# 2.1 – R System Dynamics Tutorial 1

*Introduction to Computational Science:*
*Modeling and Simulation for the Sciences, 2nd Edition*

Angela B. Shiflet and George W. Shiflet
Wofford College


R materials by Stephen Davies, University of Mary Washington
stephen@umw.edu

## Getting familiar with R

Follow the rest of that R Computational Toolbox tutorial, *RCTTutorial1*, answering all quick-review questions as you go. That tutorial walks you through the basics of R programming, including variables, vectors, functions, control structures, and everything else you need to know to get off the ground. **The rest of this System Dynamics tutorial presupposes that you have completed that Computational Toolbox tutorial.**

## Using R for System Dynamics Problems

As described in the book, a System Dynamics problem presents itself as a complex interaction of different variables over time. Our goal in understanding such a system is to formulate a model of it and then simulate the model. The results of the simulation will give us a good idea about how the complex system will act over time when it is given certain initial conditions. Even simple looking models are often not possible to solve analytically, and hence the simulation tells us something that is not feasible to determine any other way.

There are two general approaches to creating such models and simulations. One approach is to use a specialized System Dynamics tool (such as Berkley Madonna, Vensim, or STELLA) that lets us draw diagrams of system components and then interprets our pictorial syntax to perform the simulation. The other approach is to use a general-purpose programming language (such as R, Python, or Java) that lets us write the simulation code directly, using variables, functions, and loops. There are advantages to both approaches. One advantage to using a specialized tool is that learning to draw diagrams can be a somewhat easier task for beginners than writing code. One advantage to using a general-purpose programming language is that it is more generic and flexible, giving us more freedom to control our simulation and refine it in ways that a System Dynamics tool designer might not have envisioned. We can also gain more insight into the details of the simulation, since all aspects of it are visible to us and modifiable, rather than being hidden behind the simplified diagrammatic elements a System Dynamics tool provides us with.

This tutorial is about using R — a general-purpose programming language — to write simulation programs to solve System Dynamics problems. Following this approach will give you valuable skills that enable you to go beyond what a specific tool designer may have equipped you with and to tackle a large variety of computational science problems

(System Dynamics or otherwise) in the same way. The cost for doing this is that we need to take care of some of the details ourselves that a System Dynamics tool might have done for us. It's kind of like getting the extra control and performance that a stick-shift automobile gives the driver, rather than the simplicity yet limited nature of an automatic transmission. It's a little steeper learning curve, but once you learn to drive it, you can drive this car anywhere — you can't solve a cellular automata problem or simulate a queueing system in Berkley Madonna, but you can do both and much more in R.

## Writing System Dynamics Simulation Programs: Components

The main task in writing an R simulation is to turn a system dynamics diagram (like Figure 2.5.1 in the book) into a running program. Hence this tutorial will concentrate on translating such diagrams into code.

First, some basic terminology and concepts about these system dynamics diagrams (also sometimes called relationship diagrams, or stock-and-flow diagrams):

- **stock variable.**
  These are represented on the diagram by boxes. A stock variable is typically something that accumulates as time goes on: it is a quantity that goes up or down in value, and we want to track how it changes over time. For this reason, we will normally use a vector to represent it, so that its value at each moment in time can be preserved.
- **derived stock variable.**
  These are represented on the diagram by any white circle that has an incoming arrow from a box (or from another derived stock variable). If a white circle has an incoming arrow from a box, then it is essentially another stock variable: we are interested in tracking how it accumulates or decreases over time. The only way in which it differs from a box is that its value is usually based on a simple calculation from a box. Hence, we can give our main attention to tracking the box's value as it changes throughout the simulation and then make a simple conversion of those values to obtain the derived stock variable's values. For the same reason as above, a derived stock variable will normally be a vector of values.
- **(ordinary) variable.**
  These are represented on the diagram by white circles that do not have incoming arrows from boxes (or from other derived stock variables). Each one will correspond to a scalar variable in the program. Some of these might have constant (unchanging) values, while others will represent quantities that change over time. For each of them, however, the program only needs to keep track of a single "current" value as the simulation progresses, not a whole array of values. Hence we will represent it in our simulation program as a scalar.
- **flow.**
  Finally, a flow is depicted on the diagram by a gray oval. It represents a quantity whose value will change over time, and which will influence the value of some stock variable. Though it is a scalar, it is a dynamic (changing) quantity, and hence will be continually computed in the main loop of the simulation in order to help properly calculate the values of the stock variable it influences.

**Quick Review Question 1** Consider Figure 2.5.1 of the book. Which of the symbols in

the figure correspond to stock variables? Ordinary variables? Derived stock variables? Flows?

## Writing System Dynamics Simulation Programs: Outline of Basic Procedure

The basic idea behind a computational simulation is to run through a loop a certain number of times, each time simulating one "tick" on a virtual clock. In other words, each time through the loop represents a (short) period of time, during which the important characteristics of the system (stored in program variables) may change. Often we wish to keep track of a particular variable's values over time, so that when the simulation is over we can look back and see its general trend as the simulation progressed. This is an ideal use for vectors (arrays) since they can store multiple, successive values at once.

To write an R program to simulate a system depicted in a stock-flow diagram, follow this general procedure:

1. Identify the circles with no incoming arrows at all. If a circle represents a constant (unchanging) value, then create a program variable to represent it. (Be careful of units.)

2. Identify all the circles whose only incoming arrows are from the circles already defined as variables in your program. Create a variable to represent each one, using a formula. (Be careful of units.) Repeat this step until the only circles left have something besides a circle pointing to them.

3. Create a vector (array) for each stock variable (box). Set the first value of this vector to its initial condition (i.e., its value when the simulation begins).

4. Run through the simulation loop for the specified total time and time increment. Each time through the loop:

   a. Determine the (temporary) values of any flows and remaining ordinary variable(s). Depending on the nature of the circle/oval, this could be (a) an equation based on current values of other circle(s) and/or stock variables, (b) a special value based on the current time (for instance, a "pulse" variable that takes on certain values at specific "clock ticks") or other things.

   b. Set the next value for each vector to its new value on the next clock tick.

5. For any derived stock variable that has not been defined yet, use a formula to create it based on its incoming arrows. Note that since at least one of these incoming arrows is from a stock variable (which is a vector), this circle's variable will also be a vector.

6. Finally, plot any vector(s) of interest.

## Example: Drug Dosage (Single-Dose)

Let's follow this procedure for a specific example. From Module 2.5, "Drug Dosage," read sections "Introduction" and "One-Compartment Model of Single Dose;" then look carefully at the diagram describing the drug dosage system in Figure 2.5.1. Notice that

there are six elements in this diagram. One of them — "aspirin in plasma" — is a stock variable. Three of them — "half life," "plasma volume," and "elimination constant" — are ordinary variables. The first two of these three are ordinary variables because they have no incoming arrows at all. The last of the three ("elimination constant") is also an ordinary variable because it has no incoming arrow from a box, only from another circle. One element — "elimination" — is a flow. Finally, "plasma concentration" is a derived stock variable because it has an incoming arrow from another box variable.

Translating this into an R simulation program is a combination of following the outlined procedure, and using our heads. We begin:

**1.** Identify the circles with no incoming arrows at all. If a circle represents a constant (unchanging) value, then create a program variable to represent it. (Be careful of units.)

The circles with no incoming arrows are "half life" and "plasma volume." From the problem description, we realize that both of these are simple constants: 3.2 hours, and 3000 ml, respectively. Hence, at the top of our R program, we will define these:

```
halfLife = 3.2        # hr
plasmaVolume = 3000   # ml
```

(Note that we have added small comments to the right of each line, documenting the units that the variable is in. This is a good practice to get into, since making incorrect unit conversions or assumptions is a common error.)

**2.** Identify all the circles whose only incoming arrows are from the circles already defined as program variables. Create a variable to represent each one, using a formula. (Be careful of units.) Repeat this step until the only circles left have something besides a circle pointing to them.

We have one circle whose only incoming arrow is from a previously defined variable: elimination constant. Hence, following the formula given in Equation Set 2.5.1, we write:

```
eliminationConstant = -log(0.5)/halfLife   # 1/h
```

Recall that in R, *log*() is the function to perform a natural logarithm. Note the comment "1/h" at the end of this line; this indicates that the elimination constant is in the units of "per hours."

**3.** Create a vector (array) for each stock variable (box). Set the first value of this vector to its initial condition (i.e., its value when the simulation begins.)

We have one stock variable, so we create a vector for it:

```
aspirinInPlasma = vector()
```

Then, we initialize the first value of that vector to be its value at the start of the simulation:

```
aspirinInPlasma[1] = 2 * 325 * 1000   # ug
```

These two lines of code create a vector called *aspirinInPlasma*, which will hold all the values over time as the simulation runs. Each of the values in this vector will have units of μg (micrograms).

**4.** Run through the simulation loop for the specified total time and time increment.

We need to decide two basic things about our simulation: (1) how long (in simulated time) will it run for? (2) how much simulated time will elapse between each virtual "tick of the clock?" These two choices are rather arbitrary at this point, but they affect the amount of memory your program requires as it runs, and ultimately, its speed. For now, we'll simulate 8 hours of the patient's body, and set our "granularity" to five minutes:

```
simulationHours = 8 # h
deltaX = 5/60       # h
x = seq(0,simulationHours,deltaX)
```

The variable *deltaX* (written mathematically as $\Delta x$) is our granularity, set to 5 minutes. The $x$ values in this vector mean that every iteration through our loop represents five minutes of time. Put another way, we will be recomputing the concentration of aspirin in the patient's blood every five minutes. We now write the loop itself:

```
for (i in 2:length(x)) {
```

This line needs some explanation. First, it establishes a variable called *i* whose value will change each time through the loop. It will change to be the successive values in the vector specified after the word "in," which begins at 2, and goes up to one greater than the number of clock ticks in the simulation. With the values mentioned above, this works out to be 2 through 97. This is because our simulation will run for 96 clock ticks of 5 minutes each, for a total of 480 minutes, or 8 hours. **Note that the variable *i* will take on values 2, 3, 4, ..., 97.** It will not have values $\frac{5}{60}$ h, $\frac{10}{60}$ h, $\frac{15}{60}$ h, ..., $\frac{480}{60}$ h. The latter is the job of the elements of the *x* vector, not *i* variable. *i* does not represent a time value, but simply an iteration number, and a vector index. The sequence 5, 10, 15, ... does represent the time (in minutes) corresponding to each point of the simulation: the first iteration through the loop, when $i = 2$, represents the time 5 minutes; when $i = 3$, the time is 10 minutes; and so on.

The reason we start our loop with *i* equal to 2 is that we have already set up the initial condition *aspirinInPlasma*[1] = 2 * 325 * 1000. In the loop, then, we need to begin by computing *aspirinInPlasma*[2] based on *aspirinInPlasma*[1], then *aspirinInPlasma*[3] based on *aspirinInPlasma*[2], and so forth. Clearly, we need to begin the process with element number 2.

**Quick Review Question 2** Suppose *simulationHours* had the value 40 hours and *deltaX* had the value 10/60 hours (10 minutes).

**a.** How many times would the loop body be executed?

**b.** What would the value of *i* be for each of those iterations?

**c.** What would each value of the *x* vector be?

**d.** What value of time (in minutes) would each iteration represent?

**e.** What value of time (in hours) would each iteration represent?

Incidentally, converting back-and-forth between *i* (the index value into the vector) and *x* (the actual time value) is so common and useful that it's a good idea to define a pair of functions at the top of our file to convert each way:

```
xtoi = function(x) x/deltaX + 1
itox = function(i) (i-1)*deltaX
```

This isn't strictly necessary, but it can make life easier, since there are times when we'll want to know what time value a particular vector element corresponds to, and vice versa.

Now we move on to the body of the loop.

Each time through the loop:

**a.** Determine the (temporary) values of any flows and remaining ordinary variable(s). Depending on the nature of the circle/oval, this could be (a) an equation based on current values of other circle(s) and/or stock variables, (b) a special value based on the current time (for instance, a "pulse" variable that takes on certain values at specific clock ticks) or other things.

We have only one flow: "elimination." (Recall that "plasma concentration" is a derived stock variable.) Hence, inside the body of the loop, we compute its value:

```
elimination =
    (eliminationConstant * aspirinInPlasma[i-1]) * deltaX
```

Consider this line of code carefully. It uses the formula given in Equation Set 2.5.1, with a couple of twists. First, we are computing the elimination amount at step *i* of the simulation. This requires using the amount of aspirin present in the plasma at the previous iteration, which is why we use *i-1* inside the "[]" symbols after the vector name. This is the common pattern of computing a new value based on a previous value. Second, we multiply by the time increment because each step in the iteration represents a certain amount (in our example, 5 minutes) of time. Hence, the amount of elimination that will occur during a simulated clock tick is, say, five-minutes' worth, and this must be accounted for.

**b.** Set the next value for each vector to its new value on the next clock tick.

```
aspirinInPlasma[i] = aspirinInPlasma[i-1] – elimination
```

In this first simple simulation, we are assuming only a single dose, and instantaneous absorption of the aspirin. Therefore, the only thing that affects the

amount of aspirin in the plasma each clock tick is the amount that is eliminated. Note carefully the vector indexes in this line of code. We are setting *aspirinInPlasma*[*i*] to a value based on *aspirinInPlasma*[*i*-1]. This means we are using the previous value of the aspirin in plasma (at vector position *i*-1) to compute the next value (at vector position *i*.)

We can now end our loop:

```
}
```

**5.** For any derived stock variable that has not been defined yet, use a formula to create it based on its incoming arrows. Note that since at least one of these incoming arrows is from a stock variable (which is a vector), this circle's variable will also be a vector.

At this point, we have accomplished a great deal — nearly our whole purpose. The *aspirinInPlasma* vector now contains values corresponding to the amount of aspirin in the patient's plasma that was present at five-minute increments over a period of 8 hours. All we need to do now is compute the concentration of the aspirin over that time, and plot it. First, we compute the concentration of the aspirin:

```
plasmaConcentration = aspirinInPlasma / plasmaVolume  # ug/ml
```

using the formula in Equation Set 2.5.1.

**6.** Finally, plot any vector(s) of interest.

And finally, we create a plot:

```
plot(plasmaConcentration)
```

For reference, here is the entire program:

```
halfLife = 3.2                                  # h
plasmaVolume = 3000                             # ml
eliminationConstant = -log(0.5)/halfLife        # 1/h
aspirinInPlasma = vector()
aspirinInPlasma[1] = 2 * 325 * 1000             # ug
simulationHours = 8                             # h
deltaX = 5/60                                   # h
x = seq(0,simulationHours,deltaX)
for (i in 2:length(x)) {
    elimination =
        (eliminationConstant * aspirinInPlasma[i-1]) * deltaX
    aspirinInPlasma[i] = aspirinInPlasma[i-1] - elimination
}
plasmaConcentration = aspirinInPlasma / plasmaVolume # ug/ml
plot(plasmaConcentration)
```

This program, with further comments and some additional variables defined for flexibility and good programming style, is available on the book's website (Module 2.5).

**Enhancing plots**

The plot produced by the above program is very "bare bones"; it would be more informative to add labels and appropriate values for the x and y axis ranges. We can give values to plot's main, *xlab*, and *ylab* parameters to set the text for the overall plot and for the axes. To specify the boundaries (limits) for the y axis, we can set *ylim* parameter to a 2-element vector containing our lower and upper bounds. We'll get a smooth line plot by specifying "type="l"." Finally, we'd like the x axis to show time, rather than iteration number, which we can create by using the *seq* function to create a vector from 0 to 8 (the number of hours) by $\frac{5}{60^{ths}}$ (the time increment in hours). Our final version looks like this:

```
plot(x = x,
     y = plasmaConcentration,
     type = "l",
     xlab = "hours",
     ylab = "plasma concentration (ug/ml)",
     ylim = c(0,500),  # min and max of y-axis
     main = "Aspirin concentration over time")
}
```

Another way to make a plot more informative is to identify important boundary values. R provides the function *abline()* which allows you to add a straight line (of any thickness, style, or color) to a plot. This is useful for visually indicating significant thresholds for values.

The simplest form of *abline()* takes two parameters: the *y*-intercept and the slope of the line (in that order). We can also add a parameter *col* to indicate the color of the line we want to draw. (Type "?abline" at the R prompt to get more information about the abline() function.)

The book mentions the minimum effective concentration (MEC) and the maximum therapeutic concentration (MTC) as two important values for a particular medication. For aspirin, MEC is about 150 µg/ml and MTC is about 350. We can enhance our plot with a blue line to show the MEC and a red line to show the MTC by adding the following commands after calling the *plot()* function:

```
abline(150,0,col="blue")
abline(350,0,col="red")
```

This produces the plot in Figure 1. From this graph, we can see that while the patient has thankfully not gotten close to the MTC range, his aspirin has probably become ineffective after only about 2 hours. Hence, another dose may be called for (see below.)

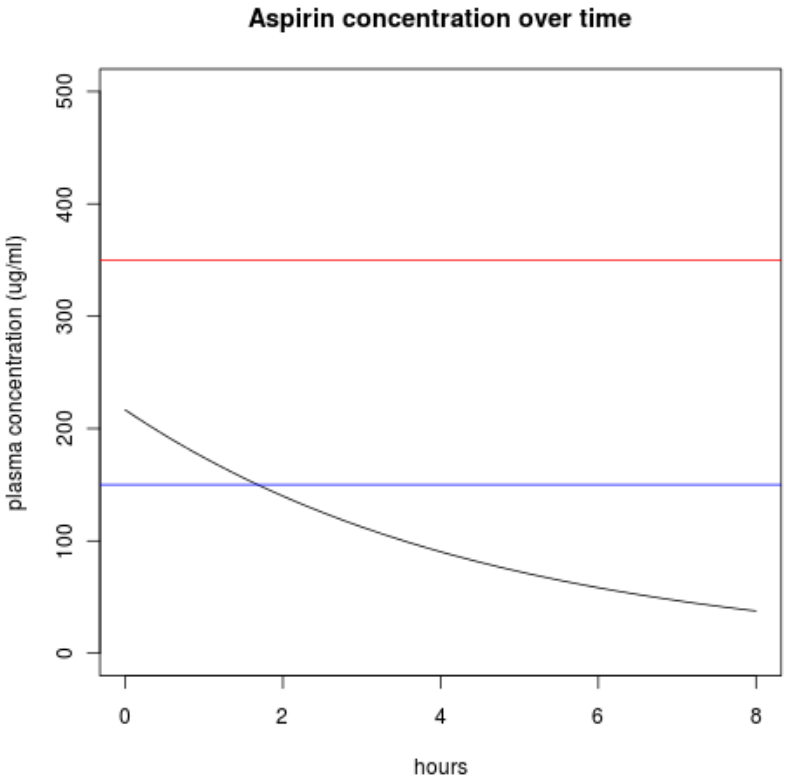


Figure 1: Plot for single-dose aspirin simulation.

**Example: Drug dosage (Repeated Doses, Module 2.5)**

We've gone through this first example in great detail. Now let's add some complexity to the model by simulating repeated doses.

Read the section "One-Compartment Model of Repeated Doses" in Module 2.5. Then, study carefully the stock-and-flow diagram in Figure 2.5.3. We will again follow the generic procedure (with less detail) to write a simulation program for this problem:

1. Identify the circles with no incoming arrows at all. If a circle represents a constant (unchanging) value, then create a program variable to represent it. (Be careful of units.)

   The circles with no incoming arrows are "MEC," "MTC," half life," "volume," "dosage," "absorption fraction," "interval," and "start." All of these are simple constants:

```
mec = 10                      # ug/ml
mtc = 20                      # ug/ml
halfLife = 22                 # h
volume = 3000                 # ml
dosage = 100 * 1000           # ug
absorptionFraction = 0.12     # (unitless)
interval = 8                  # h
start = 0                     # h
```

   It turns out that we will not need to use "start," so it is superfluous. (Our initial dosage will

automatically occur on the first clock tick.)

**2.** Identify all the circles whose only incoming arrows are from the circles already defined as program variables. Create a variable to represent each one, using a formula. (Be careful of units.) Repeat this step until the only circles left have something besides a circle pointing to them.

As before, "elimination constant" is the only relevant circle here:

```
eliminationConstant = -log(0.5)/halfLife   # 1/h
```

**3.** Create a vector (array) for each stock variable (box). Set the first value of this vector to its initial condition (i.e., its value when the simulation begins.)

We again have only one stock variable, which we create and initialize:

```
drugInSystem = vector()
drugInSystem[1] = absorptionFraction * dosage
```

Notice that we must set the first value of the vector to be the absorption fraction times the dosage, since not all of the medicine is actually absorbed into the patient's system.

**4.** Run through the simulation loop for the specified total time and time increment.

We'll run the simulation for a week at 2-minute intervals:

```
simHrs = 168        # hr
deltaX = 2/60       # hr
x = seq(0,simulationHours,deltaX)
```

These variables set us up to begin our loop. Note that we have defined a vector `x` which will contain all of the time values (in hours or parts of an hour) of the simulation. The following lines, then, actually begins the loop:

```
for (i in 2:length(x)) {
```

Each time through the loop:

**a.** Determine the (temporary) values of any flows and remaining ordinary variable(s). Depending on the nature of the circle/oval, this could be (a) an equation based on current values of other circle(s) and/or stock variables, (b) a special value based on the current time (for instance, a "pulse" variable that takes on certain values at specific clock ticks) or other things.

This is the step that is most different from the previous example. From Figure 2.5.3, we can see that there are two flows affecting how much drug is in the system: the amount ingested, and the amount eliminated. (We can see this because the stock variable has two gray lines coming into it, one from each of these quantities.) Determining how much is eliminated in each time step is the same as before. Determining the amount ingested, however, is different.

We recognize that the patient will be taking a dose of Dilantin every `interval` hours. So we need to write code that says, "if we have reached a multiple of `interval` hours, ingest

one dose during this time step. Otherwise, don't ingest anything right now." There are several ways to write this code, but the easiest is probably:

```
if (itox(i) %% interval == 0) {
    ingested = absorptionFraction * dosage
} else {
    ingested = 0
}
```

Look carefully at that if statement. It calls the *itox*() function we defined earlier, for determining what real-world time value a particular index corresponds to. It also contains the "%%" operator, which you may recall gives the remainder when performing division of integers. For instance, if you ask R to calculate "11 %% 8" it will give the answer 3, since 11 divided by 8 is 1 with a remainder of 3. Similarly, if you ask for "16 %% 8" R will respond with 0, since 8 divides 16 evenly and has no remainder.

**Quick Review Question 3** Review the meaning of the %% operator by typing "6 %% 3," "7 %% 3," "8 %% 3," and "9 %% 3" at the R console, and seeing the results.

We use this trick to determine the simulation points for the 8-hour intervals. If we take the time of each clock tick in hours (which is what *tox*(i) gives us — confirm this for yourself), and it is a multiple of exactly 8 hours, then we will set a variable called ingested to be equal to a new dose. In all other cases, we will set this variable to 0. This gives us a "pulse" effect, as desired.

**Quick Review Question 4** Suppose *deltaX* is equal to 30/60, and *interval* is equal to 12. For what values of *i* is "itox(i) %% interval" equal to 0?

Computing the amount eliminated in a time step is the same as before:

```
eliminated =
    (eliminationConstant * drugInSystem[i-1]) * deltaX
```

**b.** Set the next value for each vector to its new value on the next clock tick.

Finally, we add in the amount ingested and subtract the amount eliminated for this clock tick, and end our loop:

```
    drugInSystem[i] = drugInSystem[i-1] + ingested - eliminated
}
```

**5.** For any derived stock variable that has not been defined yet, use a formula to create it based on its incoming arrows. Note that since at least one of these incoming arrows is from a stock variable (which is a vector), this circle's variable will also be a vector.

```
concentration = drugInSystem / volume
```

**6.** Finally, plot any vector(s) of interest.

```
plot(
    x = x,
    y = concentration,
    type = "l",
```

```
    xlab = "hours",
    ylab = "concentration (ug/ml)",
    ylim = c(0,20),
    main = "Dilantin concentration over time")
abline(mec,0,col="blue")
abline(mtc,0,col="red")
```

For reference, here is the entire program:

---

```
mec = 10                    # ug/ml
mtc = 20                    # ug/ml
halfLife = 22               # h
volume = 3000               # ml
dosage = 100 * 1000         # ug
absorptionFraction = 0.12   # (unitless)
interval = 8                # hr

eliminationConstant = -log(0.5)/halfLife   # 1/h

drugInSystem = vector()     # ug
drugInSystem[1] = absorptionFraction * dosage

simHrs = 168                # h
deltaX = 2/60               # h
x = seq(0,simHrs,deltaX)

xtoi = function(x) x/deltaX + 1
itox = function(i) (i-1)*deltaX

for (i in 2:length(x)) {
    if (itox(i) %% interval == 0) {
        ingested = absorptionFraction * dosage
    } else {
        ingested = 0
    }
    eliminated =
        (eliminationConstant * drugInSystem[i-1]) * deltaX
    drugInSystem[i] = drugInSystem[i-1] + ingested - eliminated
}

concentration = drugInSystem / volume

plot(
    x = x,
    y = concentration,
    type = "l",
    ylim = c(0,25),
    xlab = "hours",
    ylab = "concentration (ug/ml)",
    main = "Dilantin concentration over time")
abline(mec,0,col="blue")
abline(mtc,0,col="red")
```

---

A slightly embellished version of this program is available on the book's website (Module 2.5).

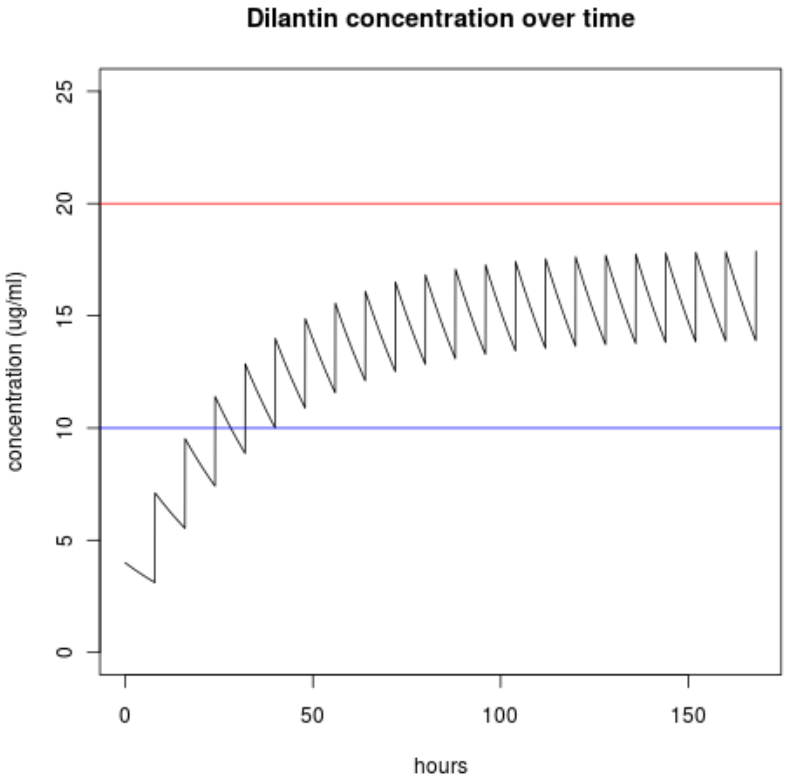


**Figure 2:** Plot for multiple-dose Dilantin simulation.

## Experimentation

One of the great values of computational simulation is the ability to modify a model's parameters and see the effect. Experiment with the repeated doses program. Download the starter code from the website (at http://wofford-ecs.org/IntroComputationalScience/dataFilePages/dataFiles/RSD.zip, save it, unzip, and run the program in R, verifying that you get a plot that looks like the one in Figure 2. Then, experiment with it in the following ways (remember that every time you make a change to the program, you must save the .R file before re-running it):

- Suppose the patient is a child, whose plasma level is only 1500 ml. Would the dosage schedule from the original file result in the child surpassing the MTC? If so, how quickly would the MTC be reached?
- If such a child took 60 mg doses in place of the 100 mg doses, would this alleviate the problem?
- Suppose this medication can only be obtained in 100 mg doses that cannot be easily divided. If the child took the drug only once per day, what effect would that have?