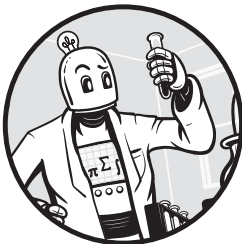


9

PHYSICS

Physics is not a religion. If it were, we'd have a much easier time raising money.
—Leon M. Lederman



Julia is a superb platform for physics calculations of all kinds. Various features of its syntax, such as the ability to use mathematical symbols and its concise array operations, make it a natural fit for programming algorithms that we use in physics. Julia's speed of execution makes it one of only a few languages used for the most demanding large-scale simulations (and the others in this club are all lower-level, statically compiled languages). Julia's physics ecosystem includes some state-of-the-art packages. Finally, Julia's unique ability to mix and match functions and data types from disparate packages to create new capabilities is especially powerful in physics calculations, as we'll see in detail in this chapter.

We begin with an introduction to two packages of general utility for dealing with units and errors. Both of these are potentially helpful in any

physics project. We'll spend some time in the first section looking into various options for producing publication-quality plots including typeset units in axis labels. Then we'll turn to specific calculations, first using a package for fluid dynamics and then using a general-purpose differential equation solver. See “Further Reading” on page 304 for each major package's URL.

Bringing Physical Units into the Computer with Unitful

The traditional way to perform physics calculations on a computer is to represent physical quantities as floating-point numbers, subject those numbers to a long series of arithmetic operations, and then interpret the results again as physical quantities. Since physical quantities are usually not simply numbers, but have *dimensions*, we need to manually keep track of the *units* that are associated with these quantities, often with code comments to remind us what the various units are.

NOTE

A dimension is a fundamental physical idea encompassing something that can be measured, such as mass or time. A unit is a specific way of measuring a dimension. The dimensions are universal, but there are various systems of units. For example, for the dimension of length, some common units are centimeter (cm), meter (m), or, if we live in the United States, inches or football fields.

In other words, the physical meanings of the numbers appearing in a program are not part of the quantities themselves, but are implicit. It may not be surprising that this can lead to confusion and errors. In 1999, NASA lost a spacecraft because two different contractors were contributing to the design, and their engineering programs used different systems of units.

In traditional languages for physics, such as Fortran, not much can be done about this issue directly. In Julia, because of its sophisticated type system, we are not limited to collections of dimensionless numbers; we can calculate with richer objects including units.

After importing the Unitful package, we can refer to many common physics units using a nonstandard string literal (see “Nonstandard String Literals” on page 128) with the prefix `u`:

```
julia> using Unitful
```

```
julia> u"1m" + u"1cm"
101//100 m
```

```
julia> u"1.0m" + u"1cm"
1.01 m
```

```
julia> u"1.0m/1s"
1.0 m s^-1
```

Here we add a meter and a centimeter, and receive the result as a rational number of meters. The package returns results as rational numbers, when

possible, to preserve the ability to carry out exact conversions. But, as the second example shows, we can coerce a floating-point result by supplying a floating-point coefficient. The third example shows how we can construct expressions within the string literal.

You can find the complete list of units only in the source code, in its GitHub repository at `src/pkgdefaults.jl`, but most of them follow the usual physics conventions. Using the string literal syntax each time we want to refer to a unit can be cumbersome, so we can assign units to our own variables to ease our typing and make the code easier to read:

```
julia> m = u"m";
```

```
julia> 1m + u"1km"
1001 m
```

We add a meter to a kilometer, showing how we can use custom variables in combination with the string literals. The result is 1,001 meters.

We can parse a string as a `Unitful` expression with another function provided by the package (undocumented at the time of writing):

```
julia> earth_accel = "9.8m/s^2";
```

```
julia> kg_weight_earth = uparse("kg * " * earth_accel)
9.8 kg m s^-2
```

Here we use `uparse()` to convert a string, created by concatenating a string representing a mass with another representing the gravitational acceleration near the surface of Earth, into a unit expression representing the mass's weight. The forms in which unit expressions appear in the REPL are not themselves legal strings for converting with `uconvert()`. For example, we need to include the multiplication operator in the string in the second line.

Using Unitful Types

We can gain access to a large supply of standard SI units by importing the `DefaultSymbols` submodule rather than defining them one by one. This practice adds a profusion of names to our namespace, however, so it may not be a good idea if we're using only a few units:

```
julia> using Unitful.DefaultSymbols
```

```
julia> minute = u"minute"
```

```
julia> 2s + 1minute
62 s
```

Here we add 2 seconds to 1 minute, resulting in 62 seconds. The `DefaultSymbols` submodule supplies the `s` unit, but we need to define `minute`, as that's not an SI unit. We're using Julia's syntax for multiplication through juxtaposition; this

expression is the same as $2 * s + 1\text{minute}$. However, these variables must be attached to numerical coefficients in arithmetic expressions; $2 * s + \text{minute}$ is a `MethodError`.

We can find the reason for this error in the types of the two expressions:

```
julia> typeof(1minute)
Quantity{Int64, T, Unitful.FreeUnits{(minute,), T, nothing}}
```

```
julia> typeof(minute)
Unitful.FreeUnits{(minute,), T, nothing}
```

The type of `1minute`, which is the same as the type of $1 * \text{minute}$, is a `Quantity`, while the type of `minute` is a `FreeUnits`. Both of these types are defined in the package. The `Unitful` package defines methods for addition and other arithmetic operations that accept arguments of type `Quantity`, but not of type `FreeUnits`.

These types contain parameters appearing as boldface Unicode characters. The `Unitful` package uses these characters to represent dimensions, so these type specifications tell us that the `minute` unit has dimensions of time, represented by `T`.

The type of `minute` and other units is an abstract type (see “The Type Hierarchy” on page 222), while the types of quantified units such as `1minute` are concrete. For good performance, we should calculate with concrete types and define our own types with fields that have concrete types only.

Stripping and Converting Units

Sometimes we need to remove the units from the result of a calculation—for example, when passing a result to a function that doesn’t understand units. We can do this with the `convert()` function:

```
julia> convert(Float64, u"1m/100cm")
1.0
```

The type of the result is `Float64`. The results returned by `Unitful` calculations may not always be what we expect, so we should use `convert()` when we require a simple number:

```
julia> u"1m / 100cm"
0.01 m cm^-1

julia> typeof(u"1m/100cm")
Quantity{Float64, NoDims, Unitful.FreeUnits{(cm^-1, m), NoDims, nothing}}
```

Here we divide a length by another length, so the result should be the simple number 1.0 (because the lengths are equal) with no dimensions. The actual result is equivalent to that, but it’s expressed in an obscure form. Checking the type of the result, we find that it’s the concrete `Unitful` type `Quantity`, with type parameters indicating that it has no dimensions.

If we use the same literal unit in the numerator and denominator, we get a result that may be closer to what we expect:

```
julia> u"1m / 2m"
0.5
```

```
julia> typeof(u"1m / 2m")
Float64
```

A further example shows that Unitful is consistent in retaining the units we use in expressions instead of making conversions that might seem obvious to a physicist:

```
julia> u"1m * 1m"
1 m^2
```

```
julia> u"1m * 100cm"
100 cm m
```

The two input expressions mean the same thing, but lead to equivalent results that are expressed differently.

The function `upreferred()` from Unitful converts expressions so they use a standard set of units. The user can establish preferred systems of units, but the default behavior uses conventional SI units:

```
julia> u"1m * 100cm" |> upreferred
1//1 m^2
```

In addition to converting to a number with `convert()`, we can use `uconvert()`, which is part of Unitful, to convert between units:

```
julia> uconvert(u"J", u"1erg")
1//10000000 J
```

```
julia> uconvert(u"kg", u"2slug")
29.187805874412728 kg
```

The function takes the unit to convert to in its first argument and the expression to convert in its second argument. In the first example we convert from ergs to joules. As both are metric units related by an exact ratio, `uconvert()` supplies the answer using a rational coefficient. The second example is a conversion from the US unit of mass, slugs, to kilograms, the standard SI unit used in physics. The conversion factor is a floating-point number.

Listing 9-1 shows another way to extract the purely numerical part of a Unitful expression with `ustrip()`.

```
julia> vi = 17u"m/s"
17 m s^-1
```

```
julia> vf = 17.0u"m/s"
17.0 m s^-1
```

```
julia> ustrip(v), ustrip(vf)
(17, 17.0)
```

Listing 9-1: Stripping units with `ustrip()`

The `ustrip()` function preserves the numerical type in the expression.

To extract just the unit from a `Unitful` expression, the package provides the `unit()` function, as shown in Listing 9-2.

```
julia> unit(vi)
m s^-1
```

Listing 9-2: Extracting units with `unit()`

We'll find applications for `ustrip()` and `unit()` in “Plotting with Units” on page 276.

Typesetting Units

Using the `UnitfulLatexify` package, we can turn our `Unitful` expressions into LaTeX-typeset mathematics: either as LaTeX source ready to be dropped into a research paper or as a rendered image. Here is a simple example:

```
julia> using Unitful, Latexify, UnitfulLatexify

julia> 9.8u"m/s^2" |> latexify
L"$9.8\;\mathrm{m}\,\mathrm{s}^{-2}$"
```

The `latexify()` function transforms the `Unitful` expression for Earth's gravitational acceleration into a LaTeX string. We encountered LaTeX strings in Listing 4-1, when we used one to generate a title for a graph. The `UnitfulLatexify` package combines the LaTeX abilities in `Latexify` with `Unitful`, which is why we need to import all three packages, as we did at the start of this example.

When used in the REPL or another nongraphical context, `latexify()` produces LaTeX markup ready to be copied and pasted into a document. We can, instead, create a PDF image of the result by passing it to the `render()` function. To do that, you need to have the external program `luaLaTeX`, which is part of standard LaTeX installations, installed. If that program is available, `render()` will use it to typeset the LaTeX string and immediately display it with the default PDF viewer. The `render()` process litters your temporary directory with files for every rendered expression, which is something to keep an eye on.

When using `UnitfulLatexify` in a graphical environment, such as a Pluto notebook, the output is rendered as LaTeX rather than LaTeX source. In most environments, typesetting uses a built-in engine rather than an external program, so no additional installations are required. For example, Pluto uses `MathJax`, a JavaScript library for LaTeX mathematical typesetting.

Figure 9-1 shows a Pluto session with Newton's Second Law of Motion.

```

• using Unitful ✓, Latexify ✓, UnitfulLatexify ✓

F = 3.5 N
• F = 3.5u"N"

m = 63.1 kg
• m = 63.1u"kg"

a = 0.0554675118858954 N kg^-1
• a =  $\frac{F}{m}$ 

0.05547 m s-2
• latexify(uconvert(u"m * s^-2", a))

```

Figure 9-1: Using `UnitfulLatexify` in Pluto

In the final cell in Figure 9-1, we convert the acceleration to a more conventional combination of units and pass the result to `latexify()`. The typeset version appears as the result. MathJax provides a contextual menu when right-clicking on the result that gives us access to the LaTeX source.

If the use of negative exponents in unit expressions is not to our taste, we can pass the `permode` keyword to tell `latexify()` to use other styles. Here's an example that demonstrates the default and the two options for `permode`:

```

julia> a = 0.0571u"m/s^2"

julia> """
a = $(latexify(a))

or

$(latexify(a; permode=:frac))

or

$(latexify(a; permode=:slash))

""" |> println
a = $0.0571\;\mathrm{m}\,\mathrm{s}^{-2}$

or

```

```
$0.0571\;\frac{\mathrm{m}}{\mathrm{s}^2}$
```

or

```
$0.0571\;\mathrm{m}\,\backslash\,\wedge\,\mathrm{s}^2$
```

The example uses the existing definition for `a`. The `:frac` option uses LaTeX fractions instead of negative exponents, and the `:slash` option uses a slash, which is usually better for inline math.

Pasting the output in the previous listing into the LaTeX source of this book shows the rendered result:

```
a = 0.0571 m s-2
or
0.0571  $\frac{\text{m}}{\text{s}^2}$ 
or
0.0571 m / s2
```

We can change the default mode for rendering units with the `set_default(permode=:slash)` command.

Plotting with Units

Listing 9-3 shows how `Plots` knows how to handle `Unitful` quantities.

```
julia> using Plots, Unitful
julia> mass = 6.3u"kg";
julia> velocity = (0:0.05:1)u"m/s";
julia> KE = mass .* velocity.^2 ./ 2;
julia> plot(velocity, KE; xlabel="Velocity", ylabel="KE",
           lw=3, legend=:topleft, label="Kinetic Energy")
```

Listing 9-3: Plotting `Unitful` arrays

Here we import `Plots`, which we need for plotting, and `Unitful`, to handle units. After defining a mass in kilograms and a range of velocities in meters per second, we create an array of kinetic energies, `KE`, from the fact that kinetic energy = $1/2 \text{ mass} \times \text{velocity}^2$. The new package gives the plotting functions in `Plots` the ability to handle quantities with units and automatically appends the units to the axis labels. Figure 9-2 shows the result of the `plot()` statement.

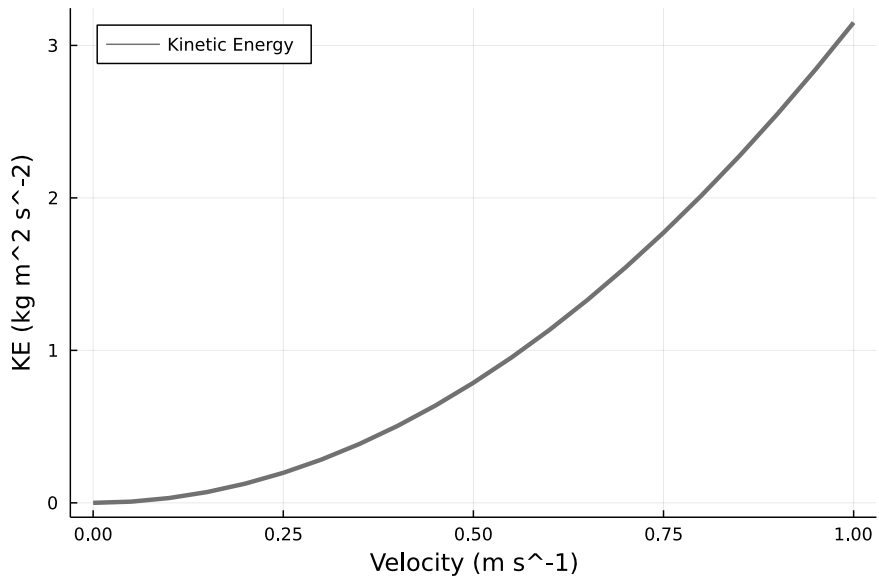


Figure 9-2: The plot that Listing 9-3 generates

I’ve left the energy units alone for this example, but more conventional physics usage would involve a conversion to joules using `uconvert()`, which we could have done before the plotting call or inline within `plot()`.

We were able to create this graph with the same `plot()` call that we might have used to plot the same quantities stored in numerical arrays without units. All the plotting functions in `Plots`, such as `scatter()` and `surface()`, work with `Unitful` arrays to produce similar axis labels.

Making Plots for Publication

When attempting to make high-quality plots for publication, however, we encounter some shortcomings. While `Plots` aspires to create a unified interface to a variety of backends, each plotting engine works somewhat differently, with each having unique capabilities and limitations.

These differences among backends become more salient when we are making the final adjustments that accompany the preparation of graphs for publication. It is at this stage that, for example, the typographic details in labels and annotations become important. Figure 9-2 was created using the `GR` backend, which, as mentioned in “Useful Backends” on page 115, is the default at the time of writing, and is fast and capable.

Figure 9-2 may be acceptable as is, but for publication we may want to improve the appearance of its graph labels, especially to make the unit notations look like conventional mathematical notation. As we saw in “LaTeX Titles and Label Positioning by Data” on page 103, we can use LaTeX notation in graph annotations with mathematical content. This also works for the automatic labeling using units with the packages we’ve already imported:

```
julia> using Plots, Unitful, Latexify, UnitfulLatexify

julia> plot(velocity, KE; xlabel="\textrm{Velocity}",
           ylabel="\textrm{KE}", unitformat=latexroundunitlabel)
```

The example repeats the plot command from Listing 9-3, but with some alterations to create LaTeX strings for the plot labels. The `unitformat` keyword processes the unit annotations through `latexify()`, with the value `latexroundunitlabel` retaining the parentheses around the units. Since this triggers placing the entire label into a LaTeX string, we also need to wrap the non-math parts of the labels in LaTeX commands to set them as normal text instead of math.

The GR Backend

The results of this approach depend critically on what backend we’re using. Obviously, it makes sense to use LaTeX strings only with backends that can do something with them. Although the default GR backend can interpret LaTeX, the results are not always adequate. This engine includes its own version of LaTeX processing, which often creates poor-quality typesetting with faulty kerning. The LaTeX engine in GR is the focus of some development activity, however, so its performance may improve.

Good-quality typesetting of labels in most cases requires processing by an external TeX engine, which involves a TeX installation such as TeXLive. As many physicists and other scientists have already made such an installation, we’ll move on to considering options that take advantage of it.

The Gaston Backend

Gnuplot can optionally be compiled with support for the `tikz` terminal, which saves plots as text files containing TikZ commands. (TikZ is a graphics language that comes with most full-featured TeX installations.) Such files are processed with LaTeX and can contain TeX or LaTeX markup for the annotations on the plot. The result is of the highest quality, with fonts and styles that match the document in which the plot is included. Unfortunately, at the time of writing, the Gaston backend, which uses gnuplot, does not properly support the `tikz` terminal, so this option is off the table. It’s being worked on, however, and once we can use Gaston with `tikz`, it will be the best option for complex plots for publication or when the best typographic quality is desired.

The PGFPlotsX Backend

Another backend that can make use of LaTeX strings is PGFPlotsX, which is invoked with the `pgfplotsx()` function. This backend creates plots by calling out to the LuaLaTeX TeX engine, which comes with most TeX installations, including TeXLive. Since LuaLaTeX does all the typesetting, the labels come out with TeX-level quality. This backend is, therefore, an excellent choice for publication-quality graphs. Gaston may still be the best future choice for complex plots because processing through LuaLaTeX can be far slower than through gnuplot if the plot contains a large number of elements, such as in a large scatterplot.

Handling Units Manually

Unfortunately, PGFPlotsX does not work properly with `Unitful`, not taking TeX processing into account. This limitation provides the opportunity to demonstrate a different way of plotting `Unitful` quantities and labeling axes with units—one that affords us complete control over the details.

The following listing contains the definition of a function that accepts two `Unitful` arrays for plotting, along with keyword arguments for labels:

```
using Plots, LaTeXStrings, Latexify, UnitfulLatexify

function plot_with_units(ux, uy; xl="", yl="", label="",
                        legend=:topleft, plotfile="plotfile")

    set_default(permode=:slash)
    x = ustrip(ux); y = ustrip(uy)
    ❶ xlabel = L"\textrm{%$xl}$ (%$(latexify(unit(eltype(ux)))))"
    ylabel = L"\textrm{%$yl}$ (%$(latexify(unit(eltype(uy)))))"

    plot(x, y; xlabel, ylabel, lw=2, label, legend)
    ❷ savefig(plotfile * ".tex")
    savefig(plotfile * ".pdf")

end
```

Using the `ustrip()` and `unit()` functions (see Listings 9-1 and 9-2), this code separates the arrays from their associated units, plotting the numerical parts and using the unit parts to construct labels with the `LaTeXStrings` package.

In order to interpolate values into a `LaTeXStrings` string, we need to use the two characters `%$` rather than a simple `$` ❶. When extracting the units from the arrays, we require the units of the elements of the array, which is why `eltype()` appears in the label assignment. The function saves both the stand-alone PDF version of the graph and its TeX version ❷ for including in a LaTeX document.

After selecting the desired backend, we call the function to create the *.pdf* and *.tex* files with the default names:

```
pgfplotsx()
plot_with_units(velocity, KE; xl="Velocity", yl="K. E.")
```

Figure 9-3 shows the result.

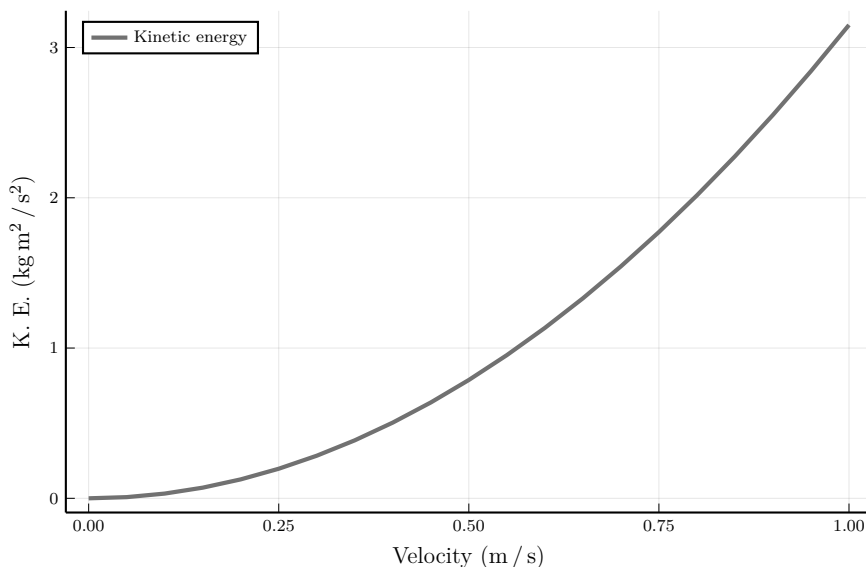


Figure 9-3: A PGFPlotsX plot with typeset unit labels

Typesetting by LuaTeX provides the excellent quality of the labels in Figure 9-3.

Error Propagation with Measurements

In the previous section we explored a package that extended the concept of numbers to include physical units. Here we'll meet `Measurements`, another package that defines a number-like object useful for calculations in physics or nearly any empirical science.

The `Measurements` package allows us to attach uncertainties to numbers. The number in question must be convertible to a float, so we can attach uncertainties directly to `Float64` numbers, integers, and `Irrational` quantities. (We can also create complex numbers with uncertainties, if we really want to, by attaching errors to their real and imaginary parts.) The `Measurements` package defines a new data type, called `Measurement{T}`, where `T` can be any size float. We can perform any arithmetic operations on `Measurement` types that are allowed on floats, and the errors, or uncertainties, will be propagated to the result using standard linear error propagation theory.

Here are some examples of creating instances of `Measurement` types:

```
julia> using Measurements
```

```
julia> 92 ± 3
92.0 ± 3.0
```

```
julia> typeof(ans)
Measurement{Float64}
```

```
❶ julia> 92.0f0 ± 3
92.0 ± 3.0
```

```
julia> typeof(ans)
Measurement{Float64}
```

```
julia> 92.0f0 ± 3f0
92.0 ± 3.0
```

```
julia> typeof(ans)
Measurement{Float32}
```

```
julia> big(1227.0) ± 2
1227.0 ± 2.0
```

```
julia> typeof(ans)
Measurement{BigFloat}
```

We create `Measurement` objects using a notation that will be familiar to scientists. We can type the \pm operator by entering `\pm` in the REPL and pressing `TAB` or by using the operating system's entry method for special characters.

In the REPL, the `ans` variable holds the most recently returned result. Since `Measurement` objects have only one type parameter, the base number and the error must be of the same type. As the `typeof()` calls show, `Measurements` promotes the smaller type as needed; the `f0` suffix is a way to enter 32-bit float literals ❶.

The package treats significant digits intelligently:

```
julia> π ± 0.001
3.1416 ± 0.001
```

```
julia> π ± 0.01
3.142 ± 0.01
```

The digits made insignificant by the error are not printed.

When printing results in the REPL, the package displays only two significant digits in the error, to keep things neat:

```
julia> m1 = 2.20394232 ± 0.00343
2.2039 ± 0.0034
```

```
julia> Measurements.value(m1)
2.20394232
```

```
julia> Measurements.uncertainty(m1)
0.00343
```

However, it retains the full values internally for computations. We can access these components with the `value()` and `uncertainty()` functions shown here, which, as they are not exported, we need to qualify with the package namespace.

Scientists often use an alternative, convenient notation to express uncertainty by appending the error in the final significant digits within parentheses. The `Measurements` package understands this notation as well:

```
julia> emass = measurement("9.1093837015(28)e-31")
9.1093837015e-31 ± 2.8e-40
```

In order to use the notation, we need to employ the `measurement()` function and supply the argument as a string. We can also use `measurement()` as an alternative to the `±` operator:

```
julia> m1 = measurement(20394232, 0.00343)
2.0394232e7 ± 0.0034
```

Arithmetic operations propagate errors correctly:

```
julia> emass
9.1093837015e-31 ± 2.8e-40
```

```
julia> 2emass
1.8218767403e-30 ± 5.6e-40
```

```
julia> emass + emass
1.8218767403e-30 ± 5.6e-40
```

```
julia> emass/2
4.5546918508e-31 ± 1.4e-40
```

```
julia> emass/2emass
0.5 ± 0.0
```

All these examples perform arithmetic as might be expected on the quantities and their errors. More interesting is the last example, where `Measurements` has recognized a ratio that has no error. The package maintains the notion of correlated and independent measurements, which is explained in its documentation. See “Further Reading” on page 304 for the URL.

Referring back to the example in Listing 9-3, we can add an uncertainty to the `Unitful` value for mass in two ways:

```
julia> using Measurements, Unitful
```

```
julia> mass = 6.3u"kg" ± 0.5u"kg"
6.3 ± 0.5 kg
```

```
julia> mass = 6.3u"kg"; mass = (1 ± 0.5/6.3) * mass
6.3 ± 0.5 kg
```

This example shows that the packages `Measurements` and `Unitful` can work together to create quantities with both units and uncertainties.

Let's continue with the example from Listing 9-3 using this new value for mass:

```
julia> using Plots
```

```
julia> velocity = (0:0.05:1)u"m/s";
```

```
julia> KE = mass .* velocity.^2 ./ 2;
```

```
julia> plot(velocity, uconvert(u"J", KE); xlabel="Velocity", ylabel="K.E.",
          lw=2, legend=:topleft, label="Kinetic energy")
```

Although, as before, velocity has no uncertainty attached to it, mass does; therefore, KE should also contain uncertainties.

Figure 9-4 shows the result.

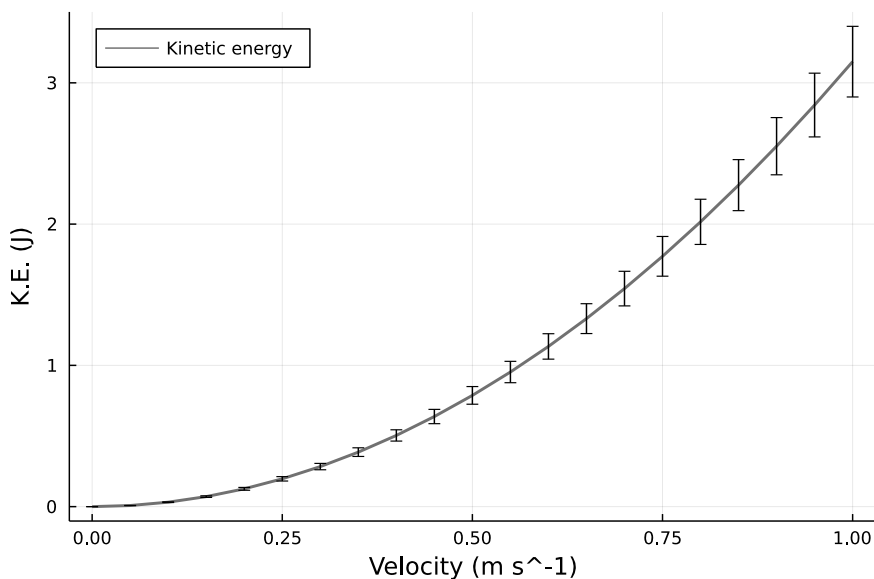


Figure 9-4: Plotting with units and errors

Figure 9-4 shows the Unitful arrays plotted as before with the axes labeled with their units. It also has error bars, showing how the error increases as the kinetic energy increases. We didn't have to change anything in the call to `plot()`. Somehow the type of the quantities to be plotted triggered the plotting function to use both unit labels and error bars. We would observe the same behavior with the other plotting functions in `Plots`, such as `scatter()` or `surface()`.

Fluid Dynamics with Oceananigans

The `Oceananigans` package for fluid dynamics simulations is especially well suited, as the name suggests, to the physics of the ocean. It provides a simulation construction kit that can include the effects of temperature and salinity variations, Earth's rotation, wind, and more. Its defaults usually perform well, but it's flexible enough that the user can specify one of several available solution methods. It has various physics models built in, including a linear equation of state, but makes it easy to substitute others of the user's devising.

The Physical System

We are setting out to simulate a two-dimensional layer of fluid in Earth's gravitational field. The bottom of the layer is maintained at a higher temperature than the top. This heating from below creates a convective motion, as can be seen in clouds or in a pan on the stove.

NOTE

Oceananigans depends on some compiled binaries in the standard library. If the pre-compile of `Oceananigans` fails and you're using a recent or beta version of Julia, try it with an earlier Julia release (the previous major version number).

The bottom and top simulation boundaries are impenetrable and free-slip, which means the fluid can slide across them. Horizontally, we impose a periodic boundary condition, requiring the solution to wrap around and be the same on the left and right boundaries. The horizontal direction is x and the vertical direction is z . We start the fluid at rest and are interested in the pattern of motion that the temperature difference creates.

Figure 9-5 shows the setup of the simulated system. The gray area represents the fluid, and the thick black horizontal lines indicate the constant-temperature boundaries.

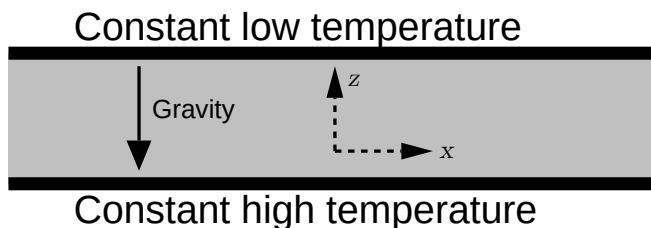


Figure 9-5: The simulation box

The Luxor program (see “Diagramming with Luxor” on page 190) that created this diagram is available in the Physics section of the online supplement at <https://julia.lee-phillips.org>.

A fluid dynamics simulation contains many pieces that we’ll need to construct separately before we can begin the calculation. In the following subsections, we’ll define the computational grid, the boundary conditions, the diffusivity models, and the equation of state, and establish the boundary conditions and the hydrodynamic model, in that order. After all the pieces are in place, we’ll run the `Oceananigans` simulation and visualize the results.

The Grid

To put together an `Oceananigans` simulation, we’ll define its various components using functions exported by the package, and then define a model using the `model()` function, passing in the components as arguments. For this example we’ll use a grid, a buoyancy model that specifies the fluid’s equation of state, a set of boundary conditions, the coefficients of viscosity and thermal diffusivity (material properties of the fluid), and initial conditions on the temperature within the fluid. We won’t include the effects of Earth’s rotation, salinity, or wind, but these ingredients are available for use in other `Oceananigans` models.

The grid is defined by its computational size (how many grid points exist in each direction), its extent (the physical lengths represented by these directions), and its topology, which is the term `Oceananigans` uses for what boundary conditions hold in each direction. For our problem we define the grid this way:

```
julia> using Oceananigans

julia> grid = RectilinearGrid(size=(256, 32);
    topology=(Periodic, Flat, Bounded),
    extent=(256, 32))
256×1×32 RectilinearGrid{Float64, Periodic, Flat, Bounded} on CPU with 3×0×3 halo
|-- Periodic x ∈ [0.0, 256.0)    regularly spaced with Δx=1.0
|-- Flat y
-- Bounded z ∈ [-32.0, 0.0]    regularly spaced with Δz=1.0
```

The `RectilinearGrid()` function that `Oceananigans` provides constructs grids as one of many data types defined in the package. We assign the grid to our own variable, `grid`, for use later when creating the model. We could have chosen any name for this variable, but `grid` is the name of the relevant keyword argument accepted by the model construction function; using the same names for our own variables will keep everything neat.

In the `topology` keyword argument, we list the boundary conditions in the x , y , and z directions, with z pointing upward. The boundary condition `Flat` means that we’re not using (in this case) the y direction. This call defines a two-dimensional, x - z grid, with periodic boundaries in x and impenetrable boundaries in z . `Oceananigans` uses a kilogram-meter-second unit system.

Because we set the extent to be equal to the size, the grid spacing is one unit in length along each dimension, giving us a fluid layer 256 meters wide and 32 meters tall.

As the example shows, `Oceananigans` has useful forms for representing its data types in the REPL, summarizing the salient information for our inspection. Here the output provides us with a summary of the grid parameters and boundary conditions.

The Boundary Conditions

We define any boundary conditions on physical variables as a separate component, which is also eventually passed into `model()`. We want to impose constant values of temperature on the top and bottom boundaries; `Oceananigans` sets this type of boundary condition with the `FieldBoundaryConditions()` function, as it sets boundary conditions on, in this case, the temperature field. We can use `Oceananigans`'s convenient definitions of `top` and `bottom`, which have their intuitive meaning (there are also `north`, `south`, `east`, and `west`, which we don't need in this problem):

```
julia> bc = FieldBoundaryConditions(
           top=ValueBoundaryCondition(1.0),
           bottom=ValueBoundaryCondition(20.0))
Oceananigans.FieldBoundaryConditions, with boundary conditions
|-- west: DefaultBoundaryCondition (FluxBoundaryCondition: Nothing)
|-- east: DefaultBoundaryCondition (FluxBoundaryCondition: Nothing)
|-- south: DefaultBoundaryCondition (FluxBoundaryCondition: Nothing)
|-- north: DefaultBoundaryCondition (FluxBoundaryCondition: Nothing)
|-- bottom: ValueBoundaryCondition: 20.0
|-- top: ValueBoundaryCondition: 1.0
-- immersed: DefaultBoundaryCondition (FluxBoundaryCondition: Nothing)
```

The `immersed` boundary refers to one that exists inside the fluid volume, but we're not using that one, nor any of the other myriad options, such as defined gradients or fluxes. The `ValueBoundaryCondition` that we use sets a constant value for a variable on the specified boundary.

The Diffusivities

We need to assign values to two constants that describe some of the fluid's material properties; this is part of the problem definition. The viscosity coefficient (ν) determines how "thick" the fluid is, and the thermal diffusivity (κ) determines how readily it conducts heat. These values are passed to the model in the `closure` keyword and can be set through the `ScalarDiffusivity()` function:

```
julia> closure = ScalarDiffusivity( $\nu=0.05$ ,  $\kappa=0.01$ )
```

The symbol for viscosity is the Greek letter ν and that for thermal diffusivity is κ . Like all Greek letters, we can precede their names with a backslash and then press TAB to enter them in the REPL.

The Equation of State

The equation of state is a function that describes how the density of the fluid at any point depends on the temperature and salinity there (the assumption of *incompressibility* usually used in Oceananigans models means that density has no dependence on pressure). Our model is salt free, but our fluid will be lighter when it's hotter. This is what will cause the fluid to move, as the lighter parts will rise and the heavier parts will sink, driven by gravity.

The `model()` function expects the keyword `buoyancy`, so we'll use that too:

```
julia> buoyancy = SeawaterBuoyancy(equation_of_state=
    LinearEquationOfState(thermal_expansion=0.01,
    haline_contraction=0))
SeawaterBuoyancy{Float64}:
|-- gravitational_acceleration: 9.80665
-- equation of state: LinearEquationOfState(thermal_expansion=0.01, haline_contraction=0.0)
```

Oceananigans offers many other options, including the ability to define our own equation of state, but we'll keep the model simple. The `SeawaterBuoyancy` component deals with buoyancy by combining gravity (with the default Earth value given here) with density variations. As we're not interested in salinity effects for this calculation, we set `haline_contraction` to 0 ("haline" is essentially a synonym for saline used by oceanographers).

The Model and Initial Conditions

Now that we have all the pieces set up, we can put them together into a *model*, the Oceananigans term for the definition of the computational problem, including all the physics along with the grid and the boundary conditions:


```
julia> model = NonhydrostaticModel(;
    grid, buoyancy, closure,
    boundary_conditions=(T=bc,), tracers=(:T, :S))
NonhydrostaticModel{CPU, RectilinearGrid}(time = 0 seconds, iteration = 0)
|-- grid: 256x1x32 RectilinearGrid{Float64, Periodic, Flat, Bounded}
    ① on CPU with 3x0x3 halo
|-- timestepper: QuasiAdamsBashforth2TimeStepper
|-- tracers: (T, S)
|-- closure: ScalarDiffusivity{ExplicitTimeDiscretization}
    (v=0.05, κ=(T=0.01, S=0.01))
|-- buoyancy: SeawaterBuoyancy with g=9.80665 and
    LinearEquationOfState(thermal_expansion=0.01, haline_contraction=0.0)
    with -ĝ = ZDirection
-- coriolis: Nothing
```

The package prints a nice summary of the result, including a reminder of some (but not all) of the features we’re not using, such as the coriolis force from Earth’s rotation.

The `NonhydrostaticModel()` function creates a model using the approximation appropriate to our problem. `Oceananigans` offers several other choices, including a hydrostatic model to simulate surface waves.

We use the abbreviated form of passing keyword arguments explained in “Concise Syntax for Keyword Arguments” on page 154.

Our boundary condition `bc` doesn’t refer to any particular physical variable; it simply defines a constant field value on the boundaries. The named tuple assigned to `boundary_conditions` enforces them on `T`, the variable used in `Oceananigans` for the temperature.

The printed result refers to the CPU , which means that this model is intended for “normal” machine architectures. The other option is to calculate on GPUs (graphics processing units). The `halo` refers to the several points outside the physical grid that the numerical algorithm uses to enforce the boundary conditions or other constraints.

The final keyword argument, `tracers`, tells the model to keep track of the temperature and salinity as those scalar fields are advected around the fluid. We’re required to include `:S` even though our equation of state means it will have no effect.

The fluid layer heated from below defined by our model is physically *unstable*, which means that a small perturbation to its initial, motionless state will be magnified and develop into a state with some form of persistent motion, driven by the temperature difference and the gravitational field. It is the development of the instability that we want to study. We need to add the small perturbation, or else, even though the system is unstable, it will never move.

The `set!()` function lets us create any desired initial condition on any of the fields. We’ll use it to add a small, random perturbation to the temperature field throughout the fluid volume:

```
julia> tper(x, y, z) = 0.1 * rand()
tper (generic function with 1 method)
```

```
julia> set!(model; T = tper)
```

The function is spelled with an exclamation point to remind us that it mutates its arguments: it alters the `T` field in place, and the model as well.

The Simulation

Next we need to create a *simulation*, using the `Simulation()` function. This object will receive the model as its positional argument, along with keyword arguments for the timestep and when to stop the calculation. It will keep track of how much simulation time and wall-clock time has elapsed and the state of all the physical fields. This allows us to continue the simulation after

the requested start time if we want, save the progress of the simulation in files, and retrieve the fields for examination and plotting.

```
julia> simulation = Simulation(model; Δt=0.01, stop_time=1800)
Simulation of NonhydrostaticModel{CPU, RectilinearGrid}(time = 0 seconds, iteration = 0)
|-- Next time step: 10 ms
|-- Elapsed wall time: 0 seconds
|-- Wall time per iteration: NaN years
|-- Stop time: 30 minutes
|-- Stop iteration : Inf
|-- Wall time limit: Inf
|-- Callbacks: OrderedDict with 4 entries:
| |-- stop_time_exceeded => Callback of stop_time_exceeded on IterationInterval(1)
| |-- stop_iteration_exceeded => Callback of stop_iteration_exceeded on IterationInterval(1)
| |-- wall_time_limit_exceeded => Callback of wall_time_limit_exceeded on IterationInterval(1)
| |-- nan_checker => Callback of NaNChecker for u on IterationInterval(100)
|-- Output writers: OrderedDict with no entries
-- Diagnostics: OrderedDict with no entries
```

This is a simple call, as `model` already contains all the details of the problem. We get a summary of various options for the simulation, most of which we didn't use. If you want to use the `delta` for the time interval in the REPL, enter `\Delta` and press `TAB`.

Before running the simulation, let's arrange for the velocity and temperature fields to be stored on disk at regular intervals so we can see its development over time (if we don't do this, we'll see only the final state of the simulation), as shown in Listing 9-4.

```
julia> simulation.output_writers[:velocities] =
        JLD2OutputWriter(model, model.velocities,
        filename="conv4.jld2", schedule=TimeInterval(1))
JLD2OutputWriter scheduled on TimeInterval(1 second):
|-- filepath: ./conv4.jld2
|-- 3 outputs: (u, v, w)
|-- array type: Array{Float32}
|-- including: [:grid, :coriolis, :buoyancy, :closure]
-- max filesize: Inf YiB

julia> simulation.output_writers[:tracers] =
        JLD2OutputWriter(model, model.tracers,
        filename="conv4T.jld2", schedule=TimeInterval(1))
JLD2OutputWriter scheduled on TimeInterval(1 second):
|-- filepath: ./conv4T.jld2
|-- 2 outputs: (T, S)
|-- array type: Array{Float32}
|-- including: [:grid, :coriolis, :buoyancy, :closure]
-- max filesize: Inf YiB
```

Listing 9-4: Setting up output writers

Adding elements to the `output_writers` property of the simulation causes it to store the results periodically. The `JLD2OutputWriter` uses the JLD2 file format, which is a compact way to store multiple Julia data structures in a single file. It's a version of the HDF5 format widely used in computational science. The `schedule` causes a data dump every 1 second, which, using our timestep, will be every 100 steps. The information in the result shows which quantities will be saved: `T` and `S` are the temperature and salinity.

With this, we're ready to run the calculation:

```
julia> run!(simulation)
[ Info: Initializing simulation...
[ Info:   ... simulation initialization complete (6.850 ms)
[ Info: Executing initial time step...
[ Info:   ... initial time step complete (80.507 ms).
```

The REPL will not have anything more to say until it reaches the final timestep, which in this case will take several hours on a typical personal computer. Then it will indicate that the calculation is complete and return to the interactive prompt. Chapter 15 explores ways to speed up such calculations by using parallel processing.

The Results

When an `Oceananigans` simulation ends, the final state of the fields (the velocity components and the temperature, in this case) is available as properties of the `model`. Listing 9-5 shows how to retrieve them.

```
julia> using Plots

julia> uF = model.velocities.u;

julia> TF = model.tracers.T;

julia> heatmap(interior(TF, 1:grid.Nx, 1, 1:grid.Nz)';
               aspect_ratio=1, yrange=(0, 1.5grid.Nz))
```

Listing 9-5: Examining the results of a simulation

The velocity and temperature fields are properties of the `model`. The `heatmap()` call will plot the two-dimensional temperature field, but first we need to turn it into an array with the `interior()` function. This function converts the `Oceananigans` field into a numerical array and trims away the halo points. Its arguments, following the field to convert, are the extents of the grid in each of the three directions; we enter a 1 to indicate an unused coordinate. In setting the `yrange`, we've accessed another property of the field, its grid shape. The prime after the array to plot transposes it so that it appears in its natural orientation, with a vertical gravity.

We would normally run a simulation for just a few timesteps and examine the fields in this way before running a long calculation, to make sure

we've set it up correctly. If we want to take another look after a few more timesteps, we can do this:

```
julia> simulation.stop_time+=10;
```

```
julia> run!(simulation);
```

These commands advance the simulation an additional 10 timesteps, after which we can repeat the steps in Listing 9-5 to see how things are going.

Returning now to the quantities stored in files, as set up in Listing 9-4, Listing 9-6 shows how to retrieve the entire history of a field.

```
julia> uF = FieldTimeSeries("conv4.jld2", "u")
256×1×32×1030 FieldTimeSeries{InMemory} located at
(Face, Center, Center) on CPU
|-- grid: 256×1×32 RectilinearGrid{Float64, Periodic, Flat, Bounded}
    on CPU with 3×0×3 halo
|-- indices: (1:256, 1:1, 1:32)
-- data: 256×1×32×1030 OffsetArray{::Array{Float64, 4},
    1:256, 1:1, 1:32, 1:1030} with eltype Float64 with
indices 1:256×1:1×1:32×1:1030
    -- max=7.66057, min=-7.88889, mean=2.79295e-11
```

Listing 9-6: Retrieving a field from the JLD2 file

The summary of the result shows that the `FieldTimeSeries` has dimensions of $256 \times 1 \times 32 \times 1,030$, which means that it's defined on a 2D, 256×32 grid and evolves over 1,030 timesteps.

After this call the entire history of the x -velocity field and its various properties are conveniently available. The data structure `uF` itself takes up almost no space:

```
julia> sizeof(uF)
544
```

The `sizeof()` function returns the amount of storage, in bytes, occupied by its argument. The actual data occupies $256 \times 32 \times 1,030 \times 8 = 67,502,080$ bytes.

We can plot the horizontal velocity field at any timestep:

```
julia> using Printf

julia> i = 50;

julia> h50 = heatmap(interior(uF[i], 1:grid.Nx, 1, 1:grid.Nz)';
    aspect_ratio=1, yrange=(0, 1.5grid.Nz),
    colorbar=:false, ylabel="z",
    annotations=[
        (0, uF.grid.Nz+15,
         text("Horizontal velocity at timestep $i", 12, :left)),
```

```

(0, uF.grid.Nz+5,
 text(@sprintf "Max = %.3g" maximum(uF[i])), 8, :left)),
(100, uF.grid.Nz+5,
 text(@sprintf "Min = %.3g" minimum(uF[i])), 8, :left)],
grid=false, axis=false)

```

We've added some labeling to the version in Listing 9-5, annotating the plot using properties read out from the field. Creating similar plots for timesteps 100 and 500, adding an xlabel to the last one, and putting them together with `plot(h50, h100, h500; layout=(3, 1))` creates the plot in Figure 9-6.

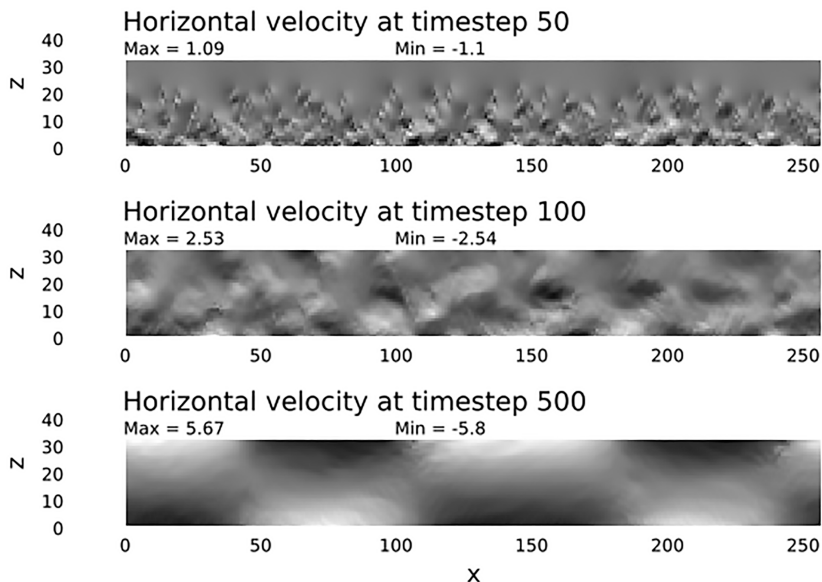


Figure 9-6: Results of an *Oceananigans* simulation

The system evinces the regime called *turbulent convection*; it's interesting to observe the emergence of large-scale order from randomness and its persistent coexistence with the turbulent flow.

In order to make an animation of the simulation, we need to generate plots at equally spaced time intervals and stitch them together into a video file. Our simulation used a constant timestep, so in this case, equal time intervals translates into equal numbers of timesteps. However, that won't always be the case. *Oceananigans* has options for automatically adjusted timesteps, and we may perform a simulation in stages with differently sized Δt . It's convenient, therefore, to have a function that creates a plot given a *time*. Since a given time may not correspond to any particular stored field, but may fall between two consecutive data dumps, we'll need a function that determines which stored field is closest to the time requested. The Julia program shown in Listing 9-7 retrieves the simulation output and produces a movie of a specified duration.

using Oceananigans, Reel, Plots

```
function heatmap_at_time(F, time, fmin, fmax, duration)
    ts = F.times
    time = time * ts[end]/duration
    i = indexin(minimum(abs.(ts .- time)), abs.(ts .- time))[1] ❶
    xr = yr = zr = 1
    if F.grid.Nx > 1
        xr = 1:F.grid.Nx
    end
    if F.grid.Ny > 1
        yr = 1:F.grid.Ny
    end
    if F.grid.Nz > 1
        zr = 1:F.grid.Nz
    end
    heatmap(interior(F[i], xr, yr, zr)'; aspect_ratio=1, yrange=(0, 1.5F.grid.Nz),
            clim=(fmin, fmax)) ❷
end

uF = FieldTimeSeries("conv4.jld2", "u")
const fmin = 0.5minimum(uF) ❸
const fmax = 0.5maximum(uF)
const duration = 30

function plotframe(t, dt)
    heatmap_at_time(uF, t, fmin, fmax, duration)
end

uMovie = roll(plotframe; fps=30, duration)

write("uMovie.mp4", uMovie)
```

Listing 9-7: Creating an animation of an Oceananigans simulation

The `heatmap_at_time()` function does what’s needed, creating a heatmap at the time closest to the time in its argument. In this function, `F` is a field retrieved with a call to `FieldTimeSeries()`, as in Listing 9-6. It makes use of the `times` property of these objects, which is an array holding all the times at which the field has been saved. The index `i` holds the dump corresponding to the time closest to the supplied time ❶. When making an animation of a heatmap, we want to use the same mapping from values to colors in each frame, so our call to `heatmap()` uses the `clim` keyword ❷.

With this function in place we can create an animation using the `Reel` package introduced in “Animations with Reel” on page 206. To work with that package, we need to define a function of time `t` and (an unused) `dt` that returns a plot corresponding to `t`: the `plotframe()` function. The three constants ❸ in the script set the palette limits based on the data and the desired

total duration of the animation. The palette limits are scaled so that more details are visible near the beginning of the run, but we can adjust it based on the features of interest.

NOTE

See the online supplement at <https://julia.lee-phillips.org> for the resulting animation, along with full-color versions of the figures.

The final call saves the animation as an MP4 file. Other options that will work with Reel are gif and webm. To create these file types, we merely need to use the appropriate file ending.

Solving Differential Equations with DifferentialEquations

Since the 18th century, differential equations have been the language of physical science and engineering, and of the quantitative aspects of other sciences as well. Julia’s DifferentialEquations package is a massive, state-of-the-art facility for solving many types of differential equations using a multitude of methods. It incorporates recent research on the use of machine learning to apply the best line of attack for solving a given equation.

This section introduces the use of DifferentialEquations by solving an example problem. Interested readers can delve into its detailed documentation for more information (see “Further Reading” on page 304).

Defining the Physics Problem and Its Differential Equation

As an example, let’s investigate the pendulum. Figure 9-7 diagrams the problem and defines the string length (L) and the angle (θ).

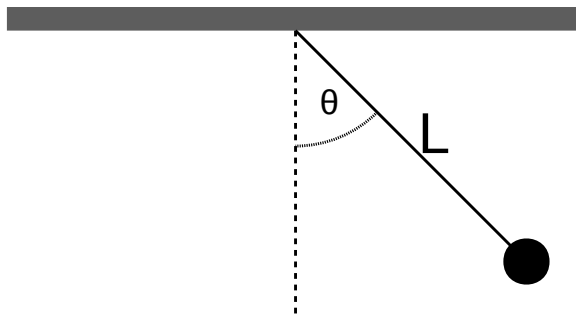


Figure 9-7: The pendulum system

We measure θ counterclockwise from the vertical reference line, which is dotted in the diagram, and the gravitational acceleration points down.

NOTE

The Luxor program that produced the diagram is available in the code section of the Physics chapter on the online supplement at <https://julia.lee-phillips.org>.

A straightforward analysis of the forces on the pendulum bob (the black circle in the diagram) and Newton's Second Law leads to the differential equation

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L} \sin(\theta)$$

which is derived in any introductory general physics text. Here t is time and g is the gravitational acceleration. The usual next step is to confine the problem to small angles ($\lesssim 5^\circ$), where $\sin(\theta) \approx \theta$, and solve the resulting differential equation for simple harmonic motion. We're going to solve the "exact" pendulum equation numerically, using the `DifferentialEquations` package. We'll be able to examine the solution for any initial θ , up to π radians.

The package works with systems of first-order equations, which means differential equations limited to first derivatives of the unknown function. To handle the pendulum equation, therefore, we first need to cast it into the form of two coupled first-order equations. This first step is also part of many analytic solution methods. We can proceed easily by defining a new variable:

$$\begin{aligned} \frac{d\theta}{dt} &= \omega \\ \frac{d\omega}{dt} &= -\frac{g}{L} \sin(\theta) \end{aligned}$$

Now we're solving for two functions of time, the angle $\theta(t)$ and the angular velocity $\omega(t)$.

Setting Up the Problem

The first step in translating the mathematical problem into a form that `DifferentialEquations` can digest is to define a Julia function of four positional arguments:

- du** An array for the derivatives of the solutions
- u** An array for the solution functions
- p** An array of parameters
- t** The time

Listing 9-8 is the version for the pendulum problem.

```
function pendulum!(du, u, p, t)
    L, g = p
    θ, ω = u
    du[1] = ω
    du[2] = -g/L * sin(θ)
end
```

Listing 9-8: The Julia version of the pendulum equation

This is a mutating function, as indicated by the exclamation point, because as the calculation progresses, the solution engine mutates the `u` and `du` arrays to hold the results. Here `L` and `g` are set through destructuring the array `p`, and `θ` and `ω` are read from the array `u`. The solver from `DifferentialEquations` will repeatedly call `pendulum!()` as it builds up the solution, passing in `p`, `t`, and the developing solution arrays themselves.

Solving the Equation System

To calculate the solution, we first define the computational problem and then pass that problem to the `solve()` function. The components of the computational problem are the parameter array, the initial conditions, the time span over which we want the solution, and the function that defines the differential equations to be solved, in this example `pendulum!()`. Other options include such things as the numerical method to be employed, but in this simple example we'll leave those options unspecified. The package generally does an excellent job of choosing the solution method best suited to the nature of the equations we present to it. Listing 9-9 shows the problem set up and initiated.

```
using DifferentialEquations

p = [1.0, 9.8]
#   L   g   <- Parameters

u0 = [deg2rad(5), 0]
#   θ   ω   <- Initial conditions

tspan = (0, 20)

prob = ODEProblem(pendulum!, u0, tspan, p)
sol5d = solve(prob)
```

Listing 9-9: Solving differential equations using DifferentialEquations

The only two functions in this section from the `DifferentialEquations` package are `ODEProblem()` and `solve()`. `ODEProblem()` takes four positional arguments: the function defining the equation system, an array of initial conditions, the time span, and the parameter array. We defined the function in Listing 9-8 and we define the other three arguments here. Allowing the solver to pass the parameters as arguments makes it convenient to generate families of solutions with a range of parameters.

The result returned by `ODEProblem()` contains the complete solutions of all functions (in this example, two) bundled into a data type defined in the package. This data type is designed to make it easy to examine and plot the solutions, and it contains, in addition to the computed functions, information about the problem and the calculation.

Examining the Solutions

For small angles, the analytic solution to our pendulum problem is

$$\theta(t) = \theta_0 \cos\left(\sqrt{\frac{g}{L}} t\right)$$

where θ_0 is the initial angle. The initial conditions in Listing 9-9 have the pendulum at rest with a starting angle of 5° , so the small angle approximation should be valid.

Since we know the analytic solution, we can check the numerical result against it. Listing 9-10 shows how we can plot one against the other.

using Plots

```
plot(sol5d; idxs=1, lw=4, lc=:lightgrey, label="Numeric",
      legend=:outright, title="Pendulum at  $\theta_0 = 5^\circ$ ")
```

```
L, g = p
```

```
plot!(t -> u0[1]*cos(sqrt(g/L)*t); xrange=(0, 20),
       ls=:dash, lc=:black, label="Analytic")
```

Listing 9-10: Solving for the small angle case

The first `plot()` call uses only one data argument, the solution itself, assigned to `sol5d` in Listing 9-9. This is neither an array nor a function, yet `plot()` seems to know how to display it. The first keyword argument, `idxs`, requests that (in this case) the first function, θ , is plotted. `idxs` does not appear in the documentation for the `Plots` package, and in fact is not defined in that package. Thus, it has no effect unless we first import `DifferentialEquations`.

The plot, shown in Figure 9-8, gives us confidence that we've set up the problem correctly and that the numerical solution methods are working.

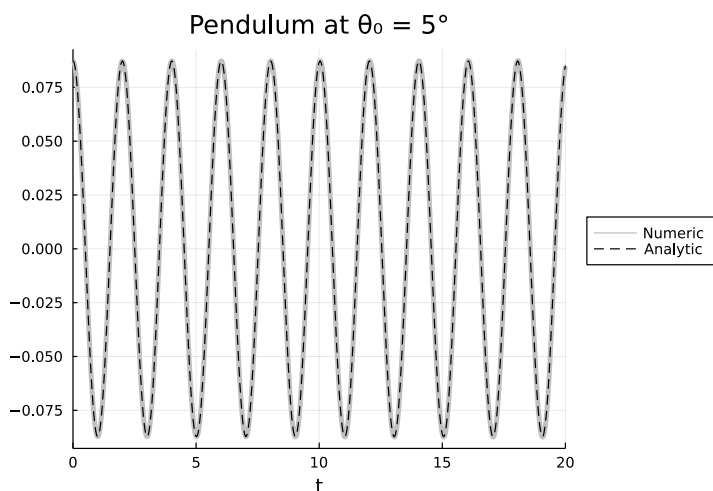


Figure 9-8: Checking the small angle solution of the pendulum equation

Plotting the solution as we did here does not simply plot the solution arrays. It also interpolates between calculated values in order to generate a smooth plot. In this case, the solution contains only 83 points, which, if plotted directly, would make a coarse graph.

Although the solution objects are not arrays, the package defines methods for indexing that make it convenient to extract the data. If we do want access to the uninterpolated solution data, we can get it by indexing. Here, `sol5d[1, :]` returns a Vector of the 83 points for the first variable, θ , and `sol5d[2, :]` for the second, ω . To get the times at which these values are defined, we use a property: `sol5d.t`.

Using the solution objects as functions returns the result interpolated to the time passed as an argument. (We're using time in this section, but in other problems the independent variable may be something else.) The `sol5d(1.3)` function call returns a Vector of two elements, one for each variable, interpolated to the time 1.3. These functions accept ranges and arrays as well, so `sol5d(0:0.1:1)` returns the interpolated solution data at 11 times from 0 to 1. To extract just the angle variable at these times, we can call `sol5d(0:0.1:1)[1, :]`. Controlling the density of the interpolation by using the functional form of the solution objects can be helpful when making, for example, scatterplots, where we need to control the density of plotted points.

How does the solution depend on the initial angle? Redefining `u0` to try two larger initial angles, and proceeding as in Listing 9-10 to generate two new solutions, we get the results shown in Figure 9-9.

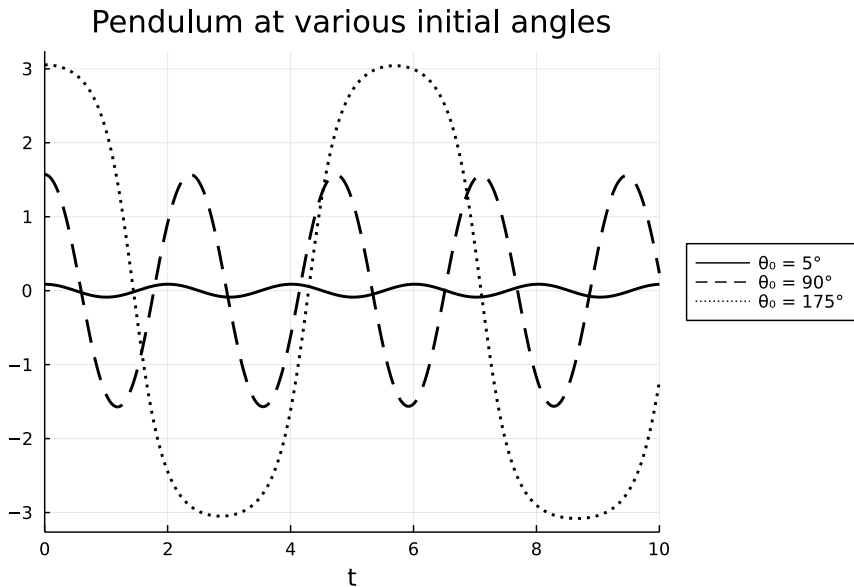


Figure 9-9: The pendulum with larger initial angles

The 90° solution, with the pendulum string initially horizontal, appears approximately sinusoidal, but with the frequency around 25 percent lower than the small angle case. When the initial angle is 175° , the period is nearly three times the small angle period, and the solution is clearly far from sinusoidal. In generating Figure 9-9, we limit the range of the independent variable by passing another `DifferentialEquations`-defined keyword to `plot()`: `tspan=(0, 10)`.

Defining Time-Dependent Parameters

By replacing one or more of the constant parameters in the `p` array with functions of time, we can study the system's response to time-dependent parameters. In this way we can include inhomogeneous terms in the differential equations, forcing functions, and time-varying parameters in general.

Let's find out what happens if we pull up on the string steadily as the pendulum oscillates. We'll start at 45° and calculate the solution over 10 seconds, replacing the constant `L` by a linearly decreasing function of time:

```
tspan = (0, 10)
u0 = [ $\pi/4$ , 0]
Lt(t) = 1 - 0.999t/10
```

We need to create a slightly different version of our `pendulum()` function, shown in Listing 9-11, that can use the time-dependent string length.

```
function pendulum2!(du, u, p, t)
    L, g = p
     $\theta$ ,  $\omega$  = u
    du[1] =  $\omega$ 
    ❶ du[2] = -g/L(t) * sin( $\theta$ )
end
```

Listing 9-11: The pendulum function with a time-dependent `L`

The only change we made to the previous function is replacing `L` with `L(t)` ❶. We proceed just as before. The `ODEProblem()` function needs a new parameter array, shown in Listing 9-12, to pass in to `pendulum2()`.

```
p = [Lt, 9.8]
prob = ODEProblem(pendulum2!, u0, tspan, p)
sol1t = solve(prob)
```

Listing 9-12: Getting the numerical solution with a time-varying `L`

The ease of generalizing the problem to include a time-varying parameter clarifies the advantages of the parameter-passing approach in `DifferentialEquations`. The result, in Figure 9-10, shows a steadily decreasing period and amplitude with an increasing angular velocity (ω).

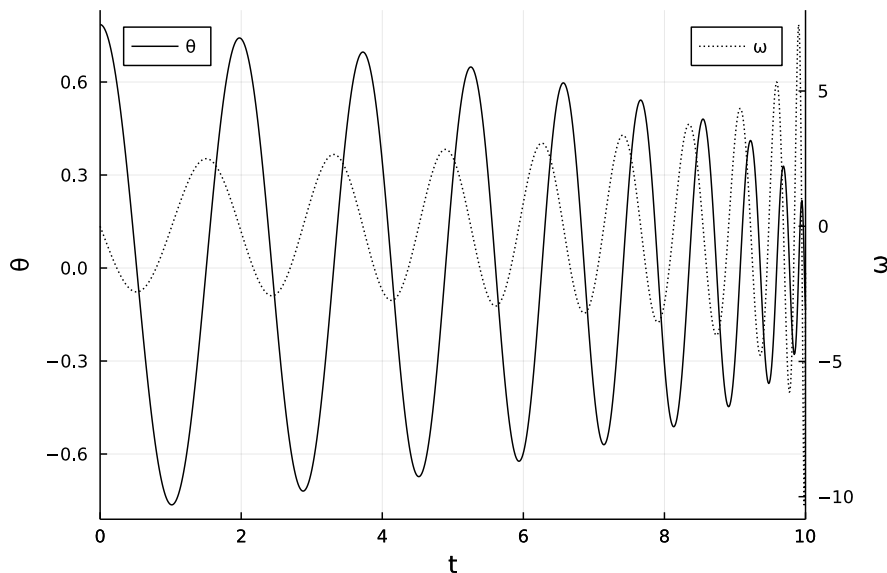


Figure 9-10: Pulling up the string on the pendulum

We create Figure 9-10 with the following calls:

```
plot(sollt; idxs=1, label="θ", legend=:topleft, ylabel="θ",
     ❶ right_margin=13mm)
plot!(twinx(), sollt; idxs=2, label="ω", legend=:topright,
     ylabel="ω", ls=:dot)
```

In the call to `plot!()`, the first argument, `twinx()`, creates a subplot overlay that shares the horizontal axis with the first plot and draws a new vertical axis; we use it so the two curves don't have to share the same scale. We need some extra room on the right ❶ for the labels on the second vertical axis. This margin setting requires the import of `Plots.PlotMeasures`, as explained in “Working with Plot Settings” on page 101.

Parametric Instability

A child “pumping” a swing in the playground to get it moving is exploiting a *parametric instability*. The driver of this instability is the periodic change in the effective length of the pendulum string. The results of linear theory (the small angle version of the differential equation that we’re attacking in this section) tell us that a resonance occurs when the forcing frequency is twice the natural frequency of the pendulum, which, using our $L = 1$, is $2\sqrt{g}$. If the string length is perturbed sinusoidally at this frequency, the amplitude of small oscillations will increase exponentially.

Since we know how to insert any time-dependent function $L(t)$ into the numerical solution, we can investigate the response of the pendulum to parametric excitation beyond the small angle approximation. We’ll start

with a small initial angle, follow the evolution for a longer span, and define a new function of time for the string length:

```
const g = 9.8
tspan = (0, 400)
u0 = [ $\pi/32$ , 0]
lt(t) = 1.0 + 0.1*cos(2*sqrt(g)*t)
```

$lt(t)$ will perturb the nominal length of 1 meter by 10 percent at the frequency of parametric resonance.

Our work proceeds exactly as before, with one adjustment. We use `pendulum2()`, defined in Listing 9-11, and set up the problem as in Listing 9-12. The adjustment is that we need to supply a keyword argument to the solving function:

```
sollt = solve(prob; reltol=1e-5)
```

The `reltol` parameter adjusts the adaptive timestepping as needed to limit the local error to the value that we supply. Its default of 0.001 led to a solution that seemed suspicious, as it was not quite periodic. I generated solutions with `reltol = 1e-4`, `1e-5`, and `1e-6`. The `1e-4` solution looked reasonable, but the `1e-5` solution was slightly different. As the solution with `reltol = 1e-6` looked identical to the one at `1e-5`, they're probably accurate. Figure 9-11 shows the resulting graph of θ versus time.

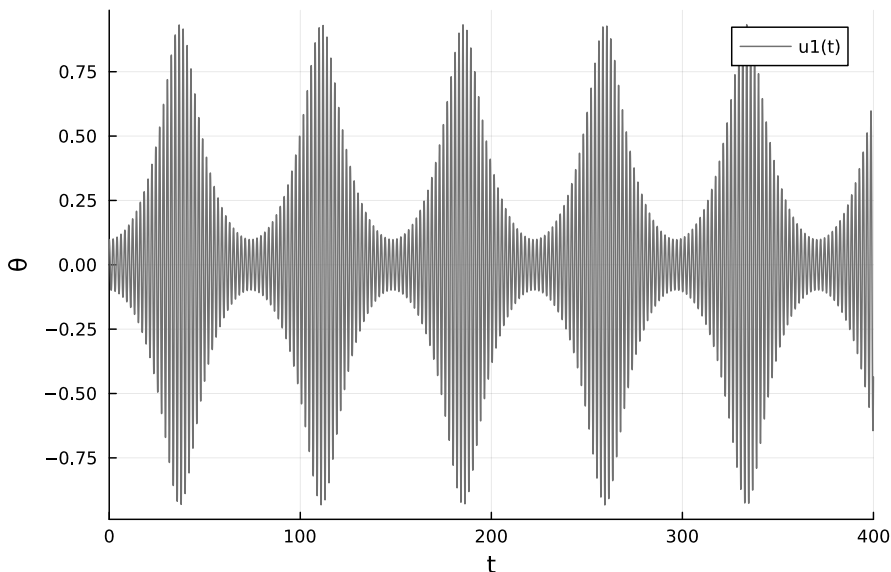


Figure 9-11: Parametric instability of the finite-angle pendulum

Initially, the amplitude increases exponentially, as predicted by the linear theory. But we know from our previous solutions that the frequency of the pendulum decreases with amplitude; therefore, it moves continuously out

of resonance with the forcing function, and the amplitude decreases back to close to its initial value. At that point it's closer to resonance, and the amplitude again grows exponentially. As the solution shows, the process repeats.

Combining DifferentialEquations with Measurements

Suppose we want to verify the predictions of our pendulum solutions with an experiment. There will be some error inherent in the setting of the initial angle. If we estimate that uncertainty to be one degree, we might think to state the initial conditions this way (see “Error Propagation with Measurements” on page 280):

using Measurements

```
u0 = [ $\pi/2 \pm \text{deg2rad}(1)$ , 0]
```

The function `deg2rad()` converts from degrees to radians.

We can proceed exactly as before, repeating the procedure shown in Listings 9-8 and 9-9. A plot of the solution for $\theta(t)$ now looks like Figure 9-12.

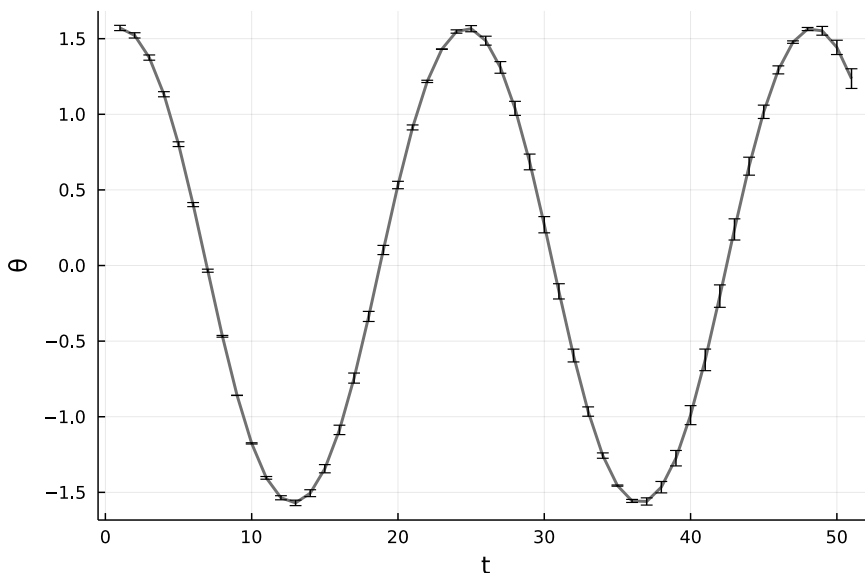


Figure 9-12: *Combining DifferentialEquations with Measurements*

Although we don't tell the `plot()` function anything about drawing error bars, they appear in the plot. The plot shows how the error in the angular position grows, on average, over time. The error doesn't grow monotonically, however. It decreases when the exact solution and those at the limits of the error bound happen to be in phase.

We generate the solution and plot it in Figure 9-12 as follows:

```
prob = ODEProblem(pendulum!, u0, tspan, p)

solM = solve(prob)

plot(solM(0:0.1:5)[1, :]; legend=false, lw=2, ylabel="θ", xlabel="t")
```

Since `DifferentialEquations` places an error on every point of the solution, including the points interpolated when creating a plot, we have to use the technique described in “Examining the Solutions” on page 297 to limit the number of points plotted; otherwise, the plot becomes too crowded with error bars and is impossible to interpret.

Conclusion

Although we delved into several physics packages at some length in this chapter, we really only scratched their surfaces. I hope, however, that the introductions here are sufficient to help you assess whether any of the packages explored in this chapter might be a good choice for your projects and to show you how to get started.

Another purpose of this chapter is to serve as an introduction to a superpower of Julia and the Julia ecosystem. In several examples we were able to combine the abilities of two or three packages without making any particular arrangements to do so. We made plots and typeset expressions that contained units, and saw that they were handled sensibly. We handed the output of a differential equation solver to a plotting function from a different package, and it extracted the relevant data and plotted it. We solved differential equations with error estimates in their initial conditions, and the error was propagated through the solution correctly. We plotted *this* result, and, as if by magic, the solution displayed error bars.

We wrote scripts and programs that combined the abilities of five packages in various combinations, giving them capabilities neither envisioned nor planned by their authors. Most of these packages were written without any knowledge of the others that we combined them with. The authors of these packages wrote their code in a generic way that allows Julia’s type system and its method of multiple dispatch to enable its functions to work with data types defined in other packages.

Julia initially attracted attention as a language that was as easy to pick up and be productive in as a high-level interpreted language, but one that was fast enough for the most demanding scientific work: “as easy as Python and as fast as Fortran.” The second reason for Julia’s increasing adoption in the sciences is its ability to combine the abilities of disparate packages with no additional work on the part of the application programmer. Julia creators and package authors refer to this property as the *composability* of packages, in analogy with the composition of functions.

FURTHER READING

- The GitHub community “Julia’s Physics Ecosystem” (<https://julia-physics.github.io/latest/ecosystem/>) maintains a convenient list of packages related to all areas of physics, and includes related packages for mathematics and plotting.
- The Unitful package is available at <https://github.com/PainterQubits/Unitful.jl>.
- See <https://www.simscale.com/blog/2017/12/nasa-mars-climate-orbiter-metric/> for details on how a mixup in units destroyed the Mars Climate Orbiter.
- The documentation for UnitfulLatexify is at <https://gustaphe.github.io/UnitfulLatexify.jl/dev/>.
- The Measurements package resides at <https://github.com/JuliaPhysics/Measurements.jl>.
- To get started with Oceananigans, see https://clima.github.io/OceananigansDocumentation/stable/quick_start/.
- The DifferentialEquations.jl documentation is available at <https://diffeq.sciml.ai/stable/>.
- Animations, color images, and supplementary code for this chapter are available at <https://julia.lee-phillips.org>.
- You can find simple examples of the use of DifferentialEquations.jl at <https://lwn.net/Articles/835930/> and <https://lwn.net/Articles/834571/>.
- The parametric instability of a pendulum is demonstrated in the video at https://www.youtube.com/watch?v=dGE_LQXy6c0.
- The theory of parametric resonance for the general harmonic oscillator is treated at https://www.lehman.edu/faculty/dgaranin/Mechanics/Parametric_resonance.pdf.