Chapter 3

Code Development

In this chapter we will

- discuss the development of algorithms for solving real-world problems.
- study strategies for testing, verifying, and validating algorithms.
- evaluate the uncertainties of computational models

Our main goal throughout this book will be to develop and implement accurate algorithms that will allow us to solve problems encountered in the real world. Several decades of experience have shown that the success of our endeavor is not determined only by our ability to program in any computer language. What guarantees our success is, in fact, the effort we devote in analyzing the problem before we even start typing the first lines of code as well as the time we spend verifying and validating our approach after we finish coding.

In general, the development of a successful algorithm to solve a problem of the physical world follows three basic steps (see Figure 3.1) :

From the real world to a mathematical model.—Digital computers allow us to solve problems that are too difficult to be addressed using traditional analytical techniques. However, physical systems in the real world are often too complex and interconnected even for a computer to handle. In many situations, we may be able to write, in principle, the exact equations that describe every single phenomenon and interaction in a particular physical system. Our computers, however, almost never have the capabilities to solve them all.

Consider, for example, the simulation of air currents in a room. There are approximately 10^{28} molecules of nitrogen and oxygen, as well as a trace of other elements in a typical room. The motion of each molecule can be described by six differential equations: three for the time evolution of the coordinates of each molecule and three for each component of its velocity. In total, we would need to solve simultaneously about 10^{29} equations in order to simulate completely the motion of air in the room. In order to simply store the positions and velocities of each molecule at any given time, let alone solve the corresponding equations, we would need many trillions of Petabytes of memory. This is, of course, beyond the capabilities of even the largest and fastest computers to date.

In order to solve a problem from the physical world using a digital computer, we often have to make a number of simplifications and approximations. We first

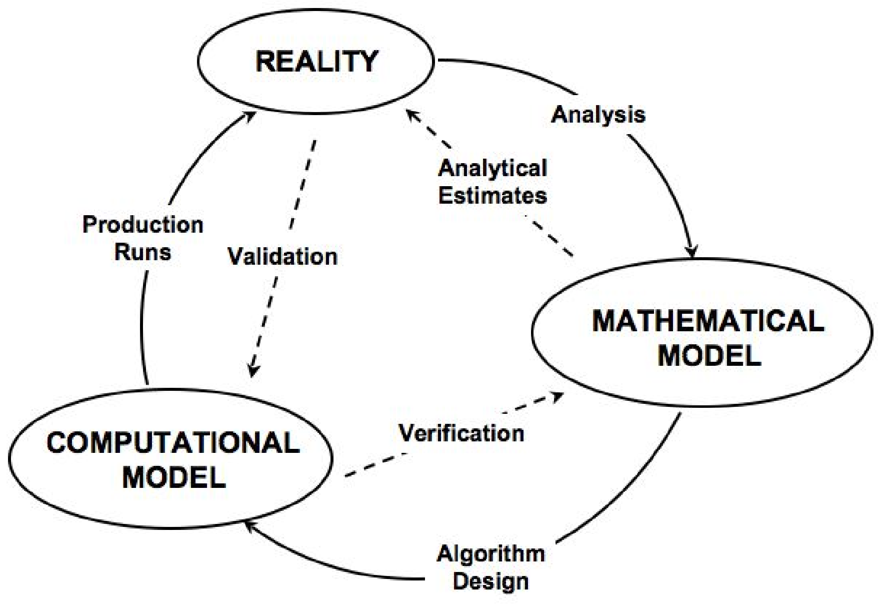


Figure 3.1: The three major steps in the development of a computational algorithm (adapted from Refs. [1] and [2]).

have to study all the phenomena that occur in the real system and, after careful analysis, identify those that are the most important in determining the evolution of the system. We often need to use statistical methods or empirical models to describe the collective effects of large ensembles of particles without having to model each particle individually. Finally, we use analytical estimates to motivate and justify our choices. Our aim in this first step is to develop an idealized mathematical model of a physical system that encompasses all the important phenomena of the real world but is not burdened by unnecessary complications.

From a mathematical to a computational model.—The equations of even an idealized mathematical model are seldom in a form suitable to be handled with a digital computer. As we discussed in the previous chapter, computers are notorious in their inability to perform calculations with very small or very large numbers such as those involved in most physical constants. Moreover, digital computers can only manipulate discrete objects and numbers and not the continuous functions that we often use in our description of physical phenomena.

Our aim in this second step is to develop an efficient technique that will allow us to handle the equations of the mathematical model with a digital computer. We identify, for example, the natural physical units of our problem in order to remove very small or very large numbers from the equations. We choose the coordinate system(s) to use, based on the symmetries of the problem or other considerations. We identify the proper method of discretizing the continuous functions that we wish to calculate over a discrete set of representative grid points as well as the particular numerical algorithm that we will use to solve the problem. We evaluate the storage requirements of the algorithm and assess various techniques of storage and data mining. Finally, we choose the digital computer as well as the computer language we will use, based on the requirements of the computational model. At this point only, we begin writing the program in order to implement the chosen algorithm on our computer.

The Birth of the Algorithm

The word algorithm derives from the name of the Persian astronomer and mathematician Muhammad ibn Musa al-Khwarizmi (∼780–850). In 825 AD, he wrote a treatise in Arabic titled *On Calculation with Hindu Numerals*. It was translated into Latin in the 12th century with the title *Algoritmi de numero Indorum* and it was one of the original books that introduced the modern-day numerals to the western world. The latin transliteration of his name and its similarity to the greek word $\alpha\rho\iota\theta\mu o s$, which means "number", is responsible for the word algorithm that soon afterwards entered the vocabulary of western mathematicians. In its modern use, it describes a set of well defined mathematical steps that, when applied to a set of initial data, generate a desired outcome.

A key ingredient of the second step of code development is the **verification** of the computational algorithm. Our goal here is to ensure that the computational model is a good representation of the idealized mathematical model. We first ensure that the coding is free from typographical or other formal programming errors. We then follow a number of techniques that push our algorithm to the limits of its validity. We devise problems with known analytic solutions, which we compare against the computational results. We verify that our algorithm satisfies conservations laws and does not break imposed symmetries on problems. Finally, we study limitations of the algorithms because of the amplification of round-off errors, discritization errors, etc.

From a computational model to reality.—Having developed a computational model, we are now in the position to use it in order to understand real physical systems or predict the outcome of phenomena in situations that we have not been explicitly observed.

The key ingredient of this last step in code development is the **validation** of the idealized mathematical and computational models. In order to check the validity of our assumptions and approximations we need to compare the results of our calculations directly to experiments. Failure of our simulations to describe a real physical system will signify an insufficient mathematical model and require for the whole process to be repeated.

Several of the steps in the sequence described above are the same whether we are addressing a particular problem using analytical or numerical means. In this chapter, we will study in some detail those steps that are specific to or especially important for solving physical problems with a digital computer. A number of review articles, cited at the end of the chapter, provide a wealth of additional information and references for future reading. The book *Code Complete: A Practical Handbook of Software Construction* by S. McConnel[3] also contains a remarkable collection of examples that illuminate various aspects of code development.

3.1 Setting up a Numerical Problem

The beauty of our physical theories lies in the fact that they involve only a small number of equations that we use to describe phenomena occurring over a vast range of scales. For example, we use Newton's laws to model both the trajectory of a baseball that leaves the hand of a pitcher as well as the motion of galaxies in the Universe. In the former case, we discuss the lengths and times involved in terms of feet and seconds, whereas in the latter case we use kiloparsecs and megayears, respectively.

Natural Units

Identifying the *natural units* with which to describe the behavior of a particular physical system is one of the key ingredients in developing a successful and efficient computational algorithm. In most cases, these are not physical units defined a priori, such as feet or kiloparsecs. Instead, they are characteristic scales in the problem that are constructed by proper combinations of the various physical quantities or of the boundary conditions that appear in the equations.

The process of identifying the natural units of a problem is specific to each particular situation. In the following two examples, as well as throughout the remaining chapters, we will explore the basic strategies we may follow and discuss their advantages.

Example: Blackbody radiation and Wien's displacement law

The spectrum of radiation that is in thermodynamic equilibrium with the walls of an insulated cavity, also known as **blackbody radiation**, was one of the key experimental results that led to the development of quantum mechanics. Contrary to the predictions of classical physics, the measured spectrum of radiation did not continue to rise towards short wavelengths but rather peaked at a wavelength that was inversersly proportional to the temperature of the cavity. This result is formally described by Wien's displacement law

$$\lambda_{\rm max} \simeq 1.97 \left(\frac{T}{{}^\circ{\rm K}}\right)^{-1} \;{\rm cm}\;. \tag{3.1}$$

Using the fact that the wavelength of light is related to its frequency ν and the speed of light c by the relation $c = \lambda\nu$, we can also express Wien's displacement law as

$$\nu_{\rm max} = \frac{c}{\lambda_{\rm max}} \simeq 5.9 \times 10^{10} \left(\frac{T}{{}^\circ{\rm K}}\right) \;{\rm Hz}\;. \tag{3.2}$$

The absence of the predicted *infrared catastrophe* was explained by Max Planck in 1901 by requiring that photons are quanta, with an energy E proportional to their frequency ν, i.e., $E = h\nu$. In this last expression, the constant $h = 6.627 \times 10^{-34}$ m^2 kgr s^{-1} is the Planck constant. The blackbody spectrum calculated by Planck is given by the expression

$$I(\nu)d\nu = \frac{2h\nu^3}{c^2}\frac{1}{\exp\left(\frac{h\nu}{k_{\rm B}T}\right)-1}d\nu\;, \tag{3.3}$$

where the specific intensity $I(\nu)d\nu$ measures the amount of energy per unit surface area, per unit time, per unit solid angle emitted in the frequency range between ν and $\nu + d\nu$. In this expression, $k_B \equiv 1.38 \times 10^{-16}$ J $^\circ$K^{-1} is Boltzmann's constant.

Figure 3.2 shows the blackbody spectrum for a few representative values of the temperature. We can easily see that, for each temperature, the spectrum of blackbody radiation indeed peaks at a particular photon frequency that increases, in general, with temperature. Our aim, in this example, is to find the photon frequency of maximum intensity and prove that it follows the empirical displacement law (3.2) obtained by Wien.

We first rewrite the expression for the blackbody spectrum in terms of its natural units. The expression has one dependent variable, the photon frequency ν, and one independent variable, the specific intensity $I(\nu)$. As a result, we will need to define natural units for the frequency and for the specific intensity. The expression also involves one parameter, the temperature T, and three physical constants, the Planck constant h, the speed of light c, and the Boltzmann constant $k_{\rm B}$.

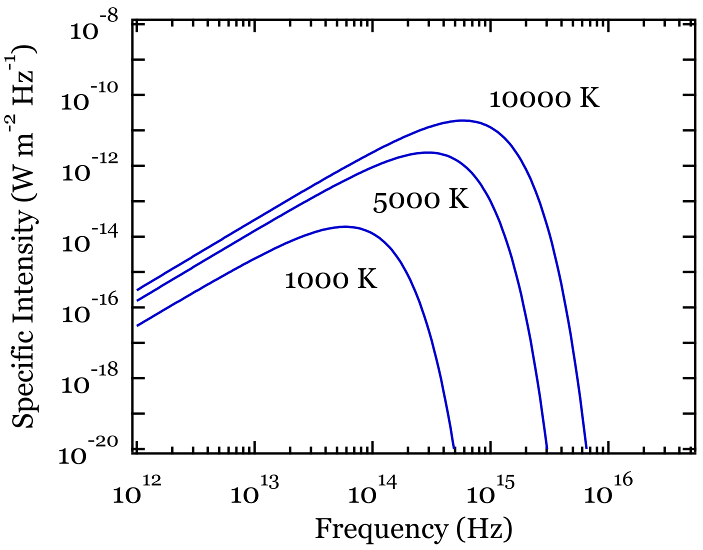


Figure 3.2: The spectrum of blackbody radiation for three different values of the temperature. The frequency of maximum specific intensity increases with temperature according to the Wien displacement law.

The table below summarizes the variables, parameters, and constants that appear in the expression for the blackbody spectrum together with their physical units. In order to facilitate the definition of characteristic scales, we have written the units of the various quantities in terms of fiducial units for mass $[M]$, length, $[L]$, time, $[T]$, and temperature, $[\Theta]$.

Physical Quantities in the Blackbody Spectrum

	Quantity	Units
Variables	ν	$[\mathrm{T}]^{-1}$
	I	$[\mathrm{M}][\mathrm{T}]^{-1}$
Parameter	T	$[\Theta]$
Constants	h	$[\mathrm{M}][\mathrm{L}]^2[\mathrm{T}]^{-1}$
	c	$[\mathrm{L}][\mathrm{T}]^{-1}$
	k_B	$[\mathrm{M}][\mathrm{L}]^2[\mathrm{T}]^{-2}[\Theta]^{-1}$

In principle, we can try various combinations of the parameter and the constants in the expression in order to obtain a natural unit for frequency and one for specific intensity. We can reduce, however, the number of trials by realizing that the arguments of exponential functions, logarithms, and trigonometric functions have to be dimensionless quanties. As a result, the ratio $h\nu/k_\mathrm{B}T$ that appears as the argument of the exponential in the denominator of the blackbody function must be a dimensionless quantity. This implies that a good natural unit for frequency in this problem is

$$\nu_0 \equiv \frac{k_\mathrm{B}T}{h} \tag{3.4}$$

Defining the dimensionless frequency $\nu' \equiv \nu/\nu_0$, we can, therefore, rewrite the expression for the blackbody spectrum as

$$I(\nu')d\nu' = \left(\frac{2k_\mathrm{B}^4T^4}{h^3c^2}\right)\frac{\nu'^3}{\exp(\nu\prime)-1}d\nu'\ , \tag{3.5}$$

Inspecting the units of the two fractions in the right-hand-side of this last expression, we also conclude that the fraction in the parenthesis has the same units as the specific intensity. We, therefore, chose for the natural unit of specific intensity the ratio

$$I_0 \equiv \frac{2k_{\rm B}^4 T^4}{h^3 c^2}\ . \tag{3.6}$$

We then define the dimensionless specific intensity $I' \equiv I/I_0$ and rewrite the expression for the blackbody spectrum as

$$I'(\nu')d\nu' = \frac{\nu'^3}{\exp(\nu\prime) - 1} d\nu'\ . \tag{3.7}$$

In order to find the dimensionless frequency $\nu'_{\rm max}$ for which the specific intensity has a maximum, we calculate the first derivative of the above expression with respect to frequency and set it to zero. The result is

$$\frac{dI'}{d\nu'}_{\nu_{\rm max}} = 0 \Rightarrow \nu'^2_{\rm max} \frac{(3-\nu'_{\rm max})\exp(\nu'_{\rm max}) - 3}{\exp(\nu'_{\rm max}) - 1} = 0\ . \tag{3.8}$$

We note that $\nu'_{\rm max} = 0$ is always a solution to the problem but is not the one we are interested in. Moreover, the presence of the denominator does not affect the solution. As a result, in order to find the frequency of the maximum intensity of the blackbody function, we only need to solve the simpler equation

$$(3-\nu'_{\rm max})\exp(\nu'_{\rm max}) - 3 = 0\ . \tag{3.9}$$

This is an equation that we cannot solve analytically and, therefore, need to employ a numerical algorithm, such as those that we will study in the following chapters. The solution is

$$\nu'_{\rm max} = 2.8214... \tag{3.10}$$

and hence the maximum of the specific intensity occurs at the frequency

$$\nu_{\rm max} \simeq 2.8 \frac{k_{\rm B} T}{h} \simeq 5.9 \times 10^{10} \left(\frac{T}{{}^\circ{\rm K}}\right)\ {\rm Hz}\ , \tag{3.11}$$

which is nothing but Wien's displacement law.

We cannot overemphasize here the importance of recasting our problem in dimensionless variables and of simplifying the final equation we need to solve. In the previous example, this procedure offered us three important benefits, even though it required considerable effort before we even sat in front of the computer,

First, the final equation (3.9) that we had to solve numerically was simple and involved only coefficients of order unity. All the very small or very large numbers of the original expression, such as the Planck constant and the speed of light, disappeared. This guaranteed that, for all practical purposes, the final solution would also be of order unity. Moreover, it minimized the possibility that our algorithm would straggle with underflows, overflows, or rounding errors, although we could never be sure of the latter; unfortunately, it is often that case that an apparently benign equation has hidden cancellations and divisions by zero!

Second, even though our original aim was to calculate the frequency of maximum intensity as a function of temperature, our final task involved the solution of only a single algebraic equation. This significantly reduced the computational resources required to solve the problem.

Finally, we effectively proved that the frequency of maximum intensity has to be proportional to temperature, as required by the Wien displacement law, before

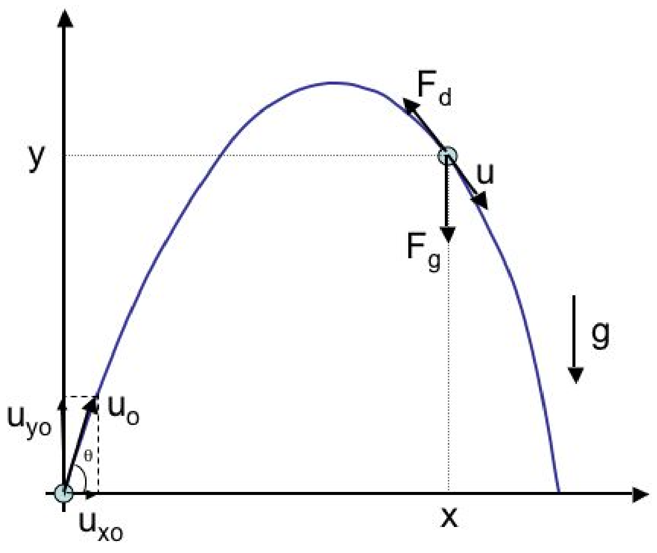


Figure 3.3: The set up for describing the motion of a projectile in a uniform gravitational field.

even solving numerically the last equation. We achieved this by defining the dimensionless frequency in terms of the temperature of the blackbody (see equation 3.4) and by eliminating the later from the final expression. Indeed, since the final equation (3.9) does not involve any physical constants or variables, its solution will be a single real number. Therefore, when dimensions are restored, the frequency of the maximum in the blackbody function will have to be proportional to temperature. As is often the case, we understood the physics of the problem by working with paper and pencil. We then simply used the computer to calculate the exact value of a constant that we, nevertheless, expected to be of order unity!

Example: Projectiles in a uniform gravitational field

As a second example, we will set up the calculation of the trajectory of a projectile launched from the Earth's surface.

Assumptions and empirical models.—We will consider only relatively short trajectories, so that we can assume that the gravitational field is uniform, and neglect the effects of the Earth's rotation. We will also employ an empirical model for the aerodynamic drag the projectile experiences during its motion through the air. In this model the aerodynamic drag force is antiparallel to the velocity vector of the projectile and its magnitude is proportional to the square of the velocity, i.e.,

$$\vec{F}_{\rm d} = -b\,|\vec{u}|\,\vec{u}\ . \tag{3.12}$$

The coefficient b depends on the shape and size of the projectile, as well as on the density of air, and its physical units are $[\mathrm{M}][\mathrm{L}]^{-1}$. Experiments have shown that this model describes accurately the aerodynamic drag experienced by a subsonic projectile.

Equations and coordinate system.— The motion of the projectile is affected by two forces: gravity that acts along the vertical direction and aerodynamic drag that acts along the instantaneous velocity of the particle. As a result, the trajectory of the projectile will always be on the plane that is defined by the vertical direction and the initial velocity vector. We will solve our problem on that plane, on which we set a Cartesian coordinate system with the x axis along the horizontal direction, the y axis along the vertical direction, and the zero point of the coordinate system at the initial position of the projectile.

The differential equation that describes the motion of the particle is given by Newton's second law. In vector form, we can write this equation as

$$\begin{aligned} m\frac{d^2\vec{r}}{dt^2} &= \vec{F}_{\rm g} + \vec{F}_{\rm d} \\ &= -m\vec{g} - b\,|\vec{u}|\,\vec{u}\ . \end{aligned} \tag{3.13}$$

Here $\vec{g}$ is the gravitational acceleration, m is the mass of the particle, $\vec{r} = x\hat{x} + y\hat{y}$ is the position vector of the projectile, (x, y) are its Cartesian coordinates, and $\hat{x}$ and $\hat{y}$ are the unit vectors along the $x-$ and $y-$directions.

Expressing the gravitational acceleration and velocity vectors in terms of their coordinates as $\vec{g} = -g\hat{y}$ and $\vec{u} = u_x\hat{x} + u_y\hat{y}$, the differential equations for the motion become

$$\frac{d^2x}{dt^2} = -\frac{b}{m}\left(u_x^2 + u_y^2\right)^{1/2} u_x \tag{3.14}$$

$$\frac{d^2y}{dt^2} = -g - \frac{b}{m}\left(u_x^2 + u_y^2\right)^{1/2} u_y\ . \tag{3.15}$$

Here we have used the Pythagorean theorem to express the magnitude of the velocity vector in terms of its components as

$$|\vec{v}| = \left(u_x^2 + u_y^2\right)^{1/2}\ . \tag{3.16}$$

The differential equations (3.14) and (3.15) are second-order in time and, therefore, require four initial conditions: the two components of the position and velocity vectors. By construction, the initial position of the projectile is at the origin of the coordinate system and hence we set

$$x(t=0) = 0 \tag{3.17}$$

$$y(t=0) = 0\ . \tag{3.18}$$

We finally allow for the two components of the initial velocity to have arbitrary values, i.e.,

$$u_x(t=0) = u_{x0} \tag{3.19}$$

$$u_y(t=0) = u_{y0}\ . \tag{3.20}$$

The differential equations (3.14) and (3.15) together with the boundary conditions (3.17)–(3.20) uniquely specify the motion of the projectile.

Natural units and dimensionless equations.—In order to identify the natural units for this problem, we begin by considering the units of the independent and dependent variables, as well as of the initial conditions and the parameters that appear in the equations. In this problem, there is one independent variable, the time t with units of time, and two dependent variables, the coordinates x and y, with units of length. We, therefore, need to identify a natural unit for time and one for length.

The table shown in the following page summarizes the various physical quantities in the problem and their units. Using this set of quantities, there are two possibilities of defining a natural unit of time, since both the quantities $m/(bu_{y0})$ and u_{y0}/g have units of time. (Note that, in these equations, we could have used the initial velocity

Physical Quantities for Calculating Projectile Trajectories

	Quantity	Units
Variables	t	[T]
	x	[L]
	y	[L]
Initial Conditions	u_{x0}	$[L][T]^{-1}$
	u_{y0}	$[L][T]^{-1}$
Parameters	m	[M]
	b	$[M][L]^{-1}$
	g	$[L][T]^{-2}$

in the x-direction, as well.) In order to choose among the two possibilities, we will use our understanding of the physics of the problem. The timescale defined by the quantity $m/(bu_{y0})$ is the characteristic time for the air resistance to slow down the projectile. On the other hand, the timescale defined by the quantity u_{y0}/g is the characteristic time for gravity to accelerate or decelerate the projectile. For most applications, gravity is the dominant force and the latter timescale is the fastest. We, therefore, choose as a unit of time the quantity

$$t_0 \equiv \frac{u_{y0}}{g} . \tag{3.21}$$

It is then natural that we identify the quantity

$$l_0 \equiv \frac{u_{y0}^2}{g} \tag{3.22}$$

as our unit of length and the quantity

$$u_0 \equiv u_{y0} \tag{3.23}$$

as the unit of velocity.

Armed with the identification of the natural units for this problem we may now define the set of dimensionless quantities

$$\begin{aligned} t' &\equiv t/t_0 & (3.24)\\ x' &\equiv x/l_0 & (3.25)\\ y' &\equiv y/l_0 & (3.26)\\ u'_x &\equiv u_x/u_0 & (3.27)\\ u'_y &\equiv u_y/u_0 & (3.28) \end{aligned}$$

and convert the system of equations (3.14)–(3.15) to

$$\begin{aligned} \frac{d^2x'}{dt'^2} &= -A\left(u_x'^2 + u_y'^2\right)^{1/2} u'_x & (3.29)\\ \frac{d^2y'}{dt'^2} &= -1 - A\left(u_x'^2 + u_y'^2\right)^{1/2} u'_y . & (3.30) \end{aligned}$$

In these equations we have used a new dimensionless parameter that we defined as

$$A \equiv \frac{bu_{y0}^2}{mg} . \tag{3.31}$$

This parameter measures the ratio of the characteristic aerodynamic drag force to the gravitational force exerted on the particle. When $A = 0$, the trajectory of the projectile is unaffected by aerodynamic drag. As the value of the parameter A increases, the relative importance of the drag force increases as well.

The dimensionless initial conditions that are required to specify uniquely the trajectory of the particle are

$$\begin{aligned} x'(t=0) = y'(t=0) &= 0 && (3.32)\\ u'_x(t=0) &= \frac{u_{x0}}{u_{y0}} \equiv \cot\theta && (3.33)\\ u'_y(t=0) &= \frac{u_{y0}}{u_{y0}} = 1\ . && (3.34) \end{aligned}$$

These are the equations that we will need to solve with one of the numerical techniques for handling ordinary differential equations that we will discuss in a later chapter.

The above example illustrates again the advantages of properly setting up a numerical method and of identifying the natural units of a physical problem. The solution to the initial set of equations required the specification of four initial conditions, i.e., the two components of the initial position and velocity vectors, and of three parameters: the coefficient b of the aerodynamic drag force, the mass m of the projectile, and the local gravitational acceleration g of the location from which the projectile is launched. Studying the trajectory of the projectile under all possible conditions would, therefore, require exploring a seven-dimensional parameter space, which is a demanding computational task. Indeed, assuming, as an example, that we would like to perform a calculation for 10 representative values for each of the seven parameters and initial conditions would require that we complete a total of 10^7 calculations.

On the other hand, by properly aligning the axes of the coordinate system and by identifying the natural units of the problem, the final set of equations depends only on the single dimensionless parameter A and on one boundary condition (equation 3.33), which is related to the angle θ between the initial velocity vector and the horizontal axis. As a result, we can investigate the complete behavior of the same physical system by exploring only a two-dimensional parameter space. Assuming again that we would like to perform a calculation for 10 representative values of the parameter and of the initial condition would require that we complete a total of only 100 calculations. The proper numerical set up has, therefore, reduced the total number of computations by a factor as large as 10^5!

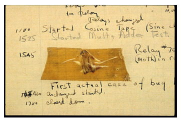
3.1.1 Code Verification

In the early days of computer programming it was often said that "a computer code that runs correctly on the first trial is so trivial that is not worth writing". Indeed, it is practically impossible to develop a computer program without introducing inadvertently a multitude of errors. Moreover, there are always undocumented pitfalls in every numerical method that we use, which we usually find by trial and error. The process in which we identify and correct errors in our programming is called **code verification**.

A coding error, which is commonly referred to as a **bug**, may be something as simple as a typographical or a bookkeeping mistake. Using variable names that are very similar to each other or not being careful with the use of indices in arrays are among the most common sources of such errors. Several studies of commercial software have found that a fraction as large as $18-36\%$ of all coding errors can be traced to typographical or bookkeeping mistakes[3].

The First Real Computer Bug

In modern computer jargon, a bug is a mistake introduced inadvertently in an algorithm that prevents it from completing its execution or from providing accurate results. The process of correcting an algorithm from such mistakes is called *debugging*. Although this term is used figuratively today, the operation of early computers was often affected by insects flying between their vacuum tubes! The first "real" computer bug was found in 1947, in the computer Mark II at Harvard University. Computer engineers taped the bug in their logbook, which can be found today at the Smithsonian Museum of American History.



In the opposite extreme, a coding bug might arise from a complex logical error that appears only when a very particular set of conditions exists. For example, several techniques for solving numerically algebraic equations of the form $f(x) = 0$ fail when the function $f(x)$ has a local maximum or minimum in the domain of solution. These logical mistakes are typically the hardest to identify and correct.

Although good programming practices reduce significantly the number of coding bugs in a computer program, it is practically impossible for any amount of verification to lead to an algorithm that is completely debugged. Throughout the computer industry, an average of 1 to 25 bugs per 1000 lines of code remain unidentified even after the software has been extensively tested and released for public use[3]. In computational physics and engineering, a very similar rate of serious faults can be found in computer codes that are widely used (see box in the following page). The moral of the story is that code verification has to be a continuous process that does not stop even after the code has been used in producing results.

A second important lesson in code verification learned after decades of experience in the computer industry is that no single method of debugging is sufficient to detect the vast majority of defects in any given piece of computer software. Techniques that range from informal code reviews between co-workers all the way to extensive testing by sample users (often called *beta testing*) are capable of detecting only as much as $\sim 40\%$ of coding bugs[4]. It is only when a combination of different techniques is used that as much as 95% of coding bugs have been identified in some of the most successful software packages.

There exist a large number of strategies developed and books written on ways of testing efficiently computer codes. The classic monograph *The Art of Software Testing* by G. J. Myers[5] as well as more recent books[3,6] offer a wide range of suggestions and case studies that are relevant to general computer algorithms. In the remaining of this chapter, as well as throughout this book, we will consider only those aspects of verification that are particular to computational physics and engineering[1,7,8].

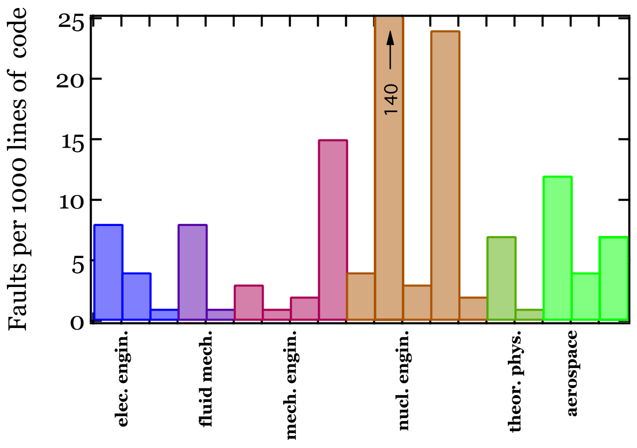
3.1.2 Validation

Further Reading

1. *Terminology for Model Credibility*, by S. Schlesinger et al., Simulation, 32, 103 (1979)

No code is perfect; some are not nearly correct!

In 1997, an experiment was conducted that aimed to investigate the number of serious faults found in mature computer codes used in a very large number of disciplines, from medical imaging to aerospace[4]. More than 100 computer packages were analyzed with a combined total of more than 5 million lines of computer code in C and in Fortran. All of these packages were considered to have been fully tested by their developers.
A static analysis was performed on each package and every identified fault was given an appropriate weight, depending on its severity. The following figure, which was adopted from the study, shows that 1 to 25 serious faults per 1000 lines of code are commonplace in computational physics and engineering. One particularly disturbing result involves a package in nuclear engineering, which reached 140 faults per 1000 lines of code. As the author of the study remarked, this code "in spite of the aspirations of its designers, amounted to no more than a very expensive random number generator".



2. *Verification, Validation, and Predictive Capability in Computational Engineering and Physics*, by W. L. Oberkampf, T. G. Trucano, & C. Hirsch, Sandia National Laboratory Report SAND2003-3769 (2003).

3. *CODE Complete: A Practical Handbook of Software Construction*, by S. McConnel (Microsoft Press), 2nd ed., (2004)

4. *The T-Experiments: Errors in Scientific Software*, by L. Hatton, IEEE Comp. Science and Engineering, 4, 2, 27 (1997).

5. *The Art of Software Testing*, by G. J. Myers (Wiley) 1979.

6. *Testing Computer Software*, by C. Kaner, J. Falk, & H. Q. Nguyen (Wiley) 2nd ed. (1999).

7. *Verification and Validation in Computational Science and Engineering*, by P. J. Roache (Hermosa Pubs., 1998)

8. *Review of Code and Solution Verification Procedures for Computational Simulations*, by C. J. Roy, Journal of Comp. Physics, 205, 131 (2005).

9. *Quantification of Uncertainty in Computational Fluid Dynamics*, by P. J. Roache, Annual Rev. of Fluid Mech., 29, 123 (1997).

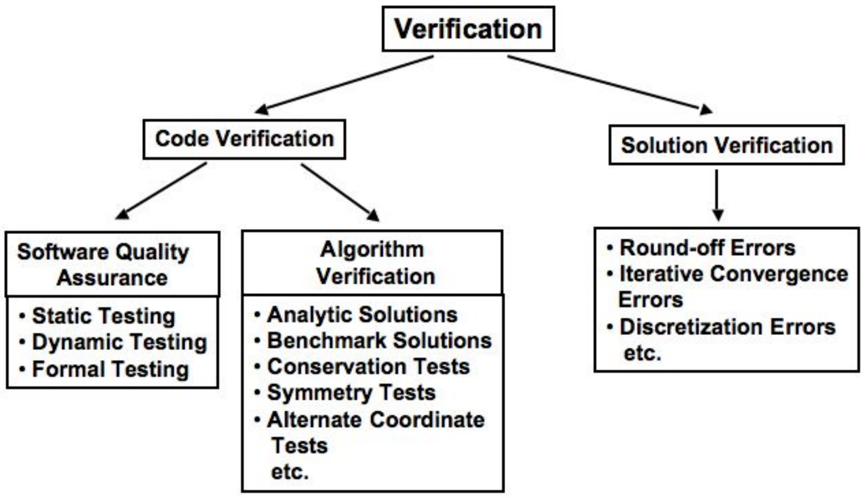


Figure 3.4: The various steps of verifying a computational physics algorithm (based on Refs. [2]-[5]).