Modeling and Simulation of Physical Systems for Hobbyists

Essential Tools for Developing, Testing and Debugging Systems Interacting with the Real World

Manuel Aiple

29 December 2018

Contents

1. Introduction

This is the accompanying script to the talk on the 35th Chaos Communication Congress (35C3) in Leipzig on 29 December 2018. It is meant to lower the threshold for beginners without extensive mathematical background to start modeling and simulation of physical systems. Furthermore, the goal is to provide methods to hobbyists, who do not have the resources of expensive simulation software, for integrating simulation models into their projects, e.g., building mobile robots, drones, etc. These methods (based on the Euler method) can easily be implemented on commonly available tools, like in a Python script or in C source code, so they can also be integrated in other programs, e.g., embedded software deployed on a micro-controller.

In the context of this script modeling refers to creating a mathematical description of a physical system. A physical system will most of the time be a mechanical or electromechanical system, such as an electric motor driving a mobile platform or a robotic arm through some gears, lever mechanisms, etc. But in any case it should be a system that can be built in hardware, not a system from some fictional reality. Simulation is then used to run the model in the time-domain, i.e., starting from an initial system state, the simulation software will calculate the system state at the following moments in time. The system state refers to the collection of all relevant variables at a given instant.

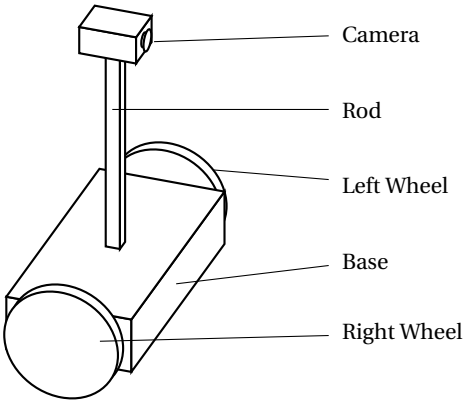

Figure 1: Two-wheeled self-balancing robot main components

2. Motivation

Often in projects aiming at building mobile robots or other machines, it is desirable to split up the work load between different persons, e.g., one person building the hardware, the other writing the control software, in order to map the required skills or simply to split up the work by personal preference. However, in this approach, the person writing the software will need to wait for the hardware to be ready to effectively test the software, as it interacts very closely with the hardware and needs to react appropriately to sensor inputs, to command actuators, etc. Thus, the supposed advantage of splitting up the work is eliminated. Simulation can help to solve this conflict by providing a placeholder that will take the same inputs as the component that it simulates and react in the same manner as its physical counterpart.

For example, a reasonably easy but challenging mobile robot could be a self-balancing platform with two wheels, driven by two motors, and a camera mounted at the tip of a rod to provide a higher view point (cf. Fig. 1). This is an inverted pendulum and therefore inherently unstable. It needs to permanently take small compensating actions to not fall over to one side or the other (similar to a human standing on two legs). Therefore, it will require a controller implementing a feedback loop with the inclination measured by an accelerometer as input and command to the motors as output. In order to start the development of the controller and everything related to the control (e.g., a user interface for steering the robot), we can implement a simulator that provides the inputs to the controller such as they would be provided by the sensors based on the physical effects applying to the robot and taking the command inputs of the controller (see appendix A for an example implementation in Python). Then, once the robot is built in hardware, the simulator is removed and the controller interfaces directly with the robot sensors and actuators. But also once the hardware is ready the simulator can still be useful as a virtual test bench. Test scenarios that are rare, difficult to produce or risk to damage the hardware can be played through in the simulator first.

In the following, we are going to see how to get started with modeling a physical system and to implement a simple simulation engine. This cannot be an extensive introduction, but is meant to provide a starting point to show that writing a basic simulator is simple and requires only few resources. So it can be integrated even in small projects and embedded computers or micro controllers with little computing power.

3. Modeling

The first step to build a simulator for a physical system is to represent the physical effects acting on the system in mathematical equations that can be solved by a computer. For this, it is useful to think about the level of detail that is required. It will be a trade-

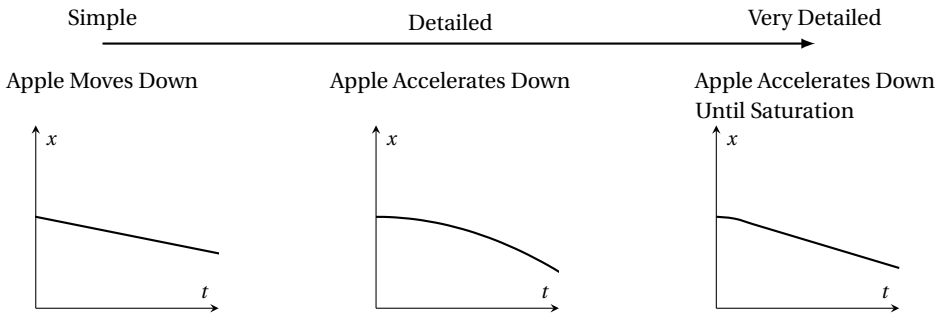

Figure 2: Different levels of modeling an appel falling down from simple to very detailed. The plots show the position x over time t.

off between the computing power available, restricting the model complexity, and the fidelity of the model, i.e., how well the model can predict the behavior of the real system, requiring a higher model complexity. For example, we can model how an apple falls down at at least three levels of complexity from simple to very detailed (cf. Fig. 2). At the simple level, the apple moves down, which shows in the plot as a line going downwards with time from the start position (left plot). This can easily be observed in daily life without measurements. At the more detailed level, the apple accelerates down, which is shown by the parabolic shape of the position plot (center plot). At this level of detail, more precise measurements are required to make the effect apparent, for example with a camera and a stroboscope. If the apple falls from a higher point, one can notice that it does not accelerate forever, but the aerodynamic drag will counter the acceleration until a final velocity is reached. The curve in the plot transitions from a parabola shape in the beginning of the motion into a straight line over time (right plot).

The modeling process can usually be performed in an iterative manner. First, a simulation is done with a very simple model taking into account only the most dominant physical effects. In the example with the apple falling, this could be the downward motion. The simulation output is then compared to the physical system behavior. If the error between simulation output and physical system behavior is "small enough", then this model can be used. Otherwise, the less dominant physical effects also need to be taken into account. In the example, this would mean to change from constant speed motion to accelerated motion. This is repeated until the simulation output is sufficiently close to the real world.

The vague formulation of when to stop the refinement process is chosen on purpose here. Indeed, it strongly depends on the question that needs to be answered by the simulation. For example, if we want to catch the apple, we probably track its current position closely and anticipate its position at the next instant from its current speed, without taking into account acceleration, thus using a very simple model. If we want to know how long it takes for the apple to fall from a 3 m high tree, then we should take into account the acceleration phase and choose the more detailed model. And a skydiver jumping from a plane at 3000 m might want to use the very detailed model taking into account the aerodynamic drag to know how long she has until she needs to open the parachute.

4. Simulation

4.1. Differentiation, Integration, and the Euler Method

Differentation and integration play an important role in physics as many physical measures are related to each other through differentiation or integration. For example, the velocity $\boldsymbol{v}$ of an object can be obtained from the position $\boldsymbol{x}$ through differentiation:

$$\boldsymbol{v}(t) = \frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t}\,, \tag{1}$$

where $\frac{\mathrm{d}}{\mathrm{d}t}$ stands for the differentiation operation over the time variable t. Likewise, the acceleration $\boldsymbol{a}$ is the derivative of the velocity:

$$\boldsymbol{a}(t) = \frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t}\,. \tag{2}$$

Inversely, the velocity can be obtained from the acceleration and the position from the velocity through integration:

$$\boldsymbol{v}(t) = \int_0^t \boldsymbol{a}(\tau)\,d\tau\,, \tag{3}$$

and

$$\boldsymbol{x}(t) = \int_0^t \boldsymbol{v}(\tau)\,d\tau\,, \tag{4}$$

where $\int_0^t \ldots d\tau$ signifies to integrate from the start instant 0 to the end instant t, using τ as helper variable.

Integration does in fact nothing else than summing up an infinite number of values of the function at points infinitely close to each other, multiplied by the infinitely small width $d\tau$ of the interval between the points. As this is, by its nature, not easily computable by a computer if the symbolic form is not known, there are many methods for calculating a numeric approximation of the integral of a function. In this script, we are going to use the Euler method, which is the easiest method.

In fact, when inspecting the definition of the derivative

$$\boldsymbol{v}(t) = \lim_{h\to 0} \frac{\boldsymbol{x}(t+h) - \boldsymbol{x}(t)}{h}\,, \tag{5}$$

one notices that transforming this equation through multiplication by h and addition of $\boldsymbol{x}(t)$ gives a form of integration calculation that allows to calculate the value of one measure (here the position $\boldsymbol{x}$) at an instant $t+h$ solely from a known value of the measure and its derivative (here the velocity $\boldsymbol{v}$) at an instant t:

$$\lim_{h\to 0} \boldsymbol{x}(t+h) = \boldsymbol{x}(t) + \lim_{h\to 0} \boldsymbol{v}(t)\,h\,. \tag{6}$$

The Euler method consists of using a finite value for h instead of $\lim_{h\to 0}$. This makes the calculations very easy, now consisting only of one multiplication and one addition. Typical robotic systems operate with discrete time controllers, which execute periodically at fixed time intervals T_s. The controller performs calculations only at the time steps

$$t = k\,T_s\,, \tag{7}$$

with k being an integer number increasing by one at every calculation. Thus, equation 6 can be written as

$$\boldsymbol{x}(k+1) = \boldsymbol{x}(k) + \boldsymbol{v}(k)\,T_s \tag{8}$$

for obtaining the next value of $\boldsymbol{x}$ from the known values of $\boldsymbol{x}$ and $\boldsymbol{v}$.

In the following, we are only going to use integration with this equation and not differentiation. For this, we are going to transform all equations describing physical effects in such a way that they allow us to calculate the highest order derivative (the acceleration $\boldsymbol{a}$ for most equations from mechanics), and then integrate to obtain the lower order derivatives (the velocity $\boldsymbol{v}$ and the position $\boldsymbol{x}$). This way, we calculate the values of all measures from one step to the next, enabling to extrapolate into the future from a known initial state, which is exactly what we want for a simulation.

4.2. Building Blocks

For building the simulation model, one needs to know which physical effects apply to the system, which ones are relevant, and which ones can be neglected. As it might be overwhelming to open a general physics book and study all existing physical effects, table 1 shows a short summary of physical effects and their equations, which are generally useful for robotics and electromechanical systems.

4.3. Programming the simulation engine

The equations of the previous section can be used as building blocks to obtain the equations for a system. For example, to model an electric motor, the law of motion is combined with the motor equation and viscous damping for the mechanical part:

$$\mathrm{I}\frac{\mathrm{d}\omega}{\mathrm{d}t} = K_t\, i - b\,\omega\,. \tag{9}$$

And the electrical part is obtained as a combination of Ohm's law, inductance and generator equation:

$$V = R\, i + L\frac{\mathrm{d}i}{\mathrm{d}t} + K_v\,\omega\,. \tag{10}$$

Equations 9 and 10 are then transformed to isolate the highest order derivative:

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = \mathrm{I}^{-1}\,(K_t\, i - b\,\omega)\,, \tag{11}$$

$$\frac{\mathrm{d}i}{\mathrm{d}t} = \frac{1}{L}\,(V - R\, i - K_v\,\omega)\,. \tag{12}$$

The lower order derivatives are obtained from the highest order derivatives by integration:

$$\begin{aligned}\boldsymbol{\omega}(t) &= \int_0^t \frac{\mathrm{d}\boldsymbol{\omega}}{\mathrm{d}t}(\tau)\,d\tau \\ &= \boldsymbol{\omega}(t-d\tau) + \frac{\mathrm{d}\boldsymbol{\omega}}{\mathrm{d}t}(t-d\tau)\,d\tau\,,\end{aligned} \tag{13}$$

$$\begin{aligned}i(t) &= \int_0^t \frac{\mathrm{d}i}{\mathrm{d}t}(\tau)\,d\tau \\ &= i(t-d\tau) + \frac{\mathrm{d}i}{\mathrm{d}t}(t-d\tau)\,d\tau\,.\end{aligned} \tag{14}$$

Using the Euler method, we obtain the following pseudo-code to implement the simulation in software (simplified to scalar variables):

```
1:  procedure RunSimulation
2:      InitSimulation
3:      while t < T_end do
4:          SimulationStep
5:      end while
6:  end procedure
7:  procedure InitSimulation
8:      T_end := 10e−3   ▷ Simulation end time
9:      T_s := 0         ▷ Sampling period
10:     k := 0           ▷ Sample counter
11:     t := 0           ▷ Simulation time
12:     ω := 0           ▷ Rotational speed
13:     ω_dot := 0       ▷ Derivative of rot. speed
14:     i := 0           ▷ Current
15:     i_dot := 0       ▷ Derivative of current
16:     I := 5.3e−6      ▷ Moment of inertia
17:     b := 1e−6        ▷ Viscous damping factor
18:     V := 24          ▷ Voltage
19:     R := 0.2         ▷ Resistance
20:     L := 8.4e−5      ▷ Inductance
21:     K_t := 0.023     ▷ Torque constant
22:     K_v := 0.023     ▷ Voltage constant
23: end procedure
24: procedure SimulationStep
25:     k := k + 1
26:     t := k ∗ T_s
27:     ω := ω + ω_dot ∗ T_s
28:     i := i + i_dot ∗ T_s
29:     ω_dot := 1/I ∗ (K_t ∗ i − b ∗ ω)
```

Table 1: Summary of relevant physical effects for modeling electromechanics

Effect	**Equation**	**Legend**	**Comment**
Second law of motion (translations)	$\boldsymbol{F} = M\,\boldsymbol{a}$	$\boldsymbol{F}$: Sum of forces M : Mass of object	Starting point of modeling moving object (translations)
Second law of motion (rotations)	$\boldsymbol{T} = \mathrm{I}\,\frac{\mathrm{d}\boldsymbol{\omega}}{\mathrm{d}t}$	$\boldsymbol{T}$: Sum of torques I : Moment of inertia $\boldsymbol{\omega}$: Rotational speed	Starting point of modeling moving object (rotations)
Weight	$\boldsymbol{F} = M\,\boldsymbol{g}$	$\boldsymbol{F}$: Weight M : Mass of object $\boldsymbol{g}$: Gravitational acceleration	Direction is “downwards”
Spring force	$\boldsymbol{F} = -\kappa\,(\boldsymbol{x} - \boldsymbol{x_0})$	$\boldsymbol{F}$: Spring force κ : Spring stiffness $\boldsymbol{x}$: Deflected position $\boldsymbol{x_0}$: Equilibrium position	Can be used together with viscous damping to model contacts
Viscous damping	$\boldsymbol{F} = -b\,\boldsymbol{v}$	$\boldsymbol{F}$: Damping force b : Damping factor $\boldsymbol{v}$: Velocity	Simplified friction model or for generic kinetic energy losses
Ohm's law	$V = R\,i$	V : Voltage R : Resistance i : Current	Modeling of resistors, conductors, etc.
Inductance	$V = L\,\frac{\mathrm{d}i}{\mathrm{d}t}$	V : Voltage L : Inductance i : Current	Modeling of coils of motors, antennas, etc.
Capacitor	$i = C\,\frac{\mathrm{d}V}{\mathrm{d}t}$	i : Current L : Capacitance V : Voltage	Modeling of capacitors, antennas, etc.
Motor	$\boldsymbol{T} = K_t\,i$	$\boldsymbol{T}$: Torque K_t : Motor constant i : Current	Torque generated by the coils inside a motor
Generator	$V = K_v\,\omega$	V : Voltage K_v : Voltage constant $\boldsymbol{\omega}$: Rotation speed	Voltage generated by the rotating coils inside a motor, when rotating

30: $i_{dot} := 1/L * (V - R * i - K_v * \omega)$
31: **end procedure**

5. Summary & Tips

Before implementing a simulation of a physical system, one should first clarify what level of detail is required. This will determine which physical effects should be taken into account, and which ones can be neglected. This involves defining the system boundaries and the interfaces of the simulation to external inputs and outputs. It is also useful to define how the simulation can be verified for correctness (ideally by comparing it to the real system, however, often this will be a plausibility check). It might sometimes be required to implement different models for the same system to focus either on one aspect or another while not increasing the model complexity too much.

The sampling period should be around hundred times shorter than the system time constant. For mechanical systems, often a sampling period of 1 ms will be sufficient. For electromechanical systems like motors 10 µs or less is probably needed. The smaller the sampling period, the more accurate and robust the simulation, but the longer the computation time.

Block diagrams like the one shown in Fig. 3 can help to keep an overview of how the physical measures relate to each other. Especially when several equations have cross-relations.

Specialized tools like SciPy, OpenModelica/OMEdit, Scilab/XCos, can help to model more complex models or to have a reference. They provide better differential equation solving (e.g., with BDF, Runge-Kutta, and other methods), work more efficiently through variable time-step, and feature nice data logging and visualization tools.

6. Background & Further Reading

Wikipedia keywords:

- Scientific modeling
- Ordinary differential equation
- Numerical methods for ordinary differential equations
  - Euler Method
  - Runge-Kutta
  - Backward differentiation formula (BDF)
- Discrete time and continuous time
- State-space representation
- Inverted pendulum
- Equations of motion

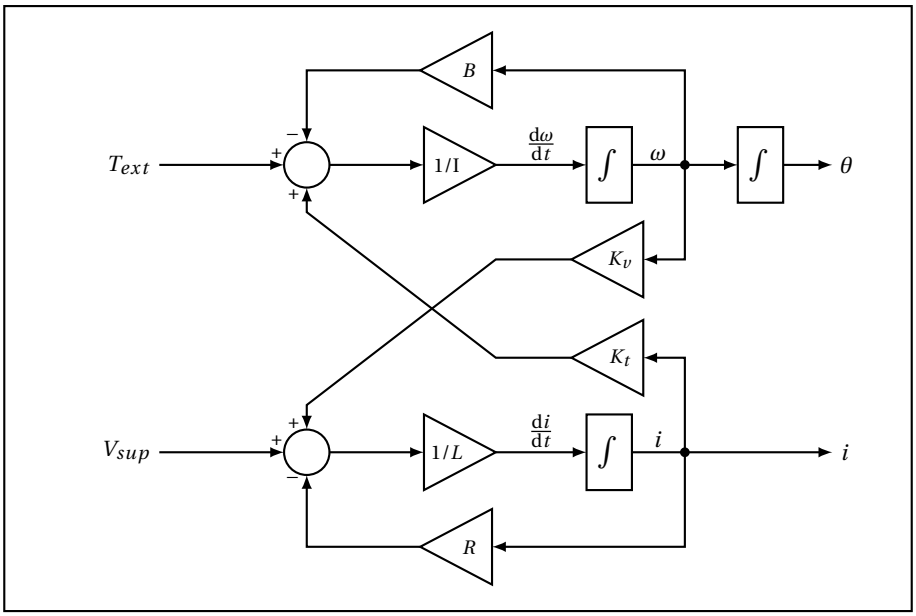

Figure 3: Example block diagram of an electric motor model

A. Self-balancing Robot Example Python Code

```
#! /usr/bin/env python
# Self balancing robot simulation with visualization
#
# Copyright (c) 2018 Manuel Aiple
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.

import Tkinter as tk # for visualization
from threading import Thread # to let sim and visualization run "in parallel"
from time import clock, sleep # for wall time tracking and fake "real time"
from math import sin, cos, pi, copysign # for trigonometric relations

class RobotSim (Thread):
    """
    Simulation thread running in fake "real time", calculating the model
    """
    def __init__(self):
        """
        Constructor
        """
        Thread.__init__(self)
        self.reset()

    def reset(self):
        """
        Initialize all simulation variables, can be called to restart simulation.
        """
        # sampling period (in s)
        self.Ts = 1e-4
        # linear position pid controller P gain (in N/m)
        self.Kpx = 0.7
        # linear position pid controller I gain (in N/m/s)
        self.Kix = 0
        # linear position pid controller D gain (in N*s/m)
        self.Kdx = 0.9
        # linear position error (in m)
        self.xerr = 0
        # linear position integration of error (in m*s)
        self.xierr = 0
        # linear position derivative of error (in m/s)
        self.xderr = 0
        # linear position last error (to calculate xderr, in m)
        self.xerrlast = 0
```

```
        # rotational position pid controller P gain (in N/rad)
        self.Kpt = 3
        # rotational position pid controller I gain (in N/rad/s)
        self.Kit = 0
        # rotational postion pid controller D gain (in N*s/rad)
        self.Kdt = 1.2
        # rotational position error (in rad)
        self.terr = 0
        # rotational position integration of error (in rad*s)
        self.tierr = 0
        # rotational position derivative of error (in rad/s)
        self.tderr = 0
        # rotational position last error (to calculate tderr, in rad)
        self.terrlast = 0
        # gravitational constant (in m/s^2)
        self.g = 9.81
        # mass of the rod tip (in kg)
        self.mtip = 0.5
        # mass of the base (in kg)
        self.mbase = 2
        # length of the rod (in m)
        self.l = 1
        # sample index
        self.k = 0
        # linear position set point (in m)
        self.xset = 0
        # linear position set point interpolation (in m)
        self.xseti = 0
        # maximum linear position change (in m)
        self.deltaxmax = 2 * self.Ts
        # linear position (in m)
        self.x = 0
        # linear velocity (in m/s)
        self.v = 0
        # linear acceleration (in m/s^2)
        self.a = 0
        # rotational position (in rad)
        self.theta = 0
        # rotational position set point (in rad)
        self.thetaset = 0
        # rotational velocity (in rad/s)
        self.omega = 0
        # rotational acceleration (in rad/s^2)
        self.psi = 0
        # external torque (in N*m)
        self.taue = 0
        # wheel radius (in m)
        self.r = 0.3
        # floor stiffness for contact model (in N/m)
        self.Kfloor = 10000
        # floor damping for contact model (in N*s/m)
        self.Bfloor = 100
        # wheel rotation angle (in rad)
        self.alpha = self.x / (self.r)
        # generate sinusoidal linear position set point
        self.periodic = False
        # sinusoidal set point amplitude (in m)
        self.periodicAmplitude = 1.5
        # sinusoidal set point frequency (in Hz)
        self.periodicFrequency = 0.1
        # time scaling factor (< 1 faster than real time)
        self.timeScale = 1
```

```
122         # stop simulation after next iteration
123         self.stopSim = False
124         # simulation start time in wall time (in s)
125         self.t0 = clock()
126         # reset simulation at next iteration
127         self.doReset = False
128         # activate controller
129         self.doControl = False
130 
131     def run(self):
132         """
133         Run the simulation model
134         """
135         print("Starting sim thread\n")
136         # run simulation until stop requested by gui
137         while self.stopSim == False:
138             # reset simulation if requested by gui
139             if self.doReset == True:
140                 self.reset()
141 
142             # calculate simulation time and do fake "real time"
143             self.k += 1
144             self.t = self.k * self.Ts
145             while (clock()   self.t0) < (self.t*self.timeScale):
146                 sleep(self.Ts/1000.)
147 
148             # rotational component
149             self.theta += self.omega * self.Ts
150             self.omega += self.psi * self.Ts
151             self.taug = self.g * sin(self.theta)
152             self.taua = self.a * cos(self.theta)
153             # simple collision model when the tip falls on the floor
154             self.ytip = self.r + self.l * cos(self.theta)   0.04
155             self.ytipdot =   self.l * sin(self.theta) * self.omega
156             if self.ytip < 0:
157                 if self.theta > 0:
158                     self.tauc = self.ytip * self.Kfloor + self.ytipdot * self.Bfloor
159                 else:
160                     self.tauc =   (self.ytip * self.Kfloor + self.ytipdot * self.Bfloor)
161                 self.Ff =   self.Bfloor * self.v
162             else:
163                 self.tauc = 0
164                 self.Ff = 0
165             # second law of motion for rotation
166             self.psi = 1/self.l * (self.taug + self.taua + self.taue + self.tauc)
167             # reset external torque to obtain impulse effect
168             self.taue = 0
169             # linear component
170             self.x += self.v * self.Ts
171             self.v += self.a * self.Ts
172             # sinusoidal set point generator
173             if self.periodic == True:
174                 self.xset = self.periodicAmplitude * sin(2 * pi * self.t * self.periodicFrequency)
175             # interpolator for linear set point
176             deltax = 5 * (self.xset   self.xseti) * self.Ts
177             if abs(deltax) < self.deltaxmax:
178                 self.xseti += deltax
179             else:
180                 self.xseti += copysign(self.deltaxmax, deltax)
181             # linear position pid controller
182             self.xerr = self.x   self.xseti
183             self.xierr += self.xerr * self.Ts
```

```
184             self.xderr = (self.xerr   self.xerrlast) / self.Ts
185             self.xerrlast = self.xerr
186             # rotational position pid controller
187             self.thetaset = 0
188             self.terr =  self.theta   self.thetaset
189             self.tierr += self.terr * self.Ts
190             self.tderr = (self.terr   self.terrlast) / self.Ts
191             self.terrlast = self.terr
192             # forces from controller (actual force = set force simplification)
193             self.Fx =  (self.Kpx * self.xerr + self.Kix * self.xierr + self.Kdx * self.xderr) /
    self.Ts
194             self.Fm =   (self.Kpt * self.terr + self.Kit * self.tierr + self.Kdt * self.tderr) /
    self.Ts
195             # forces from rotational component
196             self.Fg = self.mtip * self.l * self.psi * cos(self.theta)
197             self.Fc =   self.mtip * self.l * self.omega ** 2
198             # second law of motion for translation
199             if self.doControl == True:
200                 self.a = 1/(self.mbase + self.mtip) * (self.Fm + self.Fg + self.Ff + self.Fc + self
    .Fx)
201             else:
202                 self.a = 1/(self.mbase + self.mtip) * (self.Fg + self.Fc + self.Ff)
203             # calculate wheel rotation angle
204             self.alpha = self.x / self.r
205
206 class RobotGui (Thread):
207     """
208     Gui thread doing the visualization
209     """
210     def __init__(self, sim):
211         """
212         Constructor
213         """
214         Thread.__init__(self)
215         self.sim = sim
216         self.canvasWidth = 1920
217         self.canvasHeight = 1080
218         self.screenX0 = self.canvasWidth/2
219         self.screenY0 = self.canvasHeight/2
220         self.screenScaleX = 400
221         self.screenScaleY =  400
222         self.baseRad = sim.r * self.screenScaleX
223         self.tipRad = 15
224         self.setPointRad = 10
225         self.wheeldotRad = 5
226         self.disturbanceCounter = 0
227         self.disturbancePointRad = 12
228         self.disturbanceOffset = 0
229         self.cursorX = 0
230
231     def screenCoords(self, x, y):
232         """
233         Translate world coordinates to screen coordinates
234         """
235         return (x * self.screenScaleX + self.screenX0, y * self.screenScaleY + self.screenY0)
236
237     def drawLoop(self):
238         """
239         Draw the visualisation once and reschedule for next drawing
240         """
241         # delete previous elements from canvas
242         items = self.canvas.find_withtag("dynamic")
```

```
243         for i in items:
244             self.canvas.delete(i)
245 
246         # calculate screen coordinates of the elements
247         baseCoords = self.screenCoords(sim.x, 0)
248         tipCoords = self.screenCoords(sim.x + sim.l * sin(sim.theta), sim.l * cos(sim.theta))
249         xSetCoords = self.screenCoords(sim.xset, 0)
250         xiSetCoords = self.screenCoords(sim.xseti, 0)
251         wheeldotCoords = self.screenCoords(sim.x   0.8*sim.r*sin(sim.alpha),  0.8*sim.r*cos(sim.
    alpha))
252         disturbanceCoords = tipCoords
253         disturbanceCoords = (disturbanceCoords[0] + self.disturbanceOffset, disturbanceCoords[1])
254 
255         # draw the elements
256         self.canvas.create_line(0, self.canvasHeight/2+self.baseRad,
257                                 self.canvasWidth, self.canvasHeight/2+self.baseRad, width=5, tag="
    dynamic")
258         self.canvas.create_line(baseCoords[0], baseCoords[1],
259                                 tipCoords[0], tipCoords[1], width=5, tag="dynamic")
260         self.canvas.create_oval(baseCoords[0] self.baseRad, baseCoords[1] self.baseRad,
261                                 baseCoords[0]+self.baseRad, baseCoords[1]+self.baseRad, width=5, fill="
    white", tag="dynamic")
262         self.canvas.create_oval(baseCoords[0] 1, baseCoords[1] 1,
263                                 baseCoords[0]+1, baseCoords[1]+1, fill="black", tag="dynamic")
264         self.canvas.create_oval(tipCoords[0] self.tipRad, tipCoords[1] self.tipRad,
265                                 tipCoords[0]+self.tipRad, tipCoords[1]+self.tipRad, fill="black", tag="
    dynamic")
266         self.canvas.create_oval(xSetCoords[0] self.setPointRad, xSetCoords[1] self.setPointRad,
267                                 xSetCoords[0]+self.setPointRad, xSetCoords[1]+self.setPointRad, fill="
    green", tag="dynamic")
268         self.canvas.create_oval(xiSetCoords[0] self.setPointRad, xiSetCoords[1] self.setPointRad,
269                                 xiSetCoords[0]+self.setPointRad, xiSetCoords[1]+self.setPointRad, fill=
    "darkgreen", tag="dynamic")
270         self.canvas.create_oval(wheeldotCoords[0] self.wheeldotRad, wheeldotCoords[1] self.
    wheeldotRad,
271                                 wheeldotCoords[0]+self.wheeldotRad, wheeldotCoords[1]+self.wheeldotRad,
     fill="black", tag="dynamic")
272         if self.disturbanceCounter > 0:
273             self.disturbanceCounter  = 1
274             self.canvas.create_oval(disturbanceCoords[0] self.disturbancePointRad,
    disturbanceCoords[1] self.disturbancePointRad,
275                                 disturbanceCoords[0]+self.disturbancePointRad, disturbanceCoords[1]+
    self.disturbancePointRad,
276                                 fill="blue", tag="dynamic")
277 
278         # print simulation time and info
279         self.canvas.create_text(100, 15, text="wall time = %f s" % (clock() sim.t0), tag="dynamic")
280         self.canvas.create_text(100, 30, text="sim time = %f s" % (sim.t), tag="dynamic")
281         self.canvas.create_text(100, 150, text="u: disturbance left\na: disturbance right\n"
282                                 +"n: step left\nt: step right\n"
283                                 +"x: go left\nv: go right\np: toggle periodic\n"
284                                 +"r: reset simulation\nc: toggle controller", tag="dynamic")
285 
286         # schedule redraw after 16 ms for 60 Hz update rate
287         self.root.after(16, self.drawLoop)
288 
289     def keyPressed(self, event):
290         """
291         Key input handling function
292         """
293         if event.char == 'u':
294             # disturbance left
```

```
295                 sim.taue = 3/sim.Ts
296                 self.disturbanceCounter = 10
297                 self.disturbanceOffset = 50
298             elif event.char == 'a':
299                 # disturbance right
300                 sim.taue = 3/sim.Ts
301                 self.disturbanceCounter = 10
302                 self.disturbanceOffset = 50
303             elif event.char == 'n':
304                 # move a step left
305                 sim.xset = pi * sim.r / 4
306             elif event.char == 't':
307                 # move a step right
308                 sim.xset += pi * sim.r / 4
309             elif event.char == 'x':
310                 # set point to left side
311                 sim.xset = 1.5
312             elif event.char == 'v':
313                 # set point to right side
314                 sim.xset = 1.5
315             elif event.char == 'p':
316                 # toggle periodic on/off
317                 sim.periodic = not sim.periodic
318             elif event.char == 'r':
319                 # reset simulation
320                 sim.doReset = True
321             elif event.char == 'c':
322                 # toggle controller on/off
323                 sim.doControl = not sim.doControl
324 
325     def run(self):
326         """
327         Run gui thread
328         """
329         print("Starting gui thread\n")
330         self.root = tk.Tk()
331         self.canvas = tk.Canvas(self.root, width=self.canvasWidth, height=self.canvasHeight,
     background="white")
332         self.canvas.pack()
333         self.root.bind("<Key>", self.keyPressed)
334         self.drawLoop()
335         self.root.mainloop()
336         self.sim.stopSim = True
337 
338 sim = RobotSim()
339 gui = RobotGui(sim)
340 
341 if __name__ == '__main__':
342     sim.start()
343     gui.start()
```