is linked to the class path. If your are using Eclipse, go to your project, then:

right mouse click – Properties – Libraries – Add external JARS

A file dialog will pop up. Go to Physolator's installation folder, there go to the subdirectory *plugins* and select *Physolator_1.0.0.jar*.

1.9 Implementing the Physical System as a Java Class

The following program code is a Java class that implements our physical system. As you can see, Java variables of type double represent scalar physical variables. For every physical variable in our system, there is a corresponding variable declaration in the Java code. The Java type *double* stands for high precision floating point numbers. Initial values are assigned to all constants and state variables. Method *f* computes the dependent variables values – thereby using constants and state variables as inputs.

```
import static java.lang.Math.*;
import de.physolator.usr.*;

public class MoonEarthSatellite extends PhysicalSystem {

    public double G = 6.67384E-11;
    public double omega = 2.6616994E-6;
    public double rEx = -4675887.610598851;
    public double rMx = 3.801027702506129E8;
    public double mE = 5.974E24;
    public double mM = 7.349E22;

    @V(derivative = "vSx")
    public double rSx = 4.617343894E8;
    @V(derivative = "vSy")
    public double rSy = 0;
    @V(derivative = "aSx")
    public double vSx = 0;
    @V(derivative = "aSy")
    public double vSy = -1074.98157;

    public double aEx, aEy, aMx, aMy, aZx, aZy, aCx, aCy, aSx, aSy;

    public void f(double t, double h) {
        aEx = (G * mE * (rEx - rSx)) / pow(pow(rEx - rSx, 2) + pow(rSy, 2), 1.5);
        aEy = -(G * mE * rSy) / pow(pow(rEx - rSx, 2) + pow(rSy, 2), 1.5);
        aMx = (G * mM * (rMx - rSx)) / pow(pow(rMx - rSx, 2) + pow(rSy, 2), 1.5);
        aMy = -(G * mM * rSy) / pow(pow(rMx - rSx, 2) + pow(rSy, 2), 1.5);
        aZx = pow(omega, 2) * rSx;
        aZy = pow(omega, 2) * rSy;
        aCx = 2 * omega * vSy;
        aCy = -2 * omega * vSx;
        aSx = aEx + aMx + aZx + aCx;
        aSy = aEy + aMy + aZy + aCy;
    }

    public static void main(String args[]) {
        start();
    }
}
```

Let us have a closer look at the variable declarations. Right above the declarations of the state variables $r_{S,x}$, $r_{S,y}$, $v_{S,x}$ and $v_{S,y}$, there are annotations with the name *@V*. Annotations are Java language constructs (see [3]). The Physolator framework will later on be used to run this program code. The annotations in our code are used to provide some extra information to the Physolator framework. *@V*-Annotations are used for describing

physical variables. In the program code above, these annotations are located right before every state variable declaration. Each of these annotations has a parameter named *derivative*. This annotation parameter defines the derivative variable, that is assigned to the state variable. The annotation parameter *derivative* is of type *String*. Its value is the name of the derivative variable. In our example, the annotations define, that $r_{S,x}$, $r_{S,y}$, $v_{S,x}$ and $v_{S,y}$ are state variables and that $v_{S,x}$, $v_{S,y}$, $a_{S,x}$ and $a_{S,y}$, respectively, are their derivatives.

From a Java perspective, our program code is a class declaration. The class is named *MoonEarthSatellite*. Physolator provides means for loading and running physical systems. From the Physolator's perspective, physical systems are Java classes that inherit from class *PhysicalSystem*. This is why in the headline of the *MoonEarthSatellite* class declaration it says: *extends PhysicalSystem*.

Our program code could already ready be loaded into the Physolator. However, before loading and starting the code, we will add two useful pieces of code to improve the functionality. First, we want to define, that the simulation runs in fast motion mode. Without any extra code, the simulation would run in real time with one second of simulation time corresponding to one second in real world. In our example, it takes approximately 27 days for the satellite to circle around one time. One would have to wait for 27 days to see a complete circulation. By adding the following code to our class declaration, the simulation is accelerated by a factor of 86400. With this setting, one day of simulation time corresponds to one second in real world. The simulation will run much faster and one circulation will only take 27 seconds of real world time.

```
public void initSimulationParameters(SimulationParameters s) {
        s.fastMotionFactor = 86400;
}
```

Besides *fastMotionFactor*, there are more settings for controlling the execution of the simulation. Chapter Fehler: Referenz nicht gefunden explains in more detail, how Physolator executes simulations and how to control the simulation behavior using various configuration parameters.

Physolator has an integrated function plotter. The function plotter produces function graphs for a selected number of physical variables. In our example, the history of the satellite speed shall be displayed on the screen. Therefore, the following piece of code is added. With this piece of code we define, that the function plotter shall paint a diagram for the variables $v_{S,x}$ and $v_{S,x}$. The width of the x-scale shall be $3 \cdot 10^6 s$ and the y-scale shall range from -8000 to 8000.

```
public void initRecorderPlotDescriptors(RecorderPlotDescriptors r) {
        r.add("v_S_x,v_S_y", 3e6, -8000, 8000);
}
```

There are various further features, that you can use when implementing a physical system. See chapter 5 and 6 for details.

At the end of the program code, there is a *main* method declaration. In Java, main methods have a special meaning. To start a program it takes a class with a main method. Starting a Java class means starting its main method. The main method makes our program code launchable. Since our program codes does have a main method, it can directly be executed as an ordinary program. You can run it from your IDE or you can run it as a stand alone program.

After starting class *MoonEarthSatellite,* its main method is executed. The main method runs method *start* and the *start* method starts the Physolator. The Physolator framework appears on your screen. Physolator detects the name of the class it has been started from and loads this class. During load time, Physolator detects the physical variables inside the physical system, their names and values, and it detects that a function plot is to be produced for *vSx* and *vSy.*

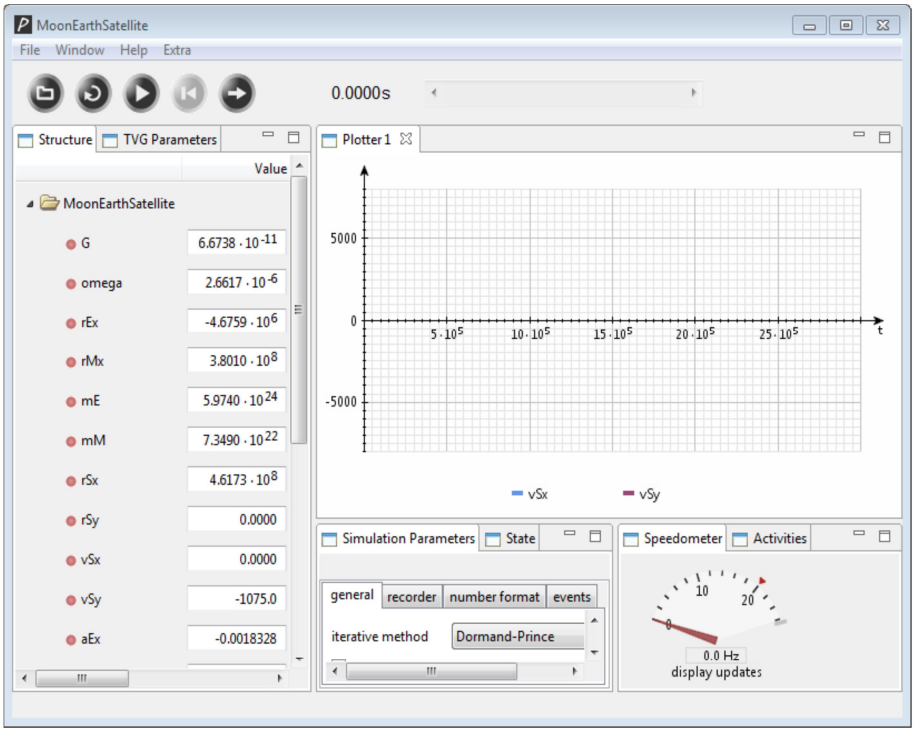

fig. 12: Physolator after Starting Class MoonEarthSatellite

1.10 Simulation

After the start, the physical system is in its initial state. All physical variables have their initial values.

Press the start button to start the simulation. Figure 13 shows the physical simulation during a simulation run. As long as the simulation is running, the start button turns into a stop button. Press the button again to stop the simulation.

The actual simulation time is displayed at the top of the Physolator window. During the simulation run, the simulation time advances and you can see, how the physical variables change their values with every simulation step. The simulation is executed in real time. By default, the step width is 50ms. Every 50ms, Physolator computes a new state of the physical system and visually displays the new state on the screen.

The history of the simulation run is automatically recorded. This record is used by the function plotter. The function plotter draws function graphs of selected physical variables. At the upper right corner of the window, there is a scroll bar. This scroll bar is used for moving forth and back inside the history of the simulation. At the left and right ends of the scroll bars, there are arrows. Use them to move a single step forth or back.

Button brings you back to the initial state.

Pressing button starts the simulation and one simulation step after another is executed unless the user stops the simulation. Button executes one simulation step only. After pressing this button, one single simulation step is executed and the simulation automatically stops after this simulation step.

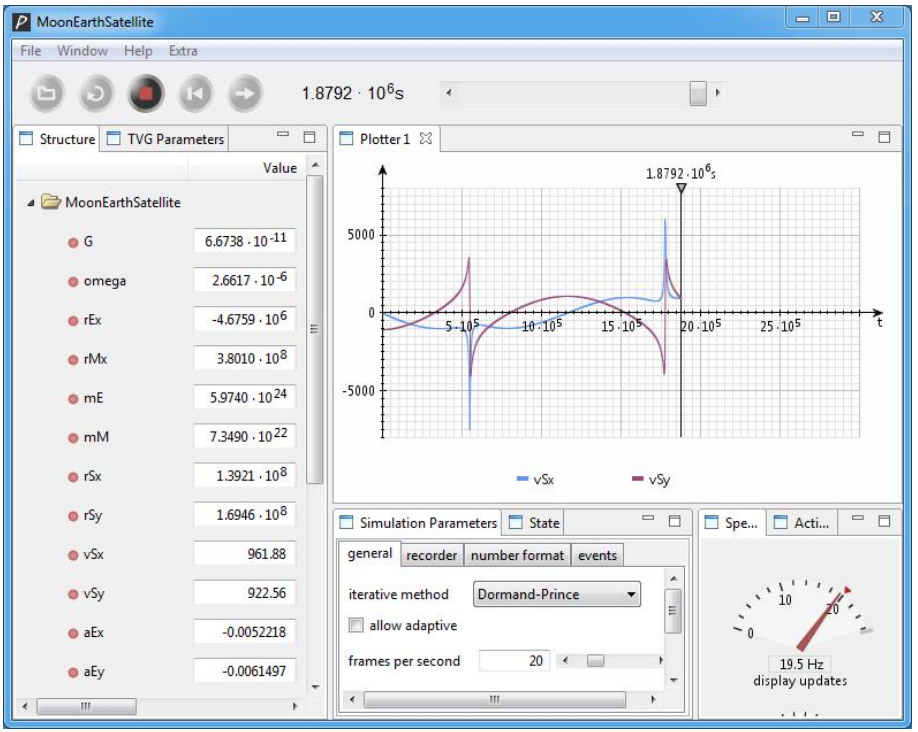

fig. 13: Physolator during Simulation

If you have loaded a physical system into Physolator and if you decide to modify the program code inside your IDE, you can reload the program code by pressing the button. Your physical system inside the Physolator is updated and all changes made in your program code are adopted.

1.11 Visualization

In a next step, we will add a graphics component to our physical system.

```
import de.physolator.usr.tvg.TVG;

public class MoonEarthSatelliteTVG extends TVG {

    private MoonEarthSatellite mes;

    public MoonEarthSatelliteTVG(MoonEarthSatellite mes) {
        this.mes = mes;
        geometry.setUserArea(-5e8, 5e8, -3e8, 3e8);
        geometry.setRim(30, 20, 20, 30);
        scalesStyle.visible = true;
    }

    public void drawCelestialBody(double x, double y, String label) {
        double d = 3 * geometry.scaleX;
        drawCircle(x, y, d);
        drawText(x + d, y + d, label);
    }

    public void paint() {
        style.useUCS = true;
        drawCelestialBody(mes.rSx, mes.rSy, "S");
        drawCelestialBody(mes.rEx, 0, "E");
        drawCelestialBody(mes.rMx, 0, "M");
    }
}
```

This program code represents a class named *MoonEarthSatelliteTVG*. It is the job of this class to visually display the state of the physical system *MoonEarthSatellite* on the screen. The program code contains an object attribute named *mes*. This object attribute stores a reference to the physical system *MoonEarthSatellite*. Object attribute *mes* is initialized in the constructor.

Besides that, the constructor also makes certain basic settings. The constructor defines, that the drawing area ranges from $-5 \cdot 10^8$ to $5 \cdot 10^8$ in x-direction and from $-3 \cdot 10^8$ to $3 \cdot 10^8$ in y-direction. Furthermore, the constructor defines, that there is a rim surrounding the painting area, with the width of the rim being 20 pixels at the top and at the bottom and 30 pixels at the left and at the right. The drawing commands are located inside the *paint* method. Each of the celestial bodies moon, earth and satellite shall be represented by a small circle and a label. Method *drawCelestialBody* draws a small circle at a given location *(x,y)* and writes a label right next to the circle. The radius of the circle is 3 pixels. The lower left corner of the label is located three pixels to the right of the center of the circle *(x,y)* and three pixels upwards. The coordinate system used for drawing uses meters as unit. Variable *d* represents a distance of three pixels with respect to this coordinate system. Inside the *paint* method *drawCelestialBody* is invoked three times to draw satellite, earth and moon.

As a next step, we add the following piece of code to our class *MoonEarthSatellite*. This piece of code links our graphics component *MoonEarthSatelliteTVG* to our physical system *MoonEarthSatellite*. Whenever the physical system is loaded, the graphics component will automatically be loaded, too. The graphics component will have a reference to the physical system, so it can access the data inside the physical system.

```
public void initGraphicsComponents(GraphicsComponents g, Structure s, Recorder r) {
    g.addTVG(new MoonEarthSatelliteTVG(this));
}
```

To adopt these changes in our code, we have to press the reload button  . Figure 14 shows, that after pressing the reload button, a graphics component appears on the screen. In the graphics component, you can see moon, earth and satellite at their initial positions.

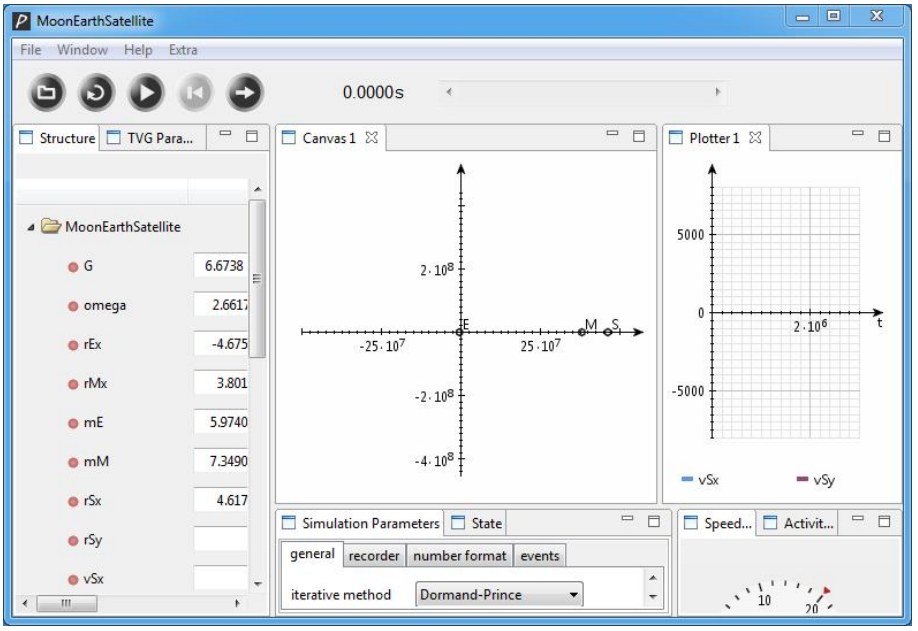

fig. 14

As soon as you start the simulation, the satellite starts moving around on its orbit between moon and earth.

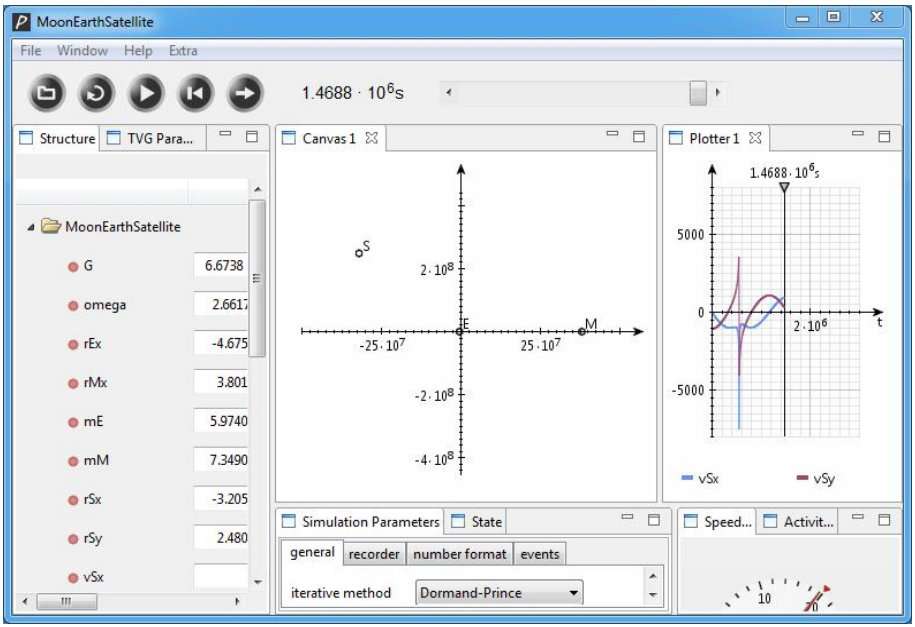

fig. 15

For programming our graphics component, a special graphics programming interface named TVG has been used. TVG stands for “technical vector graphics”. This programming interface was specially developed for the Physolator. It provides a set of graphics commands technical graphics in a simple way. The graphics painted by TVG can be directed to the screen and at the same time these graphics can also be saved as vector graphics. Vector graphics are well suited for print publications with a high resolution. On the screen, TVG graphics are interactive. As soon as you enter the painting area with your mouse, buttons show up (see figure 16). The camera button takes a snapshot from the graphics and stores the snapshot picture to a file. You can choose to store the file in PNG format, which is a bitmap format, or you may store it in a SVG format, which is a vector graphics format.

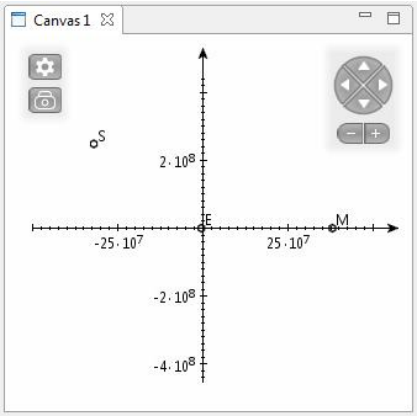

fig. 16

TVG graphics are navigable. The buttons at the upper right corner are used for this purpose. Inside a TVG graphics you can move up, down, left and right and you can zoom in and out. As an alternative to the buttons on the screen, you can also use your keyboard. Cursor keys are used for moving forth and back and key + and key – are used for zooming in and out, respectively.

Button  starts the settings dialogue. After pressing this button, a dialogue named “TVG Parameters” shows up. With this dialog you can change all the relevant setting of your graphics component. Shall there be a rim? How wide shall the rim be? Shall scales be automatically drawn on the screen? How about the density of the ticks on the scales? Shall there be grid lines? etc.

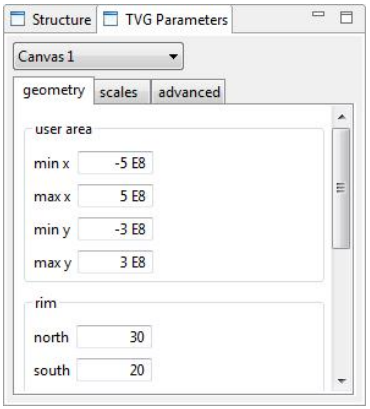

fig. 17

1.12 What is next?

In this chapter we have learned how to implement a simple physical system using the Java programming language and how to run it inside Physolator. Building on this basic knowledge, the following chapters go into the details and explain advanced features. You do not necessarily have to read the chapters in the given order. There are different kinds of tasks in the domain of physical simulation. Maybe you do not want to read all the chapters. Feel free to select the chapters with the information you need for your tasks.

It is the intention of this book, that you can read it with different kinds of background knowledge. For most chapters, simple programming skills are good enough. You do not necessarily need object oriented programming skills to get started.

Chapter 3 explains the fundamental structure of a physical system in Physolator and the different alternatives you have when building a physical system. Chapters 4 and 5 explain in detail, how Physolator runs a physical simulation, how to control the simulation using various simulation parameters and how to implement “physical events”. Object oriented concepts are not needed for chapters 3, 4 and 5.

Chapter 6 explains how object oriented concepts can be used to build physical systems in a modular and reusable manner. How to build physical components and how to reuse these physical components in different physical systems? This chapter explains how object oriented concepts such as classes, instances and inheritance can be used to implement physical systems in a modular way. It shows that physical systems and their components are good examples for explaining object oriented concepts. If you are not yet familiar with object oriented concepts, then maybe this chapter can help you getting started. Some of the following chapters will depend on the concepts presented in this chapter.

Chapters 7, 8 and 8 deal with graphics components. Very often, two dimensional drawings are used in physics books. Two dimensional drawings usually are easier to understand than three dimensional drawings. In Physolator, TVG components are used for drawing two dimensional graphics. Chapters 7 is dedicated to two dimensional graphics.

Physolator also supports three dimensional graphics. In Physolator tree dimensional graphics are called TVG3D components. TVG3D components are based on OpenGL. It takes a fundamental understanding of graphics programming to implement such graphics components. Three dimensional graphics are introduced in chapter 8.

The appearance of graphics components can be controlled by parameters. During run time, the user can modify these parameters using the settings button  and adjust the appearance to user's needs. There are already different kinds of parameters in classes TVG and TVG3D. Chapter 9 explains how you can add extra parameters to your graphics components, that are tailored to the specific needs of your application.

References

[1] Fehlberg, E.: Numerical Integration of Differential Equations by Power Expansions, Illustrated by Physical Examples, NASA TN D-2356, 1964

[2] Newton, R.R.: Periodic Orbits of a Planetoid Passing Close to Two Gravitating Masses, Smithsonian Contributions to Astrophysics Vol. 3, Number 7, 1959

[3] Eckel B.: Thinking in Java, Prentice Hall, 2006