

Chapter 2

Simulation and Numerical Methods

Simulations are doomed to succeed.

Rodney Brooks

Robotics is, among other things, the study of complex hardware-software systems. In a complex system, many components interact, giving rise to *emergent behaviors*. These are many behaviors that would be hard to explain, anticipate, or design based on knowing the components alone. From an engineering perspective though, we would often prefer first to evaluate components of the system individually rather than the system as a whole. For instance, robotic mechanism design incorporates simulation to help select and predict the performance of a mechanism before fabrication. Similarly, software is designed to operate in harmony with hardware that provides specific interface guarantees, but we would like to evaluate software apart from the mechanisms so that we're not debugging hardware and software at the same time. During the course of software development, we would like to have assurances that software modules will meet their own specification guarantees apart from the hardware.

In robotics, we use simulators in this and many other capacities. Uses occur both offline for design and testing purposes and online for planning and control purposes. Offline simulators often (but not always) provide a visualization of the simulated system, and they also provide access to raw data and analysis. They can be interactive or can be driven forward by the simulated passage of time. In this chapter, we examine how and why simulators are used for robotics. In addition, we delve briefly into the numerical methods that make them work.

2.1 Applications of Simulation in Robotics

Simulators are found in many aspects of robotics. Offline, simulators have roles in testing and verification, design and modeling, and training people or algorithms. Online during execution, simulators are frequently used to predict the future and act appropriately.

A simulator is software that predicts the behavior of some hardware or other software. It may seem counterintuitive to simulate software with other software, but here are a few common cases where this occurs:

- the original software cannot be run in isolation from the system,
- the original software's source code is unavailable, and
- the simulator needs to run at a much faster rate than the original software, such as in predicting many possible alternate futures.

Thus, simulations of software and hardware systems alike are inevitably *approximations* of the target system.

Testing and verification. One reason we use simulators is that it is generally easier to test a robot in simulation than to test a real robot – especially if you are fielding it outside of a laboratory or classroom environment.

Testing a complex hardware/software system must be done against a specification that lays out the function of each individual component. The specification defines interfaces that dictate what inputs and outputs are appropriate to each module in each context. The value of a simulator in testing, then, is to stand in for certain parts of the system while testing other parts.

Many software bugs can be caught by a simulator that is designed to replicate the interface provided by the physical robot and approximate its behavior in response to command inputs. In addition to simulating the robot itself, we can simulate the robot's environment and how the two interact with one another. Simulators may also stand in for some parts of the robot's software.

Rodney Brooks, a famous roboticist, once quipped that software tested in simulators is “doomed to succeed” because the simulation is a simplified approximation of reality. In particular, if you have a bad assumption about how the world works, you probably encoded that assumption into both the software and the simulator, so that the error will remain hidden during testing. However, it is almost universally true that any software that fails in simulation will also fail on the real robot. Therefore, since the simulator is easier to run and examine (you can stop time, probe physical variables, etc.), it makes sense to test software thoroughly in simulation before running it on the real robot.

Modeling and design. Engineers who build robots use simulators in design tools for both hardware and software to help select parameters and anticipate how

parts (hardware or software) will interact after they are built.

Hardware design tools such as solid modeling programs incorporate several kinds of simulators. They simulate surfaces to compute how parts will interact with one another, such as gears meshing or parts rubbing. They simulate the density and volume of materials to compute centers of mass and buoyancy that can be used to determine how an object will balance forces statically or in motion. Finite element analysis is a tool for estimating the deformation of materials (strain) under an applied force due to contact (stress).

Similar tools exist for modeling and design of software. Common software simulators for offline use will provide an interface to the robot model that mimics the interface to the real robot. That model simulates how the robot hardware might respond in the context of a given environment. The model will fully simulate interactions between the robot and its environment that might lead to emergent behaviors. Thus, simulators can be used during development to study specific phenomena so that we can better understand them before developing a complete robotic system around them.

Predicting the future. Simulators can be used by robots online (during execution) to predict the future in order to anticipate how the robot and environment will act in response to control inputs. This prediction can be used to support good decision making.

One common case for online future prediction is motion planning and control algorithms. Some variants of both of these algorithms compute many possible alternative possible paths or controls that the robot could follow. In order to find out where that path or control might lead the robot, the motion is simulated forward in time. This simulation must run much faster than real time (e.g. $> 1000x$) because it is necessary to simulate many possible paths or controls before picking one to execute for real on the robot. This whole process of predicting many alternative futures and picking one repeats itself at 10 Hz or greater in order to produce responsive robot behavior. One well-known example of a control algorithm that relies heavily on simulation is *model-predictive control*.

Another application for simulated predictions are Monte Carlo methods for dealing with uncertainty. Uncertainty in the present state of a system only grows when predicting into the future, so it is necessary first to have a good model of the uncertainty in the current state before predicting it forward. Monte Carlo methods approximate a continuum, such as a probability distribution over possible locations of the robot, by taking discrete, random samples. The robot can create many hypothetical realities; it does not know which one is true, but it can assign a weight to each based on its likelihood. The samples can represent different possible positions of the robot itself or different configurations of the objects it is sensing. By combining many hypothesized predictions, the weighted average is likely to be close to

correct. A simulator can then predict how the weighted average state of the system will involve over time to achieve an improved prediction under uncertainty.

Training. One other major category of simulator is for human training. This category is familiar to us in the form of flight simulators for pilot training. When it comes to robotics, training in simulation is used to learn to teleoperate a robot. For example, an explosive ordnance disposal (EOD) robot is teleoperated by a human expert, who may use either a software simulator or a hardware simulation of a real bomb for practice. Similarly, the military pilots who operate drones might practice flying them in simulation before going on a real mission.

2.2 What Does a Simulator Provide?

All models are wrong but some are useful.

George Box

A software simulator is an imperfect copy of a real system (hardware and/or software). Most simulators use numerical approximations and are subject to floating point roundoff errors. Furthermore, in order to tractably simulate physical processes in real time or faster, the software engineers who write simulators must make simplifying assumptions when they create abstractions of the way the robot and its environment work. An abstraction is a formal description that captures the essence of the real system.

So if simulators make simplifying assumptions that cause them to be an imperfect copy of the real system, why do we use them? One reason is that they are much easier to deploy than a complete robot system, and any bug that can be isolated and fixed in simulation can probably be fixed in a small fraction of the time required to track down a bug in the field. We can catch some – but not all – software bugs using simulators.

Within the abstraction created by the simulator, we build models of the robot and its environment. These models describe how the robot behaves in response to inputs from the software or the environment (the interfaces), and how the environment provides inputs to the software through sensors. Types of models include geometric, mechanical, and behavioral. The following models are not mutually exclusive, and many simulators will employ more than one of them. Let us consider the running example of a simulator for an autonomous quadrotor drone measuring crop health.

Geometric models describe where things are and where they move to. These models can capture volumes or just the surfaces of solid objects, but there are no

interaction forces. Geometric models can tell you if two objects collide, which in the absence of forces means that the two objects' volumes intersect. Geometric models can be used to simulate contact, but this is extremely difficult because the difference in geometry between a configuration that is in contact and one that is not is an infinitesimal displacement. Thus, a slight penetration (i.e. collision) is sometimes used to model contact. Collision detection is an important topic when using geometric models to plan or avoid contact. However, geometric models do not resolve collisions realistically because energy is not conserved, causing jitter and other problems.

The *kinematic chain* of a robot arm can be modeled geometrically. In this case consecutive links in the chain are considered “non-colliding” and are permitted to penetrate one another at the location of the joints that connect the links together. In order to test whether a certain shape of the robot arm is permissible, all non-consecutive links must be collision-checked against one another and with other objects in the environment.

In our example, a geometric model of the drone would keep track of its position (latitude, longitude, altitude) and orientation as it flies. It could also detect when a collision with a ground object occurs.

Statics models track the transmission of forces among objects that are in contact but not in motion. For example, civil engineers use statics models to ensure that a bridge is self-supporting; even under external loads like wind, all the forces should cancel and there should be no net motion. In robotics, structures in contact are often connected to one another through joints that form kinematic chains. These chains can be a robot's arm or leg. A joint may exert a force in one or more degrees of freedom through a motor, whereas it resists motion in all other degrees of freedom, so that the arm or leg does not fly apart. If the forces acting on the entire kinematic chain do not balance, then motion results.

A statics model of a drone would be used by the designers to measure the strength of the airframe to resist deformation and breakage due to the forces of flight. It would also be able to help evaluate design tradeoffs like weight and rigidity or weight and battery life.

Dynamics models track the motion of objects due to the action of forces upon them. These forces can be transmitted along a kinematic chain, as in the case of a running robot's legs. Forces can also cause free flight, such as an object that has been thrown by a robot arm. The equations of motion can be especially complex for multi-link and high degree-of-freedom robots. Humanoid robots with upwards of forty degrees of freedom and no fixed connection to the ground are notably among the most complex dynamics to simulate or plan.

A dynamics simulator must model the interaction of bodies belonging to the robot and those belonging to the environment. A dynamics model generally is

unaware of the volume occupied by an object, so it will also incorporate a geometric model in order to detect collision. In addition, the dynamics model itself must model gravitation, acceleration due to motors, and dynamics effects like centripetal and Coriolis forces. Timing of events is critical, particularly with collisions since the bodies interact only during the (possibly very small) interval of time during which they make contact. Unlike in a pure geometric model, collision in a dynamics model conserves energy.

Even with a dynamics model, the simulation of contact is a tricky business. Either an object is not in contact with the ground, in which case it is in freefall, or else it is penetrating the ground. The intermediate state of being precisely in contact tends not to happen due to numerical error. A dynamics model typically resolves penetration by generating an equal and opposite force that can move the object out of penetration. This approach treats rigid objects as springs that repel each other when in collision. This repulsive force leads to freefall and re-penetration. In the steady state, some small amount of penetration causes a force that balances gravity.

A dynamics model of the drone would model and predict its motion in response to the force of the four propellers. This simulator would consider effects like centrifugal force, the viscosity of air, and transient motions like acrobatics. Compared to the kinematic model, a dynamics model is generally more accurate in its predictions.

Other **physics-based models** are used to capture effects such as electromagnetic radiation and sound that are used by a robot's sensors. These models simulate passive sensors like cameras, as well as active sensors like laser rangefinders (lidar), sonar, radar, and structured projection sensors like the Kinect.

When deciding on an altitude and position to hover while studying the crops, the drone would use a physics-based model to estimate the area of crop land covered by a 90-degree field of view camera. Such a model might also be used to estimate the accuracy of the measurement under various lighting conditions based on the contrast between sick and healthy plants.

2.3 Numerical Methods

Numerical methods compute approximate solutions to mathematical problems by discretizing continuum processes. Numerical methods are used when analytical or symbolic approaches to solving math problems are computationally difficult.

2.3.1 Integration

A definite integral $\int_a^b f(x) dx$ can be solved either analytically (i.e. symbolically) or numerically. To solve a definite integral analytically, we need to find a differentiable function $F(x)$ such that

$$F(x) = \int f(x) dx. \quad (2.1)$$

$F(x)$ is called the antiderivative, or indefinite integral, of $f(x)$. Once we have obtained $F(x)$, we can solve the definite integral using the Fundamental Theorem of Calculus:

$$\int_a^b f(x) dx = F(b) - F(a). \quad (2.2)$$

An analytic solution is very useful in that it is a general formula; we can pick any a and b and plug them into $F(x)$. However, it is sometimes computationally difficult, or even impossible, to find $F(x)$, where “impossible” means that we cannot express $F(x)$ in terms of elementary functions. In these cases, we use *numerical integration* to approximate the solution to the definite integral.

One of the most basic techniques of numerical integration is **Euler’s method**, which can be obtained from the definition of the derivative. Suppose that we know the value of $F(x)$ when $x = a$. Recall from calculus that a derivative is defined as

$$f(x) = \lim_{\Delta x \rightarrow 0} \frac{F(x + \Delta x) - F(x)}{\Delta x}. \quad (2.3)$$

Suppose that rather than taking the limit, we pick a value of Δx close to 0. Then we have

$$f(x) \approx \frac{F(x + \Delta x) - F(x)}{\Delta x} \quad (2.4)$$

for a sufficiently small Δx . If we multiply both sides by Δx and then add $F(x)$ to both sides, the result is

$$F(x + \Delta x) \approx F(x) + f(x)\Delta x. \quad (2.5)$$

What this says is that if we know $F(x)$ for some $x = x_0$, then we can approximate a nearby value $F(x_0 + \Delta x)$ by replacing the curve between $F(x_0)$ and $F(x_0 + \Delta x)$ with a straight line given by the derivative at x_0 . To estimate values that are farther away, we need to repeat this process. For example, to approximate $F(x + 2\Delta x)$, we compute

$$\begin{aligned} F(x + 2\Delta x) &\approx F(x + \Delta x) + f(x + \Delta x)\Delta x \\ &\approx F(x) + f(x)\Delta x + f(x + \Delta x)\Delta x. \end{aligned} \quad (2.6)$$

Notice how we used our approximation from (2.5) in estimating (2.6). Indeed, as we continue Euler’s method of repeated linear approximation, all of our previous estimations build on each other, resulting in a sum of approximations. Hence, to estimate $F(b)$ given the initial value $F(a)$ and our chosen value of Δx , we calculate

$$F(b) \approx F(a) + \sum_{i=0}^{n-1} f(a + i\Delta x)\Delta x \quad (2.7)$$

where $n = \lceil (b - a)/\Delta x \rceil$, the number of approximations we must take to inch our way from a to b . Thus, by the Fundamental Theorem of Calculus, we have

$$\int_a^b f(x) dx = F(b) - F(a) \approx \sum_{i=0}^{n-1} f(a + i\Delta x)\Delta x, \quad (2.8)$$

again where $n = \lceil (b - a)/\Delta x \rceil$.

The summation in (2.8) is called a **Riemann sum**. We can think of a Riemann sum as a set of adjacent rectangles whose areas sum to estimate the area under the curve (see Fig. 2.1). As $\Delta x \rightarrow 0$ (that is, as the width of the rectangles gets smaller), the approximation improves.

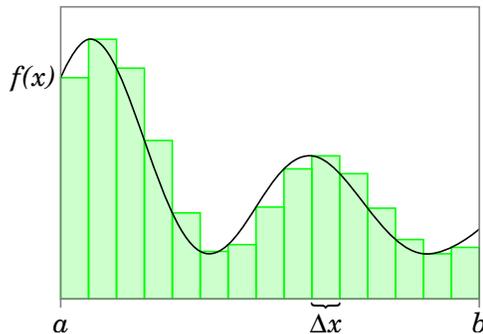


Figure 2.1: A Riemann sum estimates the area under a curve with rectangles of width Δx .

The computation of a Riemann sum is not the only method of numerical integration that appeals to the use of rectangles to estimate a curved area. In particular, the method of **Monte Carlo integration** approximates an integral by estimating the area of a rectangle whose width is the width of the interval of integration and whose height is the “average height” of the function.

What is the “average height” of a function? The Mean Value Theorem states that if a function $f(x)$ is continuous over the interval $[a, b]$, then there exists a real number c such that $a < c < b$ and

$$\int_a^b f(x) dx = (b - a)f(c). \quad (2.9)$$

In other words, the area under the curve is exactly equal to the interval width $b - a$ times the mean value, or “average height,” $f(c)$ (see Fig. 2.2).

To understand the Mean Value Theorem, imagine that the area under the curve is water in a fish tank that has just been sloshed around. When the “waves” settle down, the surface of the water will be a flat line. Then the amount of water in this planar fish tank is given by the length of the tank times the height of the water.

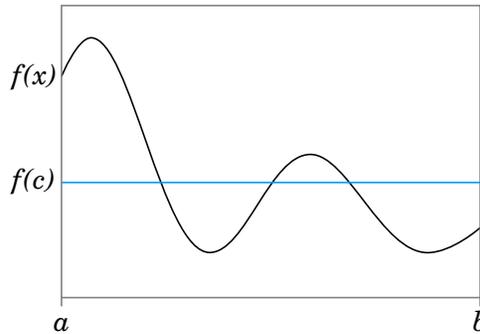


Figure 2.2: The average height of $f(x)$ is $f(c)$, so by the Mean Value Theorem, the area under $f(x)$ is exactly equal to the area under $f(c)$.

When we perform Monte Carlo integration, we approximate the mean value of the function. We do this by drawing n random numbers x_1, x_2, \dots, x_n uniformly from the interval $[a, b]$ and then taking the average of $f(x)$ evaluated at each x_i . Our approximation of the mean value is therefore

$$f(c) \approx \frac{1}{n} \sum_{i=1}^n f(x_i). \quad (2.10)$$

Then all we need to do is multiply our estimate of $f(c)$ by the width of the interval of integration. Hence,

$$\int_a^b f(x) dx \approx \frac{b-a}{n} \sum_{i=1}^n f(x_i). \quad (2.11)$$

You may recall from multivariable calculus that the concept of a function’s mean generalizes well to higher dimensions. As a result, Monte Carlo integration is most often used to approximate higher-dimensional integrals.

2.3.2 Differentiation

Consider a second-order ordinary differential equation

$$y''(t) = f(t, y(t), y'(t)), \quad a < t < b, \quad (2.12)$$

and suppose that we are given the values

$$\begin{aligned}y(a) &= y_0, \\y(b) &= y_1.\end{aligned}\tag{2.13}$$

A problem of this form is called a *boundary value problem* since it has conditions specified at the boundaries of its domain. Notice how it is related to but different from the *initial value problem* we approached with Euler’s method in (2.7) to approximate $F(b)$ given the initial value $F(a)$ and the derivative $f(x)$. A common method of numerical differentiation called **the shooting method** makes use of this relationship to estimate the solution to a boundary problem. Specifically, it takes a boundary problem and converts it into an equivalent initial value problem,

$$y''(t) = f(t, y(t), y'(t)), \quad a < t < b,\tag{2.14}$$

$$\begin{aligned}y(a) &= y_0, \\y'(a) &= v,\end{aligned}\tag{2.15}$$

and then attempts to find a root of the function

$$R(v) = f(b; v) - y_1\tag{2.16}$$

where $f(b; v)$ denotes the solution to (2.14) that uses the value v .

To gain some intuition as to why we are searching for a root of $R(v)$, imagine that you are launching a rocket with altitude given by $y(t)$ and velocity given by $y'(t)$. The boundary value problem asks,

“If I launch the rocket from altitude y_0 at time a , what initial velocity does the rocket need to reach altitude y_1 at time b ?”

Conversely, the initial value problem asks,

“If I launch the rocket from altitude y_0 at time a with initial velocity v , what altitude will the rocket reach at time b ?”

Thus, if v can be selected such that $f(b; v) = y_1$, then v will be a root of $R(v)$, and the solution $f(b; v)$ is also a solution of the boundary problem. Searching for this root is the “shooting” of the shooting method: we “shoot out” different trajectories until we find one that has the desired boundary value y_1 at time b . Various methods of root-finding can be used for this shooting process, and the method of root-finding that we choose impacts how close our approximation ends up being.

A very different but more generalizable approach to numerical differentiation is the **finite difference method**, which involves replacing derivatives with discrete

approximations and then solving the resulting discretized problem. Consider again the definition of the derivative,

$$f(x) = \lim_{\Delta x \rightarrow 0} \frac{F(x + \Delta x) - F(x)}{\Delta x}, \quad (2.17)$$

and recall that we obtained an approximation of the derivative,

$$f(x) \approx \frac{F(x + \Delta x) - F(x)}{\Delta x}, \quad (2.18)$$

by choosing Δx to be sufficiently small rather than taking the limit. The quotient in (2.18) is called a *difference quotient*, and the numerator by itself is called a *finite difference*, aptly named since it gives the difference in the value of the function between two inputs that are separated by a small but finite amount (as opposed to an infinitesimally small amount). Fig. 2.3 illustrates how the difference quotient can be used to approximate the derivative over a small interval.

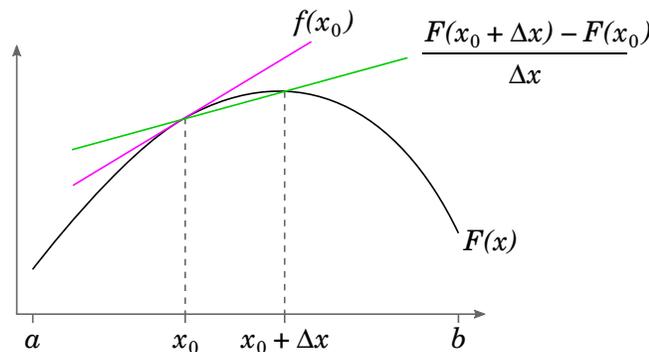


Figure 2.3: At point x_0 , the true value of the derivative is given by $f(x_0)$ (pink line) but is approximated by a difference quotient (green line).

Thus, the strategy of the finite difference method is to divide the domain into intervals of width Δx , replace the derivative at the start of each interval with the corresponding difference quotient¹, and then solve the simplified, discretized system of equations, substituting the boundary conditions in as needed.

Another approach to numerical differentiation that attempts to reduce the boundary value problem to a set of solvable equations is the **finite element method**. Although they have similar names, the finite element method is distinct from the finite difference method in that it approximates the *solution* to a differential equation

¹The difference quotients for higher-order derivatives may be obtained through the truncation of Taylor Series expansions, which you may recall from calculus.

rather than the differential equation itself. It does so with a piecewise polynomial,

$$u(x) = \sum_{i=1}^n c_i \phi_i(x), \quad (2.19)$$

where $\phi_1(x), \phi_2(x), \dots, \phi_n(x)$ is a set of functions that satisfy the boundary conditions and are piecewise continuously differentiable, and c_1, c_2, \dots, c_n is a set of yet-unknown coefficients. Solving for these coefficients gives us our approximate solution, and depending on the particular implementation of the method, it may amount to interpolation, a least squares minimization, or other fitting strategies. Since we can choose any combination of function types to constitute our set of piecewise elements (e.g. linear functions as in Fig. 2.4a or quadratic functions as in Fig. 2.4b), the finite element method is often the preferred choice for boundary value problems with complex geometries.

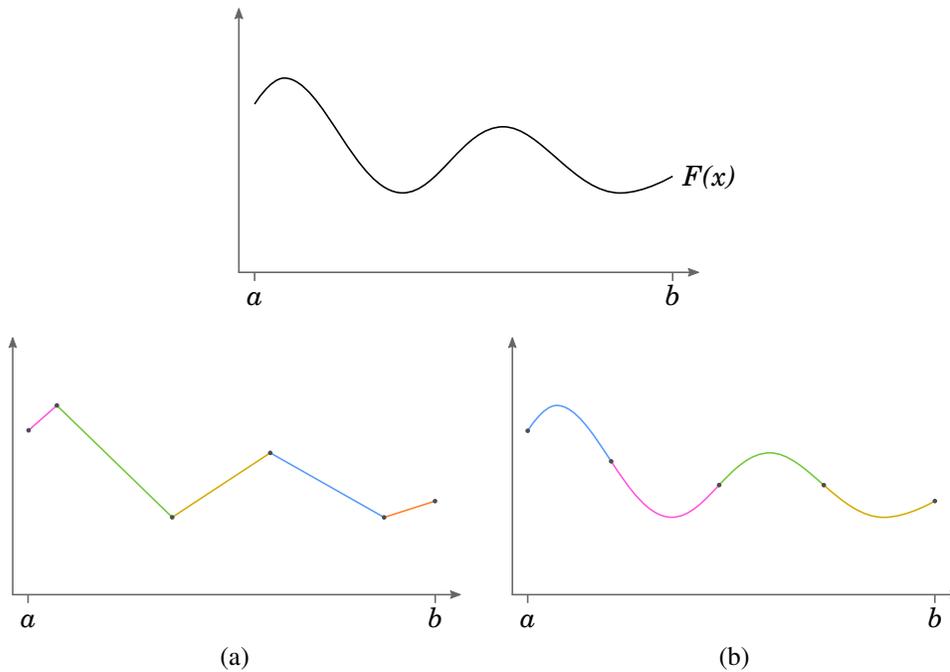


Figure 2.4: The finite element method approximates $F(x)$ with a piecewise continuously differentiable polynomial. The approximation shown in (a) uses five linear elements, and the approximation shown in (b) uses four quadratic elements (and is an exact fit).

2.3.3 Numerical Error

Numerical methods inevitably suffer from numerical error. This error comes in several forms, but the key pattern is that numerical error causes simulations to diverge from reality. The most significant source of error is discretization of time steps, iterations, and samples. The smaller the step size or the greater the number of samples, the more accurately a numerical method tends to approach reality.

However, there is another limit on accuracy of numerical methods stemming from the fact that computers use floating point numbers to compute arithmetic on real numbers as specified by IEEE 754. Floating point numbers suffer from roundoff error due to finite precision. In ordinary calculations, the roundoff error is so small (on the order of decimal 10^{-15}) that under normal circumstances you won't notice. However, we will encounter numerous situations where the roundoff error matters. To take a simple example, suppose you want to explicitly model the process of picking at random a grain of sand from a beach. One intuitively understands that the sum of the chances of picking each individual grain of sand should sum to 100%. However, the number of grains is so large that an attempt to compute the sum probably would end up much less due to roundoff error.

Another instantiation of roundoff error is more insidious, stemming from the binary representation for numbers used by computers. There is a story about a missile defense system that was deployed by the United States in the Persian Gulf war, in which the defense system had a timer that counted time in 10ths of seconds. In decimal, the representation of this number is simple: 0.1 seconds. In binary however, it is an infinite repeating number: $0.0001100110011001\dots$. A finite representation of this fraction in floating point cuts off the trailing digits at some point, making the representation slightly smaller than the actual number. Thus, the roundoff error accumulates with the ticking of the clock. After several days, the timing of the missile defense system was behind by several seconds, causing it to fail to do its job properly.

With a proper understanding of the limits of precision due to numerical error, we can design systems that minimize the consequences of this error in simulations and other numerical computations.