CHAPTER 2

FIRST EXAMPLE: SIMULATING A ROBOTIC TANK

This example serves two purposes. First, it illustrates how hybrid dynamics can appear in engineering problems. The model has three main parts: the equations of motion, a model of the propulsion system, and a model of the computer. The first two are piecewise continuous with discontinuities caused by step changes in the motor voltage and the sticking friction of the rubber tracks. The third model is a prototypical example of a discrete-event system; the tank's computer is modeled with an interruptible server and queue. The equations of motion, propulsion system, and computer are combined to form a complete model of the tank.

Second, this example illustrates the basic elements of a software architecture for large simulation programs. The simulation engine is responsible solely for calculating the dynamic behavior of the model; other functions (visualization and interactive controls, calculation of performance metrics, etc.) are delegated to other parts of the software. This approach is based on two patterns or principles: model–view–control and the experimental frame.

Model–view–control is a pattern widely used in the design of user interfaces (see, e.g., Refs. 47 and 101); the simulation engine and model are treated as a dynamic document and, with this perspective, the overarching design will probably be familiar to most software engineers. The experimental frame (as described, e.g., by Daum and Sargent [31])[1] is a logical separation of the model from the components of the program that provide it with input and observe its behavior. These principles simplify

[1]Be aware, however, of its broader interpretation [152, 157].

reuse; programs for two experiments illustrate how they are applied and the benefit of doing so.

The entirety of this example need not be grasped at once, and its pieces will be revisited as their foundations are established in later chapters. Its purpose here is to be a specific example of how the simulation engine is used, and to motivate the software architecture and algorithms that are discussed in the subsequent chapters of this book.

2.1 FUNCTIONAL MODELING

Fishwick [42] defines a functional model as a thing that transforms input into output. This view of a system is advantageous because it leads to a natural decomposition of the simulation software into objects that implement precisely defined transformations. Distinct functions within the model are described by distinct functional blocks which are connected to form a complete model of the system. The software objects that implement the functional blocks are connected in the same way to build a simulator.

There are numerous methods for designing models. Many of them are quite general: bond graphs and state transition diagrams, for instance. Others are specific to particular problems: the mesh current method for electric circuits and the Lagrangian formulation of a rigid body. The majority of methods culminate in a state space model of a system: a set of state variables and a description of their dynamic behavior. Mathematical formulations of a state space model can take the form of, for example, differential equations, difference equations, and finite-state machines.

To change a state space model into a functional model is simple in principle. The state variables define the model's internal state; state variables or functions of state variables that can be seen from outside the system are the model's output; variables that are not state variables but are needed for the system to evolve become the model's input. In practice, this change requires judgment, experience, and a careful consideration of sometimes subtle technical matters. It may be advantageous to split a state space model into several interacting functional models, or to combine several state space models into a single functional model. Some state space models can be simplified to obtain a model that is easier to work with; simplification might be done with precise mathematical transformations or by simply throwing out terms. The best guides during this process are experience building simulation software, familiarity with the system being studied, and a clear understanding of the model's intended use.

Functional models and their interconnections are the specification for the simulation software. For this purpose, there are two types of functional model: atomic and network. An atomic model has state variables, a state transition function that defines its internal response to input, and an output function that transforms internal action into observable behavior. A network model is constructed from other functional models, and the behavior of the network is defined by the collective behavior of its interconnected components. The simulator is built from the bottom up by implementing atomic models, connecting these to form network models, combining

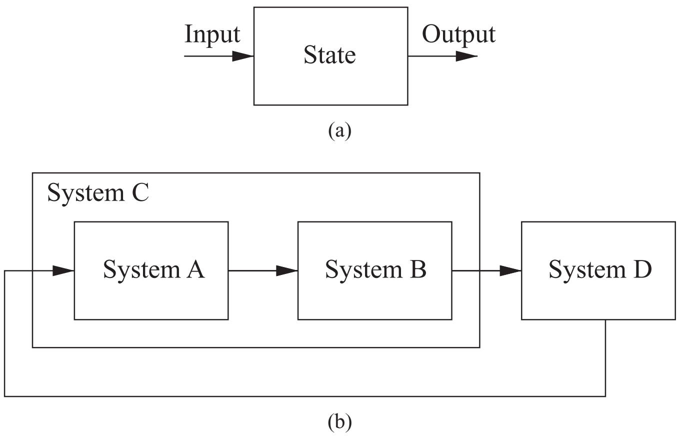


FIGURE 2.1 Bottom–up construction of a model from functional pieces: (a) input, output, and internal state of an atomic model; (b) a network model constructed from three atomic models.

these network models to create larger components, and repeating until the software is finished. This bottom–up approach to model construction is illustrated in Figure 2.1.

The simulation engine operates on software objects that implement atomic and network models. To build a simulator therefore requires the parts of a dynamic system to be expressed in this form. Functional models need not be built in a single step. Atomic and network models are more easily obtained by a set of steps that start with an appropriate modeling technique, proceed to a state space description of the model's fundamental dynamics, combine these to create more sophisticated components, and end with a—possibly large—functional model that can be acted on by the simulation engine.

2.2 A ROBOTIC TANK

The robotic tank is simple enough to permit a thorough discussion of its continuous and discrete dynamics, but sufficiently complicated that it has features present in larger, more practical systems. The robot's operator controls it through a wireless network, and the receipt, storage, and processing of packets is modeled by a discrete event system. An onboard computer transforms the operator's commands into control signals for the motors. The motors and physical motion of the tank are modeled as a continuous system. These components are combined to create a complete model of the tank.

Our goal is to allocate the cycles of the tank's onboard computer to two tasks: physical control of the tank's motors and processing commands from the tank's operator. The tank has four parts that are relevant to our objective: the radio that receives commands from the operator, the computer and software that turn these commands into control signals for the motors, the electric circuit that delivers power to the motors, and the gearbox and tracks that propel the tank. The tank has two tracks, left and right, each driven by its own brushless direct-current (DC) motor. A gearbox connects each motor to the sprocket wheel of its track. The operator drives the tank by setting the duty ratio of the voltage signal at the terminals of the motors. The duty ratio are set using the control sticks on a gamepad and sent via a wireless network to the computer.

The computer generates two periodic voltage signals, one for each motor. The motor's duty ratio is the fraction of time that it is turned on in one period of the signal (i.e., its ON time). Because the battery voltage is fixed, the power delivered to a motor is proportional to its duty ratio. Driving the tank is straightforward. If the duty ratio of the left and right motors are equal then the tank moves in a straight line. The tank spins clockwise if the duty ratio of the left motor is higher than that of the right motor. The tank spins counterclockwise if the duty ratio of the right motor is higher than that of the left motor. A high duty ratio causes the tank to move quickly; a low duty ratio causes the tank to move slowly.

If the voltage signal has a high frequency, then the inertia of the motor will carry it smoothly through moments when it is disconnected from the batteries; the motors operate efficiently and the tank handles well. If the frequency is too low, then the motor operates inefficiently. It speeds up when the batteries are connected, slows down when they are disconnected, and speeds up again when power is reapplied. This creates heat and noise, wasting energy and draining the batteries without doing useful work. Therefore, we want the voltage signal to have a high frequency.

Unfortunately, a high-frequency signal means less time for the computer to process data from the radio. If the frequency is too high, then there is a noticeable delay as the tank processes commands from the operator. At some point, the computer will be completely occupied with the motors, and when this happens, the tank becomes unresponsive.

Somewhere in between is a frequency that is both acceptable to the driver and efficient enough to give a satisfactory battery life. There are physical limits on the range of usable frequencies. It cannot be so high that the computer is consumed entirely by the task of driving the motors. It cannot be so low that the tank lurches uncontrollably or overheats its motors and control circuits. Within this range, the choice of frequency depends on how sensitive the driver is to the nuances of the tank's control.

An acceptable frequency could be selected by experimenting with the real tank; let a few people drive it around using different frequencies and see which they like best. If we use the real tank to do this, then we can get the opinions of a small number of people about a small number of frequencies. The tank's batteries are one constraint on the number of experiments that can be conducted. They will run dry after a few trials and need several hours to recharge. That we have only one tank is another constraint. Experiments must be conducted one at a time. If, however, we build a simulation of the tank, then we can give the simulator to anyone who cares to render an opinion, and that person can try as many different frequencies as time and patience permit.

TABLE 2.1 Value of Parameters Used in the Tank's Equations of Motion

Parameter	Value	Description
m_t	0.8 kg	Mass of the tank
J_t	5×10^{-4} kg · m^2	Angular mass of the tank
B	0.1 m	Width of the tank from track to track
B_r	1.0 N · s / m	Mechanical resistance of the tracks to rolling forward
B_s	14.0 N · s / m	Mechanical resistance of the tracks to sliding forward
B_l	0.7 N · m · s / rad	Mechanical resistance of the tracks to turning
S_l	0.3 N · m	Lateral friction of the tracks

2.2.1 Equations of Motion

The model of the tank's motion is adapted from Anh Tuan Le's PhD dissertation [74]. The model's parameters are listed in Table 2.1, and the coordinate system and forces acting on the tank are illustrated in Figure 2.2. The model assumes that the tank is driven on a hard, flat surface and that the tracks do not slip. The position of the tank is given by its x and y coordinates. The heading θ of the tank is measured with respect to the x axis of the coordinate system and the tank moves in this direction with a speed v.

The left track pushes the tank forward with a force F_l; the right track, with a force F_r; and B_r is the mechanical resistance of the tracks to rolling. The tank uses skid steering; to turn, the motors must collectively create enough torque to cause the tracks to slide sideways. This requires overcoming the sticking force S_l. When sufficient torque is created, the vehicle begins to turn. As it turns, some of the propulsive force is expended to drag the tracks laterally; this is modeled by an additional resistance B_l to its turning motion and B_s to its rolling motion.

The tank's motion is described by two sets of equations, one for when the tank is turning and one for when it is not. The switch from turning to not turning (and vice

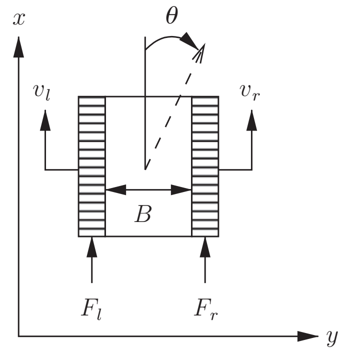


FIGURE 2.2 Coordinate system, variables, and parameters used in the tank's equations of motion.

versa) has two discrete effects: (1) the angular velocity ω changes instantaneously to and remains at zero when the tracks stick and the turn ends, and (2) the rolling resistance of the tank changes instantaneously when the tank starts and ends a turn. The Boolean variable *turning* is used to change the set of equations. The equations that model the motion of the tank are

$$turning = \begin{cases} \text{true} & \text{if } \frac{B}{2}|F_l - F_r| \geq S_l \\ \text{false} & \text{otherwise} \end{cases} \tag{2.1}$$

$$\dot{v} = \begin{cases} \frac{1}{m_t}\left(F_l + F_r - (B_r + B_s)v\right) & \text{if } turning = \text{true} \\ \frac{1}{m_t}\left(F_l + F_r - B_r v\right) & \text{if } turning = \text{false} \end{cases} \tag{2.2}$$

$$\dot{\omega} = \begin{cases} \frac{1}{J_t}\left(\frac{B}{2}(F_l - F_r) - B_l\omega\right) & \text{if } turning = \text{true} \\ 0 & \text{if } turning = \text{false} \end{cases} \tag{2.3}$$

$$\dot{\theta} = \omega \tag{2.4}$$

$$\dot{x} = v\sin(\theta) \tag{2.5}$$

$$\dot{y} = v\cos(\theta) \tag{2.6}$$

$$\text{If } turning = \text{false then } \omega = 0 \tag{2.7}$$

When *turning* changes from false to true, every state variable evolves from its value immediately prior to starting the turn, but using the equations designated for *turning* = true. When *turning* changes from true to false, every state variable except ω evolves from its value immediately prior to ending the turn, but using the equations designated for *turning* = false; ω changes instantaneously to zero and remains zero until the tank begins to turn again.

These differential equations describe how the tank moves in response to the propulsive force of the tracks. The track forces F_l and F_r are inputs to this model, and we can take any function of the state variables—v, ω, θ, x, and y—as output. For reasons that will soon become clear, we will use the speed with respect to the ground of the left and right treads; Figure 2.2 illustrates the desired quantities. The speed v_l of the left tread and speed v_r of the right tread are determined from the tank's linear speed v and rotational speed ω by

$$v_l = v + B\omega/2 \tag{2.8}$$

$$v_r = v - B\omega/2 \tag{2.9}$$

The dependence of the input on the output is denoted by the function

$$\begin{bmatrix} v_l(t) \\ v_r(t) \end{bmatrix} = M\left(\left[F_l(t)\ F_r(t)\right]^T\right) \tag{2.10}$$

This function accepts the left and right tread forces as input and produces the left and right tread speeds as output.

How were the values in Table 2.1 obtained? Two of them were measure directly: the mass of the tank with a postal scale and the width of the tank with a ruler. The angular mass of the tank is an educated guess. Given the width w and length l of the tank's hull, which were measured with a ruler, and the mass, obtained with a postal scale, the angular mass is computed by assuming the tank is a uniformly dense box. With these data and assumptions, we have

$$J_t = \frac{m_t}{12}(w^2 + l^2)$$

This is not precise, but it is the best that can be obtained with a ruler and scale.

The resistance parameters are even more speculative. The turning torque S_l was computed from the weight W of the tank and length l_t of the track, which were both measured directly, a coefficient of static friction μ_s for rubber from Serway's *Physics for Scientists and Engineers* [133], and the approximation

$$S_l = \frac{W l_t \mu_s}{3}$$

from Le's dissertation [74]. The resistances B_r and B_s to forward motion and resistance B_l to turning were selected to give the model reasonable linear and rotational speeds.

This mix of measurements, rough approximations, and educated guesses is not uncommon. It is easier to build a detailed model than to obtain data for it. The details, however, are not superfluous. The purpose of this model is to explore how the tank's response to the driver changes with the frequency of the power signal sent to the motors. For this purpose it is necessary to include those properties of the tank that determine its response to the intermittent voltage signal: specifically, inertia and friction.

2.2.2 Motors, Gearbox, and Tracks

The motors, gearbox, and tracks are an electromechanical system for which the method of bond graphs is used to construct a dynamic model (Karnopp et al. [61] give an excellent and comprehensive introduction to this method). The bond graph model is coupled to the equations of motion by using Equation 2.10 as a bond graph element. This element has two ports, one of which has the effort variable F_l and flow variable v_l, and the other, the effort variable F_r and flow variable v_r. The causality of this element is determined by the functional form of Equation 2.10: it is supplied with the effort variables and produces the flow variables. This was the reason for selecting the track speeds as output.

The model of the motors, gearbox, and tracks accounts for the inductance and internal resistance of the electric motors, the angular mass and friction of the gears, and the compliance of the rubber tracks. The electric motors are Mabuchi FA-130 Motors, the same type of DC motor that is ubiquitous in small toys. One motor drives each track. The motors are plugged into a Tamiya twin-motor gearbox. This gearbox has two sets of identical, independent gears that turn the sprocket wheels. The sprocket wheels and tracks are from a Tamiya track-and-wheel set; the tracks stretch when the tank accelerates (in hard turns this causes the tracks to come off the wheels!), and so their compliance is included in the model.

To drive the motors, the computer switches a set of transistors in an Allegro A3953 full-bridge pulsewidth-modulated (PWM) motor driver. When the switches are closed, the tank's batteries are connected to the motors. When the switches are open, the batteries are disconnected from the motors. The transistors can switch on and off at a rate three orders of magnitude greater than the rate at which the computer can operate them, and power lost in the circuit is negligible in comparison to inefficiencies elsewhere in the system. The batteries and motor driver are, therefore, modeled as an ideal, time varying voltage source.

A sketch of the connected motors, gearbox, and tracks and its bond graph are shown in Figure 2.3. Table 2.2 lists the parameters used in this model. The differential

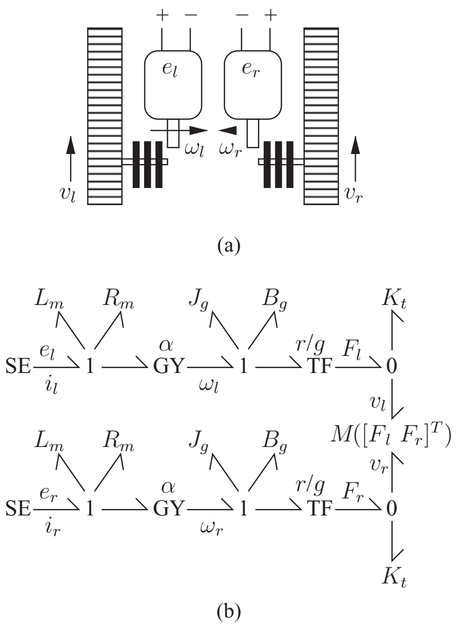

+ −
− +
e_l
e_r
ω_l
ω_r
v_l
v_r
(a)
L_m R_m J_g B_g K_t
SE e_l i_l 1 GY α ω_l 1 TF r/g F_l 0
v_l
$M([F_l\ F_r]^T)$
v_r
L_m R_m J_g B_g
SE e_r i_r 1 GY α ω_r 1 TF r/g F_r 0
K_t
(b)

FIGURE 2.3 Motors, gears, and tracks of the tank: (a) diagram; (b) bond graph.

TABLE 2.2 Parameters of the Motors, Gearbox, and Tracks

Parameter	Value	Description
L_m	10^{-3} H	Inductance of the motor
R_m	3.1 Ω	Resistance of the motor
J_g	1.2×10^{-6} kg · m^2	Angular mass of the gears
B_g	6.7×10^{-7} N · m · s / rad	Mechanical resistance of the gears to rotation
g	204	Gear ratio of the gearbox
α	10^{-3} N · m / A	Current–torque ratio of the electric motor
r	0.015 m	Radius of the sprocket wheel
K_t	10^{-3} m / N	Compliance of the track

equations are read directly from the bond graph:

$$\dot{i}_l = \frac{1}{L_m}(e_l - i_l R_m - \alpha\omega_l) \tag{2.11}$$

$$\dot{\omega}_l = \frac{1}{J_g}\left(\alpha i_l - \omega_l B_g - \frac{r}{g}F_l\right) \tag{2.12}$$

$$\dot{F}_l = \frac{1}{K_t}\left(\frac{r}{g}\omega_l - v_l\right) \tag{2.13}$$

$$\dot{i}_r = \frac{1}{L_m}(e_r - i_r R_m - \alpha\omega_r) \tag{2.14}$$

$$\dot{\omega}_r = \frac{1}{J_g}\left(\alpha i_r - \omega_r B_g - \frac{r}{g}F_r\right) \tag{2.15}$$

$$\dot{F}_r = \frac{1}{K_t}\left(\frac{r}{g}\omega_r - v_r\right) \tag{2.16}$$

where e_l and e_r are the motor voltages and v_l and v_r are the track speeds given by Equations 2.8 and 2.9.

Values for the parameters in Table 2.2 were obtained from manufacturers' data, from measurements, and by educated guesses. The gear ratio g and current–torque ratio α are provided by the manufacturers. The gear ratio is accurate and precise (especially with respect to the values of other parameters in the model). The current–torque ratio is an average of the two cases supplied by Mabuchi, the motor's manufacturer. The first case is the motor operating at peak efficiency, and the second case is the motor stalling. The difference between these two cases is small, suggesting that α does not vary substantially as the load on the motor changes. The estimate of α is, therefore, probably very reasonable.

The radius r of the sprocket wheel and the resistance R_m and inductance L_m of the motor were measured directly. A ruler was used to measure the radius of the sprocket wheel. To determine R_m and L_m required more effort. The current i through the unloaded motor is related to the voltage e across the motor by the differential equation

$$\dot{i} = \frac{1}{L_m}(e - iR_m) \tag{2.17}$$

The parameters L_m and R_m were estimated by connecting a 1.5-V C battery to the motor and measuring, with an oscilloscope, the risetime and steady state of the current through the motor. Let i_f be the steady-state current, t_r the risetime, and $0.9i_f$ the current at time t_r (i.e., the risetime is the amount of time to go from zero current to 90% of the steady-state current). At steady state $\dot{i} = 0$ and the resistance of the motor is given by

$$R_m = \frac{e}{i_f}$$

The transient current is needed to find L_m. The transient current is given by the solution to Equation 2.17:

$$i(t) = \frac{e}{R_m}\left(1 - \exp\left(-\frac{R_m}{L_m}t\right)\right) \tag{2.18}$$

Substituting $0.9i_f$ for $i(t)$ and t_r for t in Equation 2.18 and solving for L_m gives

$$\frac{1}{L_m} = -\frac{1}{R_m t_f}\ln\left(1 - 0.9\frac{R_m}{e}i_f\right) = -\frac{i_f}{et_f}\ln(0.1) \approx 2.3\frac{i_f}{et_f}$$

or, equivalently

$$L_m \approx 0.652\,\frac{t_f}{i_f}$$

A similar experiment was used to obtain B_g. In this experiment, the motor was connected to the gearbox. As before, an oscilloscope was used to measure the risetime and steady-state value of the current through the motor. The rotational velocity $\tilde{\omega}$ of the motor is given by

$$\dot{\tilde{\omega}} = \frac{1}{J_g}(\alpha i - \tilde{\omega}B_g)$$

The manufacturer gives the speed of the motor when operating at peak efficiency as $\tilde{\omega} = 731.6$ radians per second (rad/s). At steady state $\dot{\tilde{\omega}} = 0$. The steady state current

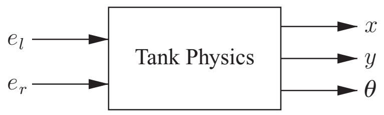


FIGURE 2.4 Input and output of the model of the tank's physics.

i_f is measured with the oscilloscope. With i_f and the motor speed, the mechanical resistance of the gearbox is given by

$$B_g \approx \frac{\alpha i_f}{\tilde{\omega}} = 1.37 \times 10^{-6}\, i_f$$

The angular mass J_g of the gearbox was estimated from its mass m_{gb}, radius of the gears r_{gb}, and the assumption that the mass is uniformly distributed in a cylinder. With this set of measurements and assumptions, the angular mass is

$$J_g = m_{gb} r_{gb}^2$$

The compliance of the tracks is an order-of-magnitude approximation. The tracks can be stretched by only a few millimeters before they slip off the wheels. The maximum propulsive force of the track is about a newton. The order of magnitude of the track compliance is, therefore, estimated to be 10^{-3} meters/10^{0} newtons, or about 10^{-3} m/N.

2.2.3 Complete Model of the Tank's Continuous Dynamics

Equations 2.1–2.9 and 2.11–2.16 collectively describe the physical behavior of the tank. The equations of motion and the equations for the motors, gearbox, and tracks were developed separately, but algorithms for solving them work best when coupled equations are lumped together. Consequently, these are put into a single functional model called "tank physics," which is illustrated in Figure 2.4. The inputs to the tank are the voltages across its left and right motors; these come from the computer. The output of the tank is its position and heading; these are observed by the tank's operator. The complete state space model of the tank's physical dynamics is

$$\mathit{turning} = \begin{cases} \text{true} & \text{if } \frac{B}{2}|F_l - F_r| \geq S_l \\ \text{false} & \text{otherwise} \end{cases} \tag{2.19}$$

$$\dot{v} = \begin{cases} \frac{1}{m_t}\Big(F_l + F_r - (B_r + B_s)v\Big) & \text{if } \mathit{turning} = \text{true} \\ \frac{1}{m_t}\Big(F_l + F_r - B_r v\Big) & \text{if } \mathit{turning} = \text{false} \end{cases} \tag{2.20}$$

$$\dot{\omega} = \begin{cases} \frac{1}{J_t}\left(\frac{B}{2}(F_l - F_r) - B_l\omega\right) & \text{if } \textit{turning} = \text{true} \\ 0 & \text{if } \textit{turning} = \text{false} \end{cases} \tag{2.21}$$

$$\dot{\theta} = \omega \tag{2.22}$$

$$\dot{x} = v\sin(\theta) \tag{2.23}$$

$$\dot{y} = v\cos(\theta) \tag{2.24}$$

$$\text{If } \textit{turning} = \text{false then } \omega = 0 \tag{2.25}$$

$$\dot{i_l} = \frac{1}{L_m}(e_l - i_l R_m - \alpha\omega_l) \tag{2.26}$$

$$\dot{\omega_l} = \frac{1}{J_g}\left(\alpha i_l - \omega_l B_g - \frac{r}{g}F_l\right) \tag{2.27}$$

$$\dot{F_l} = \frac{1}{K_t}\left(\frac{r}{g}\omega_l - \left(v + \frac{B\omega}{2}\right)\right) \tag{2.28}$$

$$\dot{i_r} = \frac{1}{L_m}(e_r - i_r R_m - \alpha\omega_r) \tag{2.29}$$

$$\dot{\omega_r} = \frac{1}{J_g}\left(\alpha i_r - \omega_r B_g - \frac{r}{g}F_r\right) \tag{2.30}$$

$$\dot{F_r} = \frac{1}{K_t}\left(\frac{r}{g}\omega_r - \left(v - \frac{B\omega}{2}\right)\right) \tag{2.31}$$

This model has 11 state variables—v, ω, θ, x, y, i_l, ω_l, F_l, i_r, ω_r, and F_r; two input variables—e_l and e_r; and three output variables—x, y, and θ.

2.2.4 The Computer

The computer, a TINI microcontroller from Maxim, receives commands from the operator through a wireless network and transforms them into voltage signals for the motors. The computer extracts raw bits from the Ethernet that connects the computer and the radio, puts the bits through the Ethernet and User Datagram Protocol (UDP) stacks to obtain a packet, obtains the control information from that packet, and stores that information in a register where the interrupt handler that generates voltage signals can find it. The interrupt handler runs periodically, and it has a higher priority than the thread that processes commands from the operator. Therefore, time spent in the interrupt handler is not available to process commands from the operator.

The frequency of the voltage signal is determined by the frequency of the interrupt handler. Frequent interrupts create a high-frequency voltage signal; infrequent interrupts, a low-frequency signal. Figure 2.5 illustrates how the interrupt handler works. It is executed every N machine cycles and at each invocation adds 32 to a counter stored in an 8-bit register. The counter is compared to an ON time that is set,

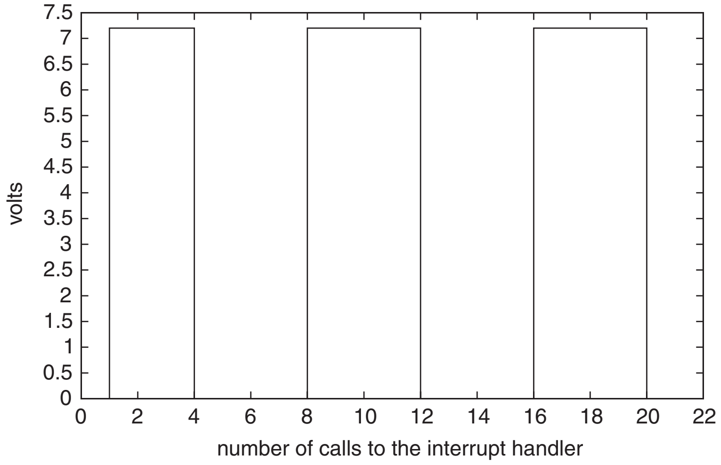


FIGURE 2.5 Generating a voltage signal with the interrupt handler.

albeit indirectly, by the operator. If the counter is greater than or equal to the ON time, then the motor is turned off. If the counter is less than the ON time, then the motor is turned on. If, for example, the tank is operating at full power, then the ON time for both motors is 255 and the motors are always on; if the motors are turned off, then the ON time is zero.

In Figure 2.5, the counter is initially zero and the motors are turned off. The ON time is 128. The first call to the interrupt handler adds 32 to the counter, compares $32 < 128$, and turns the motor on by connecting it to the tank's 7.2-V battery pack. At call 4, the counter is assigned a value of 128, which is equal to the ON time, and the motor is shut off. At call 8, the counter rolls over and the motor is turned on again.

The code in the interrupt handler is short; it has 41 assembly instructions that require 81 machine cycles to execute. According to the computer's manufacturer, there are 18.75×10^6 machine cycles per second, which is one cycle every 0.0533×10^{-6}s (0.0533 μs). The interrupt handler, therefore, requires 0.432×10^{-6} s (0.432 μs) to execute. The frequency of the voltage signal is determined by how quickly the interrupt handler rolls the counter over. On average, eight calls to the interrupt handler complete one period of the voltage signal. The length of this period is $8 \times (0.432 \times 10^{-6} + 0.0533 \times 10^{-6} \times N)$. We can choose N and thereby select the period of the voltage signal; the frequency f_e due to this selection is

$$f_e \approx \frac{10^6}{3.46 + 0.426N} \tag{2.32}$$

The discrete-event model of the interrupt handler has two types of events: *Start interrupt* and *End interrupt*. The *Start interrupt* event sets the interrupt indicator to true and schedules an *End interrupt* to occur 0.432×10^{-6} s later. The *End interrupt* event increments the counter, sets the motor switches, sets the interrupt indicator to

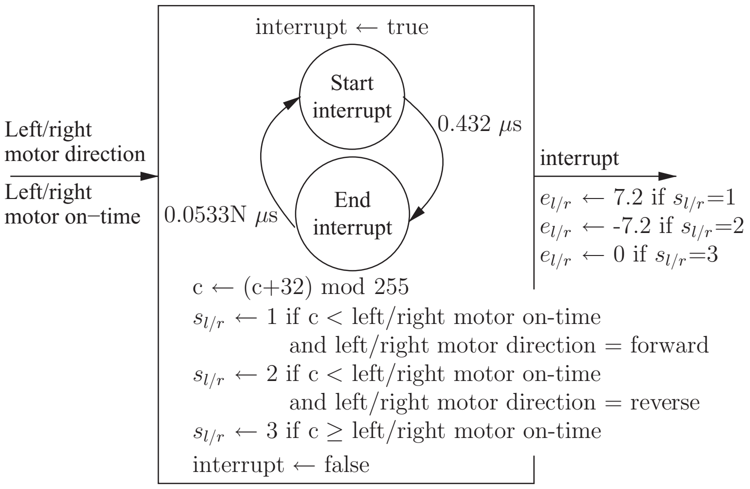

interrupt ← true
Start interrupt
End interrupt
0.432 μs
0.0533N μs
Left/right motor direction
Left/right motor on−time
interrupt
$e_{l/r}$ ← 7.2 if $s_{l/r}$=1
$e_{l/r}$ ← -7.2 if $s_{l/r}$=2
$e_{l/r}$ ← 0 if $s_{l/r}$=3
c ← (c+32) mod 255
$s_{l/r}$ ← 1 if c < left/right motor on-time and left/right motor direction = forward
$s_{l/r}$ ← 2 if c < left/right motor on-time and left/right motor direction = reverse
$s_{l/r}$ ← 3 if c ≥ left/right motor on-time
interrupt ← false

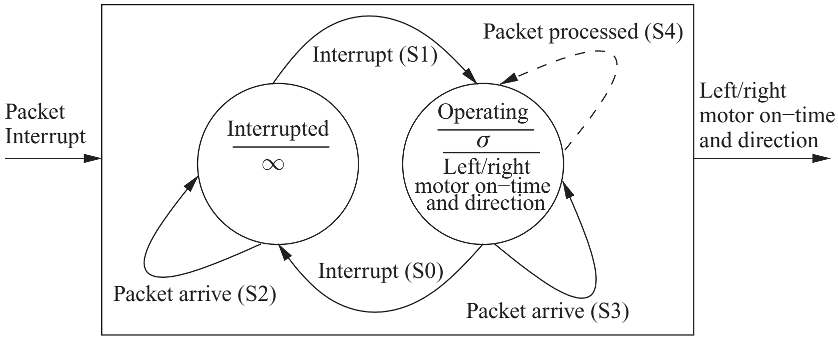
FIGURE 2.6 Event graph for the interrupt handler.

false, and schedules a *Start interrupt* event to occur $0.0533 \times 10^{-6}N$ s later. There are two software switches, one for each motor, and each switch has three positions. If the software switch is in the first position, then the motor is connected to the tank's 7.2-V battery pack. If the switch is in the second position, then the motor is connected to the batteries but the positive and negative terminals are reversed and the motor runs backward. In the third position, the motor is disconnected from the batteries. At any given time, a new ON time and direction for either motor can be given to the interrupt handler, and it acts on the new settings when the next *End interrupt* event occurs.

An event graph for the interrupt handler is shown in Figure 2.6 (event graphs were introduced by Schruben [131]; Fishwick [42] describes their use in functional models). The model has nine state variables. Four of these are apparent in the diagram: the 8-bit counter c, the *interrupt* indicator, and the switches s_l and s_r for the left and right motors. Events that change the ON time and direction of a motor are inputs to the model; these input variables are stored as the *left motor ON time*, *left motor direction*, *right motor ON time*, and *right motor direction*, bringing the count of state variables to eight. Implicit in the edges that connect the events is the time until the next event occurs, which is the ninth and final state variable for this system. The outputs from this model are the interrupt indicator and the left and right motor voltages. The output variables change immediately after the corresponding state variables. In the case that an event is scheduled at the same time that an input variable is changed, the event is executed first and then the corresponding variables are modified.

When the computer is not busy with its interrupt handler, it is processing commands from the operator. Every command arrives as a UDP packet with 10 bytes: two floating-point numbers that specify the direction and duty ratio of the left and right motors, and 2 bytes of information that are not relevant to our model. The computer can receive data at a rate of about 40 kilobytes per second (kB). This estimate, which comes from the jGuru forum [120], agrees reasonably well with, but is slightly lower than, the maximum data rate given by the manufacturer. The computer talks to the 802.11b radio through an Ethernet (the radio is controlled by a separate microprocessor) that has a minimum packet size of 64 bytes, much larger than the 10-byte payload. Consequently, we can optimistically estimate that processing a packet takes 0.0016 s (1.6 ms, or 1600 μs). We will ignore packet losses and assume that the computer (or, at least, the radio) can store any number of unprocessed packets. This is modeled with a server that has a fixed service time and an infinite queue. When the interrupt handler is executing, the server is forced to pause. The server produces ON times and directions for the motors when it finishes processing a packet.

Figure 2.7 is a DEVS graph (described by Zeigler et. al. [159] and, more recently, by Schulz et. al. [132]; these are sometimes called *phase graphs* [42]) for this model. It has three state variables: the packet queue q; the time σ remaining to process the packet at the front of the queue; and the model's phase, which is *interrupted* or *operating*. It responds to two types of input: the interrupt indicator from the interrupt handler and packets from the network. The interrupt indicator moves the system into its *interrupted* phase where it remains until a second interrupt indicator is received, and this moves the system back to its *operating* phase. When the computer finishes processing a packet, it sets the ON time and direction for the left and right motors and begins to process the next packet or, if there are no packets left, becomes idle. Each edge in the phase graph is annotated with the state variables that change when the phase transition occurs. Each phase contains its duration and the output value that is generated when the phase is left because its duration has expired.



S0: If $\sigma < \infty$ then decrement σ by the elapsed time
S1: If q is not empty and $\sigma = \infty$ then $\sigma \leftarrow 1.6$ ms
S2: Insert packet at the back of q
S3: If q is empty then $\sigma \leftarrow 1.6$ ms
else decrement σ by the elapsed time
Insert packet at the back of q
S4: Remove the first element from q
If q is empty then $\sigma \leftarrow \infty$
else $\sigma \leftarrow 1.6$ ms

FIGURE 2.7 Phase graph showing how the computer processes a packet.

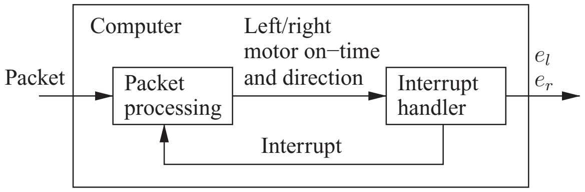


FIGURE 2.8 Block diagram of the tank's computer.

The models of the interrupt handler and thread that processes packets are connected to create a model of the computer. The interrupt handler receives the motor ON times from the thread that processes packets; the thread receives interrupt indicators from the interrupt handler and packets from the network. The output from the computer sets the voltage at the left and right motors. Figure 2.8 shows a block diagram of the computer with its inputs, outputs, and internal components.

The event graph and phase diagram are informative but not definitive. They do not specify when, precisely, output is produced or how to treat simultaneous events. These issues are deferred to Chapter 4, where state space models of discrete-event systems are formally introduced.

2.2.5 Complete Model of the Tank

The complete model of the tank comprises the computer and the tank's physics. The output of the computer is connected to the input of the tank's physics. The position and orientation of the tank are displayed for the driver. The driver closes the loop by sending packets with control information to the computer. This arrangement is shown in Figure 2.9. The tank's operator is not a model; the operator controls the simulated tank with the same software and hardware that are used to control the real tank.

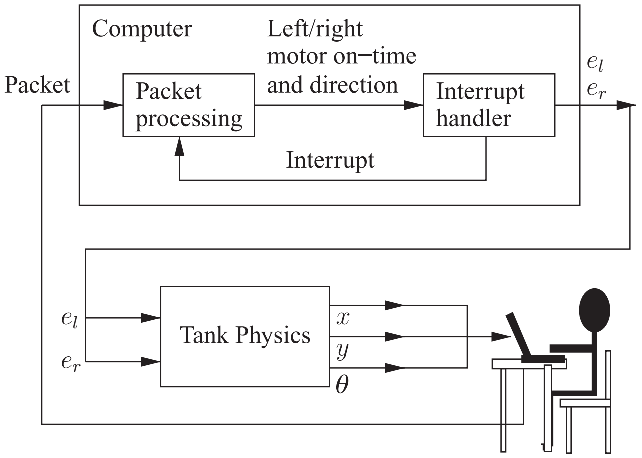


FIGURE 2.9 Block diagram of the simulated tank and real operator.

2.3 DESIGN OF THE TANK SIMULATOR

The simulator has four parts: the simulation engine, the model of the tank, the driver's interface, and the network interface. Figure 2.10 shows the classes that implement these parts and their relationships. The simulation engine and tank, which are our main concern, are implemented by the *Simulator* class and *SimEventListener* interface and the *Tank* class, respectively. The user interface is implemented by the *Display* class and *DisplayEventListener* interface, which take input from the user and display the motion of the tank. The *UDPSocket* class implements the network interface by which the simulator receives commands from the driver.

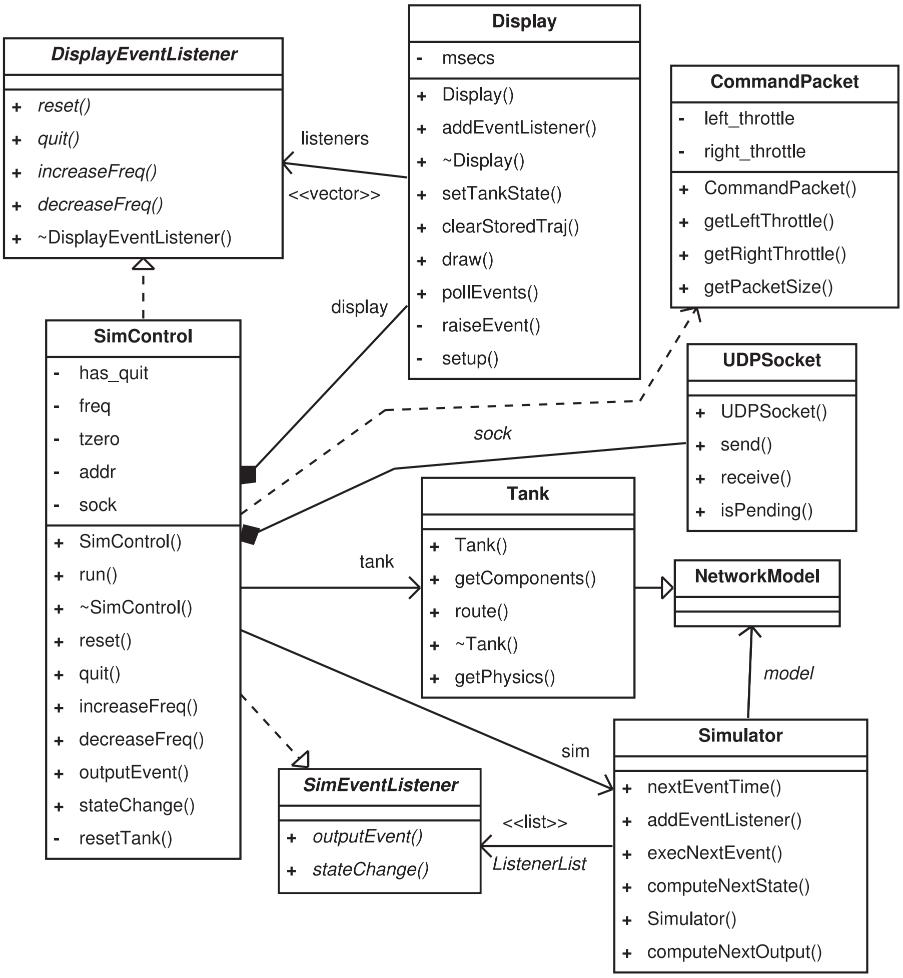


FIGURE 2.10 Class diagram showing the major components of the simulation software.

The *SimControl* class implements the main loop of the application in its *run* method. This method advances the simulation clock in step with the real clock, updates the *Display*, and polls the *Simulator*, *Display*, and *UDPSocket* for new events. The *SimControl* class implements the *SimEventListener* interface by which it is notified when components of the model change state and produce output. These callbacks are received when the *SimControl* object calls the *Simulator*'s *computeNextState* method. The *SimControl* also implements the *DisplayEventListener* class by which it is notified when the user does something to the display: for instance, pressing the quit key "q" or pressing the simulation reset key "r". These callbacks are received when the *SimControl* calls the *Display*'s *pollEvents* method. The *SimControl* object extracts *CommandPacket*s from the network by polling the *UDPSocket*'s *pendingInput* method at each iteration of the main loop.

The *Simulator* has six methods. The constructor accepts a model—it can be a multilevel, multicomponent model or a single atomic model—that the simulator will operate on. The method *nextEventTime* returns the time of the simulator's next event: the next time at which some component will produce output or change state in the absence of an intervening input. The method *computeNextOutput* provides the model's outputs at the time of its next event without actually advancing the model's state. The method *computeNextState* advances the simulation clock and injects into the model any input supplied by the caller. Objects that implement the *SimEventListener* interface and register themselves by calling the *addEventListener* method are notified by the *Simulator* when a component of the model produces an output or changes its state. These notifications occur when *computeNextState* or *computeNextOutput* is called.

Missing from Figure 2.10 are the details of how the *Tank* is implemented; its major components are shown in Figure 2.11. The relationship between the *Tank* and *Simulator* is important. The *Simulator* is designed to operate on a connected collection

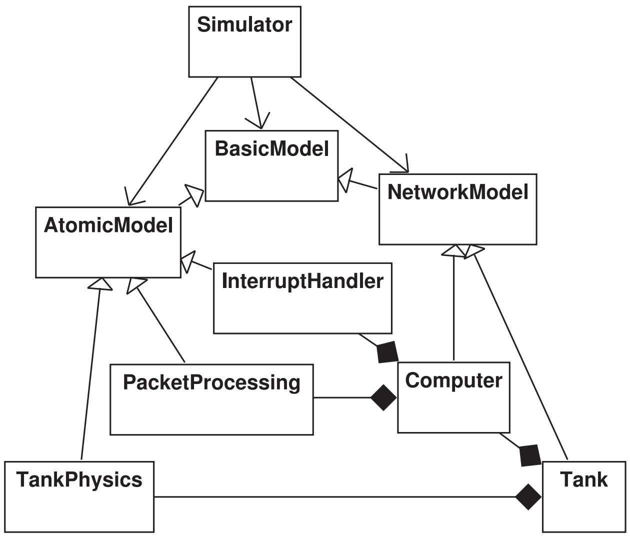


FIGURE 2.11 Class diagram showing the major components of the model.

of state space models; the *Tank* is a specific instance of such a model. The parts of the tank are derived, ultimately, from two fundamental classes: the *AtomicModel* class and the *NetworkModel* class. The *Tank* and *Computer* classes are derived from *NetworkModel* and they implement the block diagrams shown in Figures 2.8 and 2.9. The *TankPhysics* class, derived from *AtomicModel*, implements Equations 2.19–2.31. The *PacketProcessing* and *InterruptHandler* classes, also derived from *AtomicModel*, implement the models shown in Figures 2.6 and 2.7.

This design separates the three aspects of our simulation program. The *SimControl* class coordinates the primary activities of the software: rendering the display, receiving commands from the network, and running the simulation. It uses the *Simulator*'s six methods to control the simulation clock, inject input into the model, and obtain information about the model's state and output.

The *Simulator* and its myriad supporting classes (which are not shown in the diagrams) implement algorithms for event scheduling and routing, numerical integration, and other essential tasks. These algorithms operate on the abstract *AtomicModel* and *NetworkModel* classes without requiring detailed knowledge of the underlying dynamics.

Models are implemented by deriving concrete classes from *AtomicModel* and *NetworkModel*. Models derived from the *AtomicModel* class implement state space representations of the continuous and discrete-event components. Models derived from the *NetworkModel* class describe how collections of state space and network models are connected to form the complete system.

2.4 EXPERIMENTS

Before experimenting with the simulated tank, we must establish the range of frequencies that are physically feasible. An upper limit can be derived without simulation. Suppose that the computer does nothing except execute the interrupt handler. With zero instructions between invocations of the interrupt handler, Equation 2.32 gives a maximum frequency of 289 kHz for the voltage signal. At this frequency, the computer has no time to process commands from the driver and, consequently, the tank cannot be controlled.

To determine a lower limit we simulate the tank running at half-throttle and measure the power dissipated in the motors. After examining the power lost at several frequencies, we can pick the lowest acceptable frequency as the one for which higher frequencies do not significantly improve efficiency. The software for this simulation is much simpler than for the interactive simulation, but it uses all of the classes shown in Figure 2.11 and the *SimEventListener* and *Simulator* classes shown in Figure 2.10. The classes that implement the model of the tank do not change: Figure 2.11 is precisely applicable. The remainder of the program, all of the new code that must be implemented to conduct this experiment, has fewer than 100 lines.

The main function creates a *Tank*; a *Simulator* for the *Tank*; and a *TankEventListener*, which computes the power lost in the motors. The *TankEventListener* is derived from the *SimEventListener* class. After registering the *TankEventListener* with the *Simulator*, the program injects a *SimPacket* into the tank at time zero. This packet contains the duty ratio for the left and right motors. Now the simulation is run for 3 s, long enough for the tank to reach is maximum speed of approximately 0.2 m/s and run at that speed for a little over 2 s. The power P lost in the motors is

$$P = \frac{1}{3}\int_0^3 i_l(t)^2 R_m + i_r(t)^2 R_m \, dt$$

$$\approx \frac{1}{t_M}\sum_{k=0}^{M-1}(t_{k+1} - t_k)(i_{l,k}^2 R_m + i_{r,k}^2 R_m) \tag{2.33}$$

where the t_k are the times at which the *stateChange* method of the *TankEventListener* is called and $i_{l,k}$ and $i_{r,k}$ are the currents at time t_k. Note that $t_0 = 0$ and t_M may be slightly less than 3, depending on how the simulator selects timesteps for its integration algorithm (it could be made to update the state of the tank at $t = 3$, but was not in this instance).

The *stateChange* method of the *TankEventListener* is called every time the *Simulator* computes a new state for an atomic component of the *Tank*. When this occurs, the *TankEventListener* calculates one step of the summation in Equation 2.33. The *getPowerLost* method computes the lost power by dividing the lost energy by the elapsed time. The C++ code that implements the *TankEventListener* is shown below.

TankEventListener

```
#ifndef TankEventListener_h
#define TankEventListener_h
#include "Tank.h"
#include "SimEvents.h"
#include "SimEventListener.h"
#include <fstream>

class TankEventListener: public SimEventListener
{
   public:
      TankEventListener(const Tank* tank):
         SimEventListener(),
         tank(tank),fout("current.dat"),
         E(0.0), // Accumulated energy starts at zero
         tl(0.0), // First sample is at time zero
         il(tank->getPhysics()->leftMotorCurrent()), // i_l(0)
         ir(tank->getPhysics()->rightMotorCurrent()) // i_r(0)
      {
         fout << tl << " " << il << " " << ir << std::endl;
      }
      // Listener does nothing with output events
      void outputEvent(ModelInput, double){}
      // This method is invoked when an atomic component changes state
      void stateChange(AtomicModel* model, double t)
      {
         // If this is the model of the tank's physics
         if (model == tank->getPhysics()) {
            // Get the current and motor resistance
            double Rm = tank->getPhysics()->getMotorOhms();
            // Update the enery dissipated in the motors
            E += (t-tl)*(il*il*Rm + ir*ir*Rm);
            // Remember the last sample
            il = tank->getPhysics()->leftMotorCurrent();
            ir = tank->getPhysics()->rightMotorCurrent();
            tl = t;
            fout << tl << " " << il << " " << ir << std::endl;
         }
      }
      // Get the power dissipated in the left and right motors
      double getPowerLost() const { return E/tl; }
   private:
      const Tank* tank;
      std::ofstream fout;
      double E, tl, il, ir;
};

#endif
```

A shell script calls the main program repeatedly to conduct the simulation experiment. Each invocation of the simulation program computes the power dissipated in the motors at one frequency and pair of duty ratios. The first argument to the simulation program is the frequency of the voltage signal sent to the motors, the second argument is the duty ratio for the left motor, and the third argument is the duty ratio for the right motor. (The zeroth argument is the name of the executable itself.) The program prepares the experiment, runs it, prints the result to the console, cleans up, and exits. The C++ code for the main function is listed below.

Main Program for the Power Dissipation Experiment

```
#include "Tank.h"
#include "SimEvents.h"
#include "TankEventListener.h"
using namespace std;

int main(int argc, char** argv)
{
   // Get the parameters for the experiment from the command line
   if (argc != 4) {
      cout << "freq left_throttle right_throttle" << endl;
      return 0;
   }
```

```
    // Get the frequency of the voltage signal from the first argument
    double freq = atof(argv[1]);
    // Create a command from the driver that contains the duty ratios from
    // the second and third arguments.
    SimPacket sim_command;
    sim_command.left_power = atof(argv[2]);
    sim_command.right_power = atof(argv[3]);
    // Create the tank, simulator, and event listener. The arguments to the
    // tank are its initial position (x = y = 0), heading (theta = 0), and
    // the smallest interval of time that will separate any two reports of
    // the tank's state (0.02 seconds).
    Tank* tank = new Tank(freq,0.0,0.0,0.0,0.02);
    Simulator* sim = new Simulator(tank);
    TankEventListener* l = new TankEventListener(tank);
    // Add an event listener to compute the power dissipated in the motors
    sim->addEventListener(l);
    // Inject the driver command into the simulation at time zero
    ModelInputBag input;
    SimEvent cmd(sim_command);
    ModelInput event(tank,cmd);
    input.insert(event);
    sim->computeNextState(input,0.0);
    // Run the simulation for 3 seconds
    while (sim->nextEventTime() <= 3.0) sim->execNextEvent();
    // Write the result to the console
    cout << freq << " " << l->getPowerLost() << endl;
    // Clean up and exit
    delete sim; delete tank; delete l;
    return 0;
}
```

Simulations are executed for a set of frequencies with the shell script

```
for ((i=50;i<=7000;i+=50)); do ./a.out \$i 0.5 0.5; done
```

where `a.out` is the name of the simulation program (this is the default name of the executable produced by the GNU C++ compiler). This script computes the power dissipated in the motors at frequencies in the range [50, 7000] at 50 Hz increments. The result is plotted in Figure 2.12. This graph suggests 3000 Hz as a reasonable lower limit for the frequency. The interactive experiments will start at 3000 Hz and proceed to higher frequencies until we discover the highest that permits effective control.[2]

[2]What happens to this lost power? It becomes heat and noise. The frequencies shown in Figure 2.12 are in the range of human hearing. Consequently, the motors emit a distinct high-pitched hum. This is accompanied by a grumbling and grinding from the gears and, if the motors are running near full power, a faint smell of ozone.

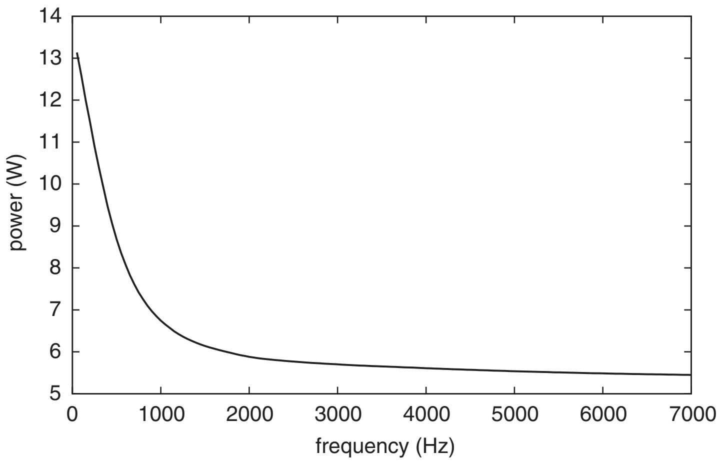


FIGURE 2.12 Total power dissipated in the motors as a function of the frequency of the voltage signal at the motor terminals.

The track shown in Figure 2.13 was used for the interactive experiments. The tank started in the center of the leftmost leg of the track, and was steered around the track to return to the starting position. If the tank left the track, that run was discarded. After several practice runs, the experimental runs were conducted at 2 kHz increments. At each frequency, the tank raced around the track until three circuits were completed. The author recorded the time to complete each circuit and the number of failed attempts at each frequency. The results are tabulated in Table 2.3.

The tradeoff between efficiency and control is immediately apparent on comparison of the data for lost power, time to complete a round of the track, and the number of failed runs. At low frequencies, the tank is very responsive and the experimental course can be safely navigated in about a minute. At higher frequencies, the driver

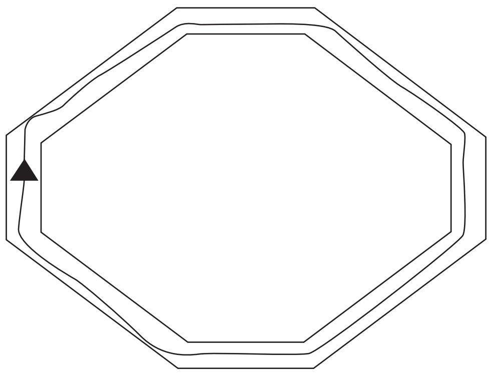


FIGURE 2.13 The test track and one path followed by the tank in a successful run.

TABLE 2.3 Data from the Interactive Experiment

	Time (s)				
Frequency (kHz)	T_1	T_2	T_3	Average	Failed
3	68.7	68.3	69.1	68.7	0
5	68.9	68.3	70.8	69.3	0
7	69.8	70.0	67.9	69.2	0
9	68.3	68.5	88.4	75.1	2
11	67.6	84.4	113.0	88.3	0
12	104.0	101.7	No data	103.2	2

must be more cautious and at 9 kHz and above the track is very difficult to negotiate. As the frequency is increased, the motors run more efficiently but the tank is more difficult to control.

From these data and the data in Figure 2.12, we can conclude that a frequency between 7 and 8 kHz is the best choice. In this range, the motors run efficiently and the computer processes commands from the driver in a timely manner. The real tank operates at about 7.4 kHz: 310 machine cycles separate invocations of the interrupt handler.

2.5 SUMMARY

This example has demonstrated the three main features of the simulation engine that is developed in the remainder of this book: modular, bottom–up construction of models, separation of the model and its simulator, and the inclusion of discrete-event and continuous components. Modular, bottom-up construction allows large simulators to be built and tested piecewise. Atomic models encapsulate basic behaviors, and if a large model is judiciously decomposed, then these smallest pieces can be built, tested, and maintained in isolation. As pieces are combined to create larger components, these, too, can be built, tested, and debugged independently of one another. This principle of encapsulation extends to the entire simulator, ultimately allowing the whole to be used as a self-contained component within a larger software system.

The separation of the model and its simulator serves a similar purpose. The algorithms contained within the simulation engine are designed for a specific class of systems. They can therefore be built, tested, and maintained without reference to any particular system. Test cases for the simulation engine will consist chiefly of simple models with behavior that can be deduced by hand calculations or with another simulator that is known to be correct. This is also an advantage for the modeler; the definition of the class of systems presumed by the simulator is a guarantee of how it will function. Improvements in the simulation engine are therefore transparent to the models, and this greatly simplifies the long-term maintenance of a simulation program.

Inclusion of discrete-event and continuous components is indispensable to modeling many engineered systems; the robotic tank is one example. This must be done accurately and with precision, but it is accuracy that presents the greatest challenge. To say that a simulation is accurate with respect to an idealized model is to say that the model's behavior can, in principle, be deduced without recourse to the simulator: there must be a correct outcome against which accuracy can be gauged. Consequently, separation of the model and the algorithms that compute its behavior is essential. These two concepts, therefore, are central to the study of modeling and simulation.