Module II

Module 2: ODEs and molecular dynamics

From the "clockwork universe" evolution of the planets to the chaotic motion of molecules in a fluid, our understanding of how systems change over time is encoded in the language of ordinary differential equations (ODEs). The foundational example is Newton's second law, $\mathbf{F} = m\mathbf{a}$, a deceptively simple equation that predicts the behavior of extraordinarily complex systems. However – for all but the most idealized setups there are no analytic solutions to the kinds of ODEs we typically encounter. We turn, instead, to the computer

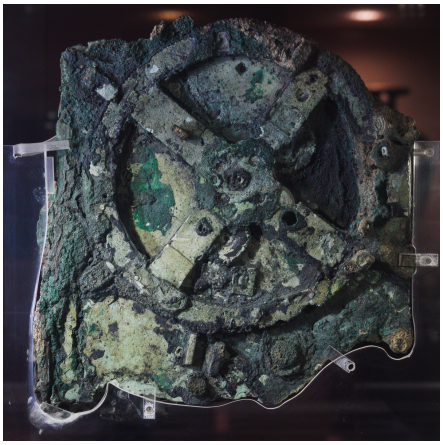

Figure 7.1: A fragment of the Antikythera mechanism – the oldest known analogue computer – which could be used to predict eclipses and the motion of various celestial bodies [22]. Photo credit: Logg Tandy, CC Attribution-ShareAlike 4.0.

In this module we'll use the classical "N-body" problem – in which we try to predict the motion of N classical point particles interacting via a conservative potential – as our paradigmatic example. We'll explore different numerical methods, from robust "black-box" solvers to specialized algorithms that leverage physical symmetries to achieve remarkable long-term stability. This contrast will reveal an important lesson: the best numerical method is *not* always the most mathematically accurate, but rather the one that respects the underlying structure of the governing physical laws.

With the N-body problem as our target, this module will be where some of the abstract ideas in Modules 0 and I become concrete[76]. "Weaving together algorithms and data structures[77]" will no longer a theoretical comment in the context of toy problems, but a practical necessity for designing our code. *Testing* will not just be about verifying simple functions, but about verifying physical conservation laws in complex systems. How can we put things together so that we can apply the right tools to the right problems? We'll think about all of this as we build simulations that are not only physically correct, but also robust, flexible, and clear.

For a deeper dive into the methods and physics discussed in this module, consider the following references [23, 24, 5]

[76] Does that count as a pun in the context of Julia?

[77] Has anyone been keeping track of how often I've said that during lectures?

Chapter 8

Ordinary differential equations

We are often interested in the time-evolution of some physical system – perhaps the trajectory that the planets will trace out over the course of a year, or how defects in a crystal will move, or how populations of competing species evolve. The central, paradigmatic challenge is to solve the *initial value problem*: We are given a set of first-order ODEs that describe how our system evolves,

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t),$$

along with the initial state of the system at a reference time, $\mathbf{y}(t_0)$. From this, we want to find the trajectory, $\mathbf{y}(t)$ for $t > t_0$.

The state vector $\mathbf{y}$ is, conceptually, a list of numbers that completely specifies the configuration of the system at any given moment. For a single point particle in one dimension, the state vector is simply the position and velocity, $\mathbf{y} = \{x, v\}$. For a molecular dynamics simulation of N particles in 3D, the state vector grows to a list of all positions and velocities – a list with $6N$ components.

The standard technique for handling higher-order equations – like Newton's law – is to recast them as a larger system of coupled first-order equations. For instance, as you already know, we can recast Newton's second law for a particle evolving in one dimension like so:

$$\ddot{x} = F/m \quad \longrightarrow \quad \dot{\mathbf{y}} \equiv \begin{pmatrix} \dot{x} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v \\ F/m \end{pmatrix} = \mathbf{f}(\mathbf{y}, t)$$

To keep the derivations in this chapter clean, we will often discuss and analyze our numerical methods using a single scalar equation, $\dot{y} = f(y)$. This is just for convenience – the same methods apply directly to the large state vectors that describe the complex physical systems we often care about.

8.1 The naive solution: Euler's Method

You will have already seen this approach in your previous computational methods class, but let's ramp up to this chapter by reminding ourselves what the simplest approach to time-discretized solutions to ODEs could be. The essential idea is to convert from a continuous-time representation of the problem to one in which the system evolves forward via small *discrete* timesteps

of size Δt. We can derive this method directly from a first-order Taylor series expansion of the state at time $t + \Delta t$:

$$\mathbf{y}(t + \Delta t) - \mathbf{y}(t) = \Delta t \cdot \dot{\mathbf{y}}(t) + \mathcal{O}(\Delta t^2) = \Delta t \cdot \mathbf{f}(\mathbf{y}(t), t) + \mathcal{O}(\Delta t^2).$$

Euler's method involves just stopping here, truncating the series and ignoring the terms of order Δt^2 and higher. This generates a *local truncation error* – the amount of error we make during each small step in our approach. We simply accept this, and use the above equation as the core iterative update rule that lets us propagate our system arbitrarily far forward in time. Adopting the common notation where $t_n \equiv t_0 + n\Delta t$ and $\mathbf{y}_n \equiv \mathbf{y}(t_n)$, the *Forward Euler Method* [25] is

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \cdot \mathbf{f}(\mathbf{y}_n, t_n). \tag{8.1}$$

This is the simplest possible numerical integrator. To see its characteristic flaws, let's apply it to a classical problem in celestial mechanics: simulating the time evolution of our solar system.

8.2 Case study: N-body simulations and planetary dynamics

Let's build a simulation of our very own Solar System. We will treat the planets and the Sun as a classical "N-body" problem, with point particles interacting according to some pairwise potential, in this case Newton's law of gravitation. Presumably we will find that the planets will trace out periodic orbits as they continue the celestial waltz that they've done for ages untold[78].

8.2.1 First attempt: A monolithic script

Our first instinct might be to write the most direct script possible – some highly specific "Solar System simulator." What might that look like? Code block 8.1 is one version. It defines the data as a simple list of lists, and has a few nested loops: an outer loop for each time step, during which we loop over objects in the solar system to compute the relevant accelerations (the function $\mathbf{f}(\mathbf{y}_n, t_n)$ in the notation above), and then perform another loop to perform a forward Euler update. This results in a short, self-contained piece of code.

This code certainly has it's merits: it is short, linear, and fairly easy to read and reason about from top to bottom. And, not forgetting that our own time is important, it can be written very quickly. On the other hand, the simplicity of this code might be deceptive. If the script actually contains everything we will ever want to do with the solar system we might be okay, but what happens when we want to change or improve it? For instance, what if[79] it turns out that Euler's method is not that good, and we want to be able to swap in a better numerical integration scheme? Or what if we want to reuse our code to simulate a similar system but with a different force law?

Unfortunately, the simple script above has a design that is much too brittle. The data and logic of it are inextricably mixed together, with the core loop accessing magic numbers and

[78] Ages untold? Just kidding – they've danced for about 4.6 billion years.

[79] Perish the thought!

```
# Data format (units?):
# SOL = [mass, x, y, z, v_x, v_y, v_z]
# MERCURY = ...
objects=[SOL,MERCURY,VENUS,EARTH,MOON,MARS,JUPITER,SATURN,URANUS,NEPTUNE]
const G = 6.67430 * 1e-11 # m^3 /(kg *s^2)

Δt = 0.0001
for t in 1:1e6
    # find the accelerations
    a = [[0.0,0.0,0.0] for _ in eachindex(objects)]
    for i in eachindex(objects)
        ai = [0.0,0.0,0.0]
        for j in eachindex(objects)
            if i != j
                r = objects[i][2:4] - objects[j][2:4]
                rn = sqrt(r[1]^2 + r[2]^2 + r[3]^2)
                forceij = (G*objects[i][1]*objects[j][1]/(rn^3)) * r
                ai += forceij / objects[i][1]
            end
        end
        a[i] = ai
    end

    # update the system
    for i in eachindex(objects)
        objects[i][2:4] += objects[i][5:7]*Δt
        objects[i][5:7] += a[i]*Δt
    end
end

# IS THIS EVEN CORRECT???????????
```

Code block 8.1: A self-contained Solar System simulator.

mutating global variables. It also clearly lacks any sense of modularity – the physics of gravity and the Eulerian numerical intergration method are mixed together in a single function, even though neither actually needs to know about the other. That lack of modularity also makes it hard to *test*. With such a script, how could we make sure that the calculation of the gravitational force is correct[80]? We cannot – we can basically only test the entire script, making it hard to isolate bugs.

There are any number of other problems with this code – some related to performance, others to brittleness, others to coding conventions – and certainly a better version of this monolithic script could be written without changing its fundamental design. Perhaps a better practice,

[80] By inspection? There's definitely a sign error in the code, right?

though, is to step back and ask ourselves how we could design a more robust, flexible, and verifiable architecture for our simulation in the first place.

8.2.2 A top-down design

One more structured philosophy we could turn to is the practice of *top-down design*. We'll *start* not with small pieces that we hope we can glue together later, but by designing the high-level function that will be our program's API[81]. We'll do that here, by first writing a high-level `run_simulation!` function. The design contract of that API will, in turn, help guide the implementation of all of the lower-level pieces. Code is about communication, and we'll try to see how a well-designed API can communicate the intent and structure of the entire underlying program.

Performance considerations and top-down design

Top-down design can be a powerful approach, but it can have serious drawbacks. Particularly when trying to design high-performance code – a common goal in computational physics – the top-down approach has a potential trap. One might design a beautiful / elegant API that ends up being fundamentally incompatible with the most efficient (or the most parallelizable) implementation of part of the program that might happen to consume the overwhelming majority of the runtime.
One solution is to practice top-down design and then rewrite everything whenever you discover this kind of implementation / performance incompatibility. A solution that involves pulling out less of your hair is sometimes called "yo-yo" or "meet-in-the-middle" design. In that pattern you sketch out the highest-level API, and then immediately jump to the lowest level – at this lowest level you build a prototype of the most computationally expensive part of the program. The performance characteristics at that level suggests some good design, allowing you to jump back to the high-level API and refine it to make sure it supports that efficient implementation. By designing the top and bottom levels in concert – working through all details and eventually meeting in the middle with a completed program – one can find a solution that is elegant *and* performant.

With that design philosophy in mind, let's define the high-level API for our simulation. The goal is to create a function signature that is clean, powerful, and flexible enough to accommodate different physical systems and numerical algorithms. There are multiple good design decisions we could make; code block 8.2 is one concrete proposal both for the function signature and its complete implementation.

Just by looking at the function signature we can read the entire architecture of our program. This most important feature is the clear *separation of concerns*: the API demands that the physical state (`system`), the force calculation logic (the `force_calculator`), and the time-stepping algorithm (the `integrator`) all be provided as distinct, independent objects. This

[81] "Application programming interface" – the contract that a piece of code presents, specifying the functions it has and how they must be used

```
function run_simulation!(
    system, force_calculator, integrator,
    time_step_size, number_of_steps;
    callback = (sys, step) -> nothing,
    callback_interval::Int=1
)
    for i in 1:number_of_steps
        integrate!(system,force_calculator,integrator,time_step_size)
        if i % callback_interval == 0
            callback(system, i)
        end
    end
end
```

Code block 8.2: A high-level API for a classical N-body simulation.

modularity in not an accident – it is a choice that mirrors the structure of the underlying problem. An integrator like Euler's method is a generic mathematical tool, and shouldn't need to know about the physics of gravity. Similarly, force laws are physical principles, independent of what numerical operations they are used for. The API mirrors, but also *enforces* this clean separation.

Top-down design and designing tests

Do you think this strategy of top-down design for APIs and writing modular code makes *writing good tests* for our code easier or harder? Why?

We also see the practical details. The function expects a scale for our discretization of time (`time_step_size`) and a number of iterations to run for (`number_of_steps`)[82]. It also provides two keyword arguments: a `callback` function[83] and a specification of how frequently to run that callback. This gives the user a "hook" into the `run_simulation!` function they can use to perform analysis or save data without having to modify the core simulation loop.

Reading the function body itself confirms the payoff of this design: the resulting code is extremely simple. "Running the simulation" becomes the trivial task of orchestrating the components, because all of the complexity has been abstracted away and encapsulated with the objects that the function calls. This is part of the essence of good design: complexity is not – cannot! – be erased, but it can be isolated to the components that are actually responsible for it.

[82] Perhaps a better design would be to specify Δt and a total duration to integrate forward in time for. However, choosing Δt and `number_of_steps` is pretty typical, and it avoids any danger of accumulating floating-point errors in the loop counter.

[83] A "callback" is a common term for function passed as an argument to another function, with the expectation that it will be "called back" (executed) at a specific time or when a certain event occurs.

8.2.3 Particle data structures

The design contract of our API dictates that we need a stateful `system`. Let's go ahead and build data structures suitable for *any* particle-based simulation.

A very intuitive approach is an "Array of Structs" (AoS) arrangement of our data, where all of the data for a single particle is bundled together:

```
struct Particle{D,T}
    position::SVector{D,T}
    velocity::SVector{D,T}
    mass::T
end
struct System{D,T}
    particles::Vector{Particle{D,T}}
end
```

This pattern is often the most convenient to work with as a programmer – all of the data associated with a given particle is immediately at hand. However, in high-performance computing you will frequently see the alternative "Struct of Arrays" (SoA) pattern:

```
struct System{D,T}
    positions::Vector{SVector{D,T}}
    velocities::Vector{SVector{D,T}}
    masses::Vector{T}
end
```

The SoA pattern often leads to faster code – this is not because of any algorithmic advantage, but rather because it organizes data in a way that is friendly to modern computer hardware. By storing the positions (e.g.) contiguously in memory, it allows for better cache efficiency and often allows the processor to perform operations on multiple data points simultaneously ("vectorization" of operations or "SIMD"). In these notes, for clarity we'll use the AoS pattern.

Given the above definitions, we can implement a custom *constructor* that builds the specific system we're interested in. In this case, we can grab actual data from NASA's remarkable JPL Horizons System[84] I've gone ahead and scraped the data for the mass, position (relative to the solar system's barycenter), and velocity of various celestial objects, and put them in constants. From that, we can build our `System` like so:

[84] An interesting combination of solar system data that includes both historical data on the location of celestial bodies, and also forward computation services.

```
const SOL::Vector{Float64} = [1988410., ...] # truncated for clarity
const MERCURY::Vector{Float64} = [0.3302, ...]
function SolarSystem()
    JPL_DATA = [SOL,MERCURY,...] # truncated for clarity
    solar_bodies::Vector{Particle{3,Float64}} = []
    MASS_UNIT = SOL[1] # 1 solar mass
    POS_UNIT = 149597870.7 # km (1 AU)
    TIME_UNIT = 31556736.0 # s (1 year)
    VEL_UNIT = POS_UNIT / TIME_UNIT # AU/year
    for orb in JPL_DATA
        mass = orb[1] / MASS_UNIT
        position = SVector{3,Float64}(orb[2:4]) / POS_UNIT
        velocity = SVector{3,Float64}(orb[5:7]) / VEL_UNIT

        particle = Particle(position, velocity, mass)
        push!(solar_bodies, particle)
    end
    return System{3,Float64}(solar_bodies)
end
```

Notice the very deliberate choice of units. On a computer all numbers are zeros and ones – they are *definitionally dimensionless*. The choice of a system of units is up to us, and due to floating point arithmetic that choice has practical consequences. If the numbers in a simulation vary by many orders of magnitude, floating point precision gets lost. A good rule of thumb is to use units so that the core quantities you need to work with are of order $\mathcal{O}(1)$. For the Solar System, using astronomical units, solar masses, and years is a natural choice.

Know your units

Never perform a simulation without knowing what your units are. Saying that the velocity is 1.0 is meaningless – is that a meter per second, or a light year per year? Failing to work with units correctly is a surprisingly common source of error in computational science. So: choose a consistent system for your simulation, convert all initial inputs to that system, and convert your outputs back to a more familiar or convenient set of units for analysis if you need to.

8.2.4 Integrators

The high-level API from code block 8.2 leads us to our next design challenge. The contract requires that we pass in an `integrator` object, and the main loop will itself call a function like `integrate!(system, force_calculator, integrator, dt)`. How will we design this next, internal contract? Just as we did with the `run_simulation!` function, we want to include only the complexity that is native to the integrator itself. How will we do this in a way that allows us to flexibly use not just the forward Euler method but any other integration scheme we later want to adopt?

Simple: we will leverage the power of Julia's multiple dispatch paradigm. We will define an abstract type that represents the *concept* of an integrator, and then build specific concrete `structs` for each different algorithm.

```
abstract type AbstractIntegrator end

struct ForwardEuler{D,T} <: AbstractIntegrator
    accelerations::Vector{SVector{D,T}}
end
function ForwardEuler(system::System{D,T}) where {D,T}
    return ForwardEuler(zeros(SVector{D,T},length(system.particles)))
end
```

Here we have made a crucial design choice: our integrator `structs` will be stateful. The `ForwardEuler` struct, for example, contains a pre-allocated vector to store accelerations. Since our integrator is stateful, we have provided a constructor – here we just need to allocate a large-enough array, but other integrators might require more complex initialization. A "purer" functional design might have a `compute_acceleration` that returns a new acceleration vector at each step, but allocating potentially large vectors at every timestep would be terrible for performance. By pre-allocating a kind of "scratch space" and mutating it in place with a `compute_acceleration!` function instead, we are making a deliberate trade-off in favor of performance. This is a classic example of a kind of meet-in-the-middle design philosophy, letting the lower-level performance considerations inform our higher-level API.

With our basic type hierarchy in place, we can now write a specific *method* for `integrate!` that dispatches on our `ForwardEuler` type. As shown in code block 8.3, this allows us to define the forward Euler algorithm as a nicely self-contained function.

```
function euler_step(p::Particle, acceleration, dt)
    new_position = p.position + p.velocity * dt
    new_velocity = p.velocity + acceleration * dt
    return Particle(new_position, new_velocity, p.mass)
end

function integrate!(system::System{D,T}, force_calc,
                integrator::ForwardEuler{D,T}, dt::Float64) where {D,T}

    compute_acceleration!(integrator.accelerations,system,force_calc)
    system.particles .= euler_step.(system.particles,
                                integrator.accelerations, dt)
end
```

Code block 8.3: An `integrate!` method that specializes on the `ForwardEuler` type. By adding new methods for other integrator types, we can extend our simulation's capabilities without changing other code.

To help with that, we've written pure `euler_step` function whose sole responsibility is to encapsulate the logic of the forward Euler algorithm, leaving the `integrate!` function as a

high level orchestrator whose code reads like a simple description of what it does: "calculate the accelerations and update all of the particles with an Euler step." Using the broadcasting assignment (`.=`) performs the operation efficiently and in place, giving us both the readability of a declarative style and the performance of a hand-written loop.

8.2.5 Force calculators

The contract that we just wrote for the integrator tells us what to do next: we need to flexibly design a way of calculating pairwise forces. Just as with the forward Euler method, we want to start with the simplest implementation, but give ourselves the flexibility to easily change to better ways of calculating the forces. Following precisely the same pattern as above, we first build a type hierarchy for our `ForceCalculators`, and while we're at it we'll define a simple helper function[85] that adds the contribution from a pairwise force to the vector of particle accelerations.

```
abstract type AbstractForceCalculator end

struct BruteForceCalculator{F} <: AbstractForceCalculator
    pairwise_force::F
end

function _apply_force_pair!(accelerations, i, j, particles, force)
    force_on_p1 = force(particles[i], particles[j])
    accelerations[i] += force_on_p1 / particles[i].mass
    accelerations[j] -= force_on_p1 / particles[j].mass
end
```

Our type declaration has the force calculators as, again, stateful entities – in this case, even our simplest force calculator will contain a reference to the specific force law that it will apply when computing interactions. But, with this work done, our `compute_acceleration!` is nicely declarative: it resets the acceleration vector and then, for every unique pair of particles, it call our helper function to do the work.

Once again we have a certain payoff for our modular design process. The `compute_acceleration` function is easy to read and reason about, with all of the details of applying Newton's third law encapsulated in a helper and all of the specifics of the physical force law encapsulated within the `calculator` object itself. In addition to making our code easier to extend with more advanced methods later, this clean separation also makes our code easier to test and maintain.

8.2.6 Results

We're now in a position to present our first test of all of this machinery by running a simulation of our Solar System! There are many ways we could present our results – traces of the trajectories of the planets relative to the Solar System's barycenter over time, phase-space plots of positions

[85] Another common Julia convention: functions in modules that start with an underscore are typically understood to be private "helper" functions that the user probably should not be calling.

```
function compute_acceleration!(accelerations::Vector{SVector{D,T}},
        sys::System{D,T},calculator::BruteForceCalculator) where {D,T}

    fill!(accelerations, zero(SVector{D,T}))

    n = length(sys.particles)
    for i in 1:(n-1)
        for j in (i+1):n
            _apply_force_pair!(accelerations, i, j, sys.particles,
                               calculator.pairwise_force)
        end
    end
    return nothing
end
```

Code block 8.4: A `compute_acceleration!` method that specializes on the `BruteForceCalculator` type. By adding new methods for other ways of computing forces, we can extend our simulation's capabilities without changing other code.

vs velocities in each planet's orbital plane. But the most fundamental test for any physical simulation is not about aesthetics; it's a test at the heart of physics: are quantities we know to be physically conserved *actually conserved numerically*.

Figure 8.1 puts our `ForwardEuler` integrator to the test. We simulate the solar system forward in time for one millennium and track the relative error in the total energy of the system. Since our method comes from a truncation of a Taylor series, we of course expect that our choice of Δt might matter. We thus compare several values of Δt – the smallest corresponds to jumping forward by less than an hour at a time, and the largest corresponds to a time step of roughly half a week at a time.

The plot is a damning indictment. There is a *systematic* error, with the total energy of our simulated solar system monotonically increasing. The rate of the energy drift depends on the step size, but the *fact* that it happens can't be changed by using ever more finely resolved increments of time.

As an aside, while the magnitude of the relative errors plotted above seem relatively modest on the scale of this plot, we have to remember that this is the relative error in the *total potential and kinetic energy of the entire solar system*. To drive the point home: in the simulation with the smallest time-step size, $\Delta t = 10^{-4}$ years, the unphysical energy gain is such that the Earth's orbit has widened all the way to ≈ 2.97 AU from the Sun after just one thousand years and the moon has drifted so that it sits a distance 0.45 AU away from us[86]. This tragic failure is not some bug in our code, but a fundamental flaw in the algorithm we chose. The obvious culprit is the crude, first-order accuracy of the Euler method – perhaps the logical next step, then, is to work with more mathematically sophisticated *higher-order* integrators? We'll explore that idea in Chapter 9.

[86] I suppose we wouldn't have to worry much about global warming – or the tides – in that scenario. Hardly comforting.

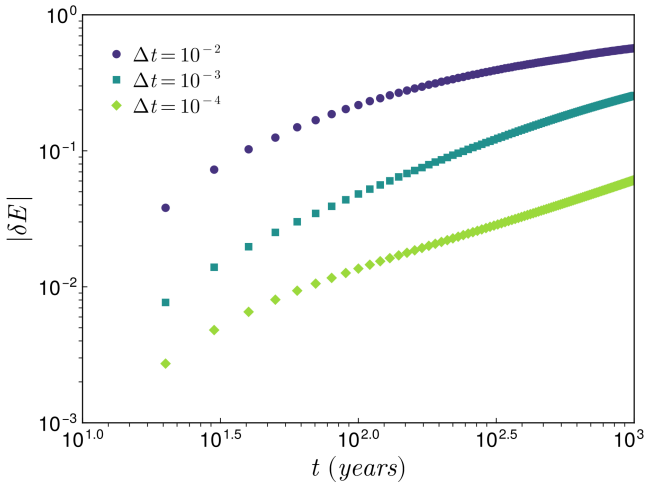

Figure 8.1: The relative error, $\delta E = (E(t) - E(t_0))\,/|E(t_0)|$, in numerically computing the total energy of our solar system simulation over the course of a millennium relative to its state when the simulation was started. Data correspond to using the forward Euler method with three different values of Δt (in units of Earth-years).

Chapter 9

Integration schemes for ODEs

In the last chapter we first introduced the simple forward Euler method for solving the initial value problem, and then set up a framework of code to simulate classical N-body systems. There are some settings where the humble Euler method works well, but when we tried to apply it to the problem of celestial mechanics with just a handful of objects in our solar system things quickly fell apart.

9.1 Higher-order integrators

One obvious suspect in the poor results above is the crude, first-order accuracy of the forward Euler method. The local truncation error of $\mathcal{O}(\Delta t^2)$ means that the *global* error, accumulated over many steps to reach a fixed time $T = N\Delta t$, will scale as $\mathcal{O}(\Delta t)$. For many applications, this is simply not good enough. A logical next step would be to find a method that accounts for higher-order terms in the Taylor series expansion of $\mathbf{y}(t + \Delta t)$.

Expanding $\mathbf{y}(t + \Delta t)$ to higher order, this time going back to a scalar example to keep the notation clean:

$$y(t + \Delta t) = y(t) + \Delta t \dot{y}(t) + \frac{\Delta t^2}{2}\ddot{y}(t) + \mathcal{O}(\Delta t^3).$$

We know that $\dot{y} = f(y, t)$. Adopting the notation in which f_y and f_t mean the derivative of f with respect to the subscripted variable, we can find the second derivative, $\ddot{y}$, by applying the chain rule to f:

$$\ddot{y} = \frac{d}{dt}f(y(t), t) = \frac{\partial f}{\partial y}\frac{dy}{dt} + \frac{\partial f}{\partial t} = f_y f + f_t.$$

Substituting these into the Taylor expansion gives us a second-order accurate update rule:

$$y(t + \Delta t) = y(t) + \Delta t f + \frac{\Delta t^2}{2}(f_y f + f_t) + \mathcal{O}(\Delta t^3).$$

While this "Taylor series method" is indeed more accurate, it's often impractical. It requires us to analytically calculate the partial derivatives of $\mathbf{f}$, which can be extremely complicated for some of the complex functions that arise in physical simulations. So, while tempting in its simplicity, such Taylor series methods are very rarely used in practice.

The insight of mathematician Carl Runge [26] – developed substantially farther by Wilhelm Kutta [27] – was to recognize that we can approximate this higher-order Taylor expansion with explicitly calculating higher-order derivatives of $\mathbf{f}$. The key idea is to use multiple evaluations of $\mathbf{f}$ *within each timestep* to probe the function's higher-order derivatives.

9.1.1 Deriving the RK2 Family

To see this, let's derive the "RK2" family of integrators. We will first take a small step which is a fraction of the discretized Δt, and then use the information from that step to compute a more accurate final update to get to the end of the timestep. The general version of this would look like:

$$\begin{aligned}\mathbf{k}_1 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n, t_n)\\ \mathbf{k}_2 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + \alpha\mathbf{k}_1, t_n + \beta\Delta t)\\ \mathbf{y}_{n+1} &= \mathbf{y}_n + a\mathbf{k}_1 + b\mathbf{k}_2\end{aligned}$$

Here the first stage, $\mathbf{k}_1$, is the familiar forward Euler step. The second stage, $\mathbf{k}_2$, evaluates the slope at some intermediate point in both time ($t_n + \beta\Delta t$) and state space ($\mathbf{y}_n + \alpha\mathbf{k}_1$). The actual integration to get to $\mathbf{y}_{n+1}$ uses a weighted average of the two slope estimates, with weights a and b. In order to actually be useful, we now choose the free paramaters a, b, α, β to make the update rule match the second-order Taylor expansion.

Performing a Taylor expansion of $\mathbf{k}_2$ around $(\mathbf{y}_n, t_n)$ (again, switching to scalar notation for clarity), we have

$$\begin{aligned}k_2 &= \Delta t \cdot f(y_n + \alpha k_1, t_n + \beta\Delta t)\\ &= \Delta t\left[f(y_n, t_n) + \alpha k_1 f_y + \beta\Delta t f_t + \mathcal{O}(\Delta t^2)\right]\\ &= \Delta t f + \Delta t^2(\alpha f f_y + \beta f_t) + \mathcal{O}(\Delta t^3)\end{aligned}$$

Substituting both this and the $k_1 = \Delta t f$ back into the update rule for y_{n+1} gives

$$y_{n+1} = y_n + (a+b)\Delta t f + b\Delta t^2(\alpha f f_y + \beta f_t) + \mathcal{O}(\Delta t^3). \tag{9.1}$$

The actual Taylor expansion is

$$y_{n+1} \approx y_n + \Delta t f + \frac{\Delta t^2}{2} f f_y + \frac{\Delta t^2}{2} f_t. \tag{9.2}$$

Matching the coefficients of Eqs. (9.1) and (9.2), we arrive at a system of three equations and four unknowns:

$$\begin{aligned}(\text{coeff of } \Delta t f): &\quad a + b = 1\\ (\text{coeff of } \Delta t^2 f f_y): &\quad b\alpha = 1/2\\ (\text{coeff of } \Delta t^2 f_t): &\quad b\beta = 1/2\end{aligned}$$

Thus, we see there is a *family* of second-order Runge-Kutta (RK2) methods: we are free to choose one parameter arbitrarily, and then determine the others from the system of equations above. Two popular choices are

- **Heun's Method:** $\{\alpha = 1,\ \beta = 1,\ a = 1/2,\ b = 1/2$. This update rule takes the average of the slope at the beginning and an estimated end of the interval.
- **The Midpoint Method:** $\{\alpha = 1/2,\ \beta = 1/2,\ a = 0,\ b = 1$. This uses an Euler step to estimate the state at the middle of the time step, evaluates the slope there, and uses *that* slope to make the step from t_n to t_{n+1}.

Both methods have a local truncation error of $\mathcal{O}(\Delta t^3)$, leading to a much more favorable global error of $\mathcal{O}(\Delta t^2)$. Different choices of the parameters do lead to different characteristics of the solvers, though. For instance, the midpoint method tends to do well when the ODE's behavior is dominated by oscillations, whereas Heun's method is a more robust general-purpose solver that tends to be slightly more stable than the midpoint method. Thus, there tends to be a certain art to determining which higher-order method to use for any particular problem.

9.1.2 The Butcher tableau

Extending this process to higher orders is a straightforward but algebraically tedious process. To simplify the notation and classification of different schemes, a compact representation known as *Butcher tableau* was developed [28]. This notation gives a simple, unambiguous "recipe" that completely specifies all of the parameters of general Runge-Kutta methods.

An s-stage explicit Runge-Kutta method is defined by the general form:

$$\begin{aligned}
\mathbf{k}_1 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n, t_n) \\
\mathbf{k}_2 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + a_{21}\mathbf{k}_1, t_n + c_2\Delta t) \\
\mathbf{k}_3 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + a_{31}\mathbf{k}_1 + a_{32}\mathbf{k}_2, t_n + c_3\Delta t) \\
&\vdots \\
\mathbf{k}_s &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + \sum_{j=1}^{s-1} a_{sj}\mathbf{k}_j, t_n + c_s\Delta t) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \sum_{i=1}^{s} b_i\mathbf{k}_i.
\end{aligned}$$

In order to be consistent (i.e., achieve *at least* first-order accuracy), we require $\sum_i b_i = 1$. A common convention is to additionally impose a row-sum requirement, $c_i = \sum_j a_{ij}$. Other constraints come from the order of accuracy that is desired, and methods are often thought of as a pair of integers labeling the number of stages they require and the order $\mathcal{O}(\Delta t^n)$ they achieve.

In any case, the coefficients that uniquely define a method – the a_{ij}, b_i, and c_i values – can be neatly arranged in the "tableau":

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^T \end{array} \quad = \quad \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \tag{9.3}$$

For the *explicit* methods we are considering, the matrix A is strictly lower-triangular ($a_{ij} = 0$ for $j \geq i$), and $c_1 = 0$. The zeros in the upper-right of A are often omitted for clarity. Using this notation, we can represent our previous examples very concisely.

0	
	1

Forward Euler (RK1)

0		
1/2	1/2	
	0	1

Midpoint method (RK2)

0		
1	1	
	1/2	1/2

Heun's method (RK2)

9.1.3 The workhorse ODE solver: RK4

Perhaps the most famous and widely used integrator is the "classic" fourth-order Runge-Kutta method, which provides an excellent balance of accuracy and computational cost. Its update rule is given by the Simposon's rule weighted average:

$$\begin{aligned}
\mathbf{k}_1 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n, t_n) \\
\mathbf{k}_2 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + \frac{1}{2}\mathbf{k}_1, t_n + \frac{1}{2}\Delta t) \\
\mathbf{k}_3 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + \frac{1}{2}\mathbf{k}_2, t_n + \frac{1}{2}\Delta t) \\
\mathbf{k}_4 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + \mathbf{k}_3, t_n + \Delta t) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)
\end{aligned}$$

And its corresponding Butcher Tableau is:

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

This method has a local error of $\mathcal{O}(\Delta t^5)$ and a global error of $\mathcal{O}(\Delta t^4)$, making it a robust and popular choice for a wide variety of problems.

But just as with the RK2 methods, integrators with different tableau may perform better or worse on specific ODEs. It is also important to consider the number of stages in these integrators – more stages imply more evaluations of the function $\mathbf{f}$. In a physical simulation each computation of the forces might be *the* most computationally expensive step; if modest accuracy is acceptable, it might be that a lower-order method with a smaller Δt might takes less total time to run for a fixed duration than a higher-order method with a larger Δt. On the other hand, if the evaluation of the function with derivative information is cheap – perhaps you are simulating a large system of linear ODEs – or you really need highly accurate answers, using higher-order methods can be tremendously advantageous.

9.1.4 Planetary dynamics with RK4

The brute-force mathematical solution to the low accuracy of Euler's method is to use a higher-order integrator. But is the extra implementation complexity worth it? Thanks to the flexible,

extensible framework we designed in Chapter 8, we don't have to refactor any of our code to find out – we just need to define a new integrator struct and write a new `integrate!` method that can dispatch on it. For RK4, we need to calculate four intermediate "slope" vectors (the k_1 - k_r above) at each time step. To avoid allocating new memory for these vectors at each time step, we'll pre-allocate a scratch space inside of our integrator object:

```
struct RungeKutta4{D,T} <: AbstractIntegrator
    initial_particles::Vector{Particle{D,T}}
    k1_accel::Vector{SVector{D,T}}
    k2_accel::Vector{SVector{D,T}}
    k3_accel::Vector{SVector{D,T}}
    k4_accel::Vector{SVector{D,T}}
end
```

The constructor will make sure these scratch spaces are the correct size, and we'll add an `integrate!` method that specializes on this new type, implementing the pattern of updates we wrote down in Section 9.1.3.

Was it worth the effort for our investigation of the future of our solar system? Figure 9.1 compares the relative error in the energy, again over the course of a millennium, for forward Euler approaches with $\Delta t = 10^{-4}$ and $\Delta t = 10^{-6}$ with the classic RK4 integrator discretized at $\Delta t = 10^{-3}$. From our analysis above we expect that the RK4 approach will take less time than either of the forward Euler integrations – it needs 10 and 1000 fewer time steps, respectively, to integrate a fixed duration in time compared, so the fact that each time step is broken into 4 relatively expensive "compute the acceleration and do some calculations" mini-steps doesn't bother us. We also expect that it will be *much* more accurate than either forward Euler approach: global errors of $\mathcal{O}(\Delta t^4)$ are just much smaller than global errors of $\mathcal{O}(\Delta t)$.

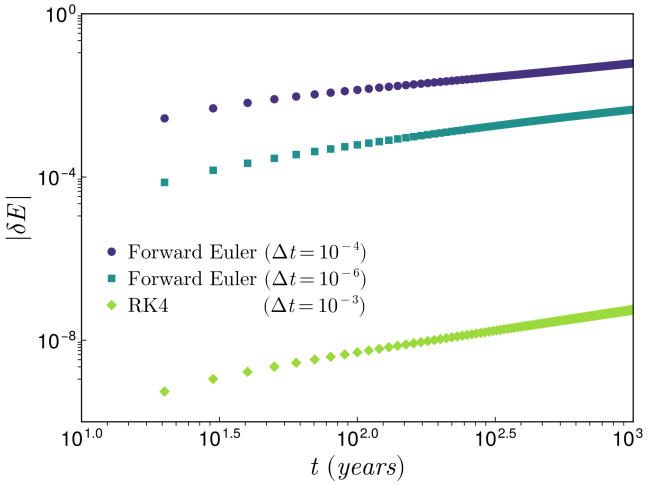

Figure 9.1: The relative error, $\delta E = (E(t) - E(t_0))/|E(t_0)|$, in numerically computing the total energy of our solar system simulation over the course of a millennium relative to its state when the simulation was started. Data correspond to using the forward Euler method with two different values of Δt (in units of Earth-years), compared to the classic RK4 method using a larger time step than either forward Euler simulation.

Well, Fig. 9.1 confirms both of those expectations. But it *also* confirms that we haven't solved the fundamental problem – just as with the forward Euler methods, the RK4 method also fails to conserve energy. The magnitude is much smaller, but the qualitative error – a slow, steady, upward drift – persists. Perhaps we've delayed the catastrophic consequences, delaying them from a milliennium's time to millions of years from today, but what's a few million years to to Solar System? The blink of an eye.

What then, are we to do? The Runge-Kutta family of integrators is extremely powerful, and is an excellent tool for general-purpose ODE solving, but it is blind to the special structure of physical laws. Do we just have to resign ourselves to using time steps so tiny that the inevitable drift of our integrator stays under some acceptable threshold? No. We have forgotten to think like physicists: the solution to our problem is not to be found in ever-higher-order Taylor expansions, but in a different class of integrators. Integrators that are designed from the ground up to respect the fundamental symmetries of physical systems.

9.2 Time-reversible integration

One of the most important features of Hamiltonian dynamics is that it is time-reversible – do the integrators we are using so far have this property? Let's explicitly show that the answer is "clearly not." To demonstrate, let's return to the forward Euler method in the context of a simple harmonic oscillator in one dimension. The state vector will just be $\vec{y} = \{q, p\}$, and (choosing a convenient system of units) the time derivative is $\vec{f} = \{p, -q\}$. We'll probe the time-(ir)reversibility of the method by (1) applying the forward Euler rule forward by one step from some initial state, and then (2) applying the rule again, but in the backwards ($-\Delta t$) time direction; we'll see if the system ends up where it started.

Starting at $\vec{y}_0$, the forward step is just

$$\vec{y}_1 = \begin{pmatrix} q_0 + \Delta t\, p_0 \\ p_0 - \Delta t\, q_0 \end{pmatrix}. \tag{9.4}$$

The final position after a backwards step is then $\vec{y}_f = \vec{y}_1 + (-\Delta t) f(\vec{y}_1)$. Evaluating this expression, we find

$$\vec{y}_f = \begin{pmatrix} q_0 + \Delta t\, p_0 - \Delta t\, p_0 + \Delta t^2 q_0 \\ p_0 - \Delta t\, q_0 + \Delta t\, q_0 + \Delta t^2 p_0 \end{pmatrix} = (1 + \Delta t^2) \begin{pmatrix} q_0 \\ p_0 \end{pmatrix}. \tag{9.5}$$

That is: after running time forward and then backwards, we find our system at a state different from where it started. Surprisingly, just a tiny adjustment to our forward Euler method can correct this flaw.

9.2.1 Symplectic Euler integration

The symplectic[87] Euler method proposes an unequal way of propagating positions and momenta forward in time. In the context of a system governed by the Hamiltonian $\mathcal{H} = T + V$ –

[87] "Symplectic" because it preserves phase space volumes under Hamiltonian evolution. The word was proposed by Weyl as alternative name to what he had previously called "complex groups" [29], a structure with close analogy to the orthogonal group.

i.e., composed of a kinetic term that depends only on particle momenta and a potential term that depends only on positions – the symplectic Euler method is formulated as either

$$\begin{aligned} p_{n+1} &= p_n - \Delta t V'(q_n) \\ q_{n+1} &= q_n + \Delta t T'(p_{n+1}) \end{aligned} \tag{9.6}$$

or

$$\begin{aligned} q_{n+1} &= q_n + \Delta t T'(p_n) \\ p_{n+1} &= p_n - \Delta t V'(q_{n+1}) \end{aligned} \tag{9.7}$$

Notice that in the first formulation, advancing the position requires knowing the *future* momenta (and vice verse in the second formulation). Notice, furthermore, that these two formulations do the same operations but in the reverse order: a "kick and then drift" or a "drift and then kick" of the particles. Because of the nicely separable structure of these equations, though, implementing this is straightforward:

```
struct SymplecticEuler{D,T}<: AbstractIntegrator
    accelerations::Vector{SVector{D,T}}
end
function symplectic_euler_step(p::Particle, acceleration, dt)
    new_velocity = p.velocity + acceleration * dt
    new_position = p.position + new_velocity * dt
    return Particle(new_position, new_velocity, p.mass)
end
function integrate!(system::System{D,T}, force_calc,
             integrator::SymplecticEuler{D,T}, dt::Float64) where {D,T}

    compute_acceleration!(integrator.accelerations,system,force_calc)
    system.particles .= symplectic_euler_step.(system.particles,
                                integrator.accelerations, dt)
end
```

We can now ask: if we use (e.g.) Eq. (9.6) to go forward by one time step, what operation would return the system to *exactly* where it started? We can answer this by simply rearranging those equations: the inverse operation is

$$\begin{aligned} q_n &= q_{n+1} - \Delta t T'(p_{n+1}) \\ p_n &= p_{n+1} + \Delta t V'(q_n) \end{aligned} \tag{9.8}$$

That is, the inverse operation of Eq. (9.6) is just Eq. (9.7), but with $\Delta t \to -\Delta t$. Adopting the notation where $\phi_{(1)}$ refers to the operator that carries out one step of the first formulation, Eq. (9.6), we have found that the *adjoint* of that operator – the operator $\phi^*_{(1)}$ which reverses the order of all operations in the operator and reverses the direction of time – is the inverse of the operator:

$$\phi^{-1}_{(1)}(\Delta t) = \phi_{(2)}(-\Delta t) = \phi^*_{(1)}(\Delta t). \tag{9.9}$$

This is, in fact, the general requirement for a time-reversible operator.

This time-reversibility is a crucial first step – we have made sure that our integrator respects a fundamental symmetry of the underlying physics, and this gives us a method which is inherently more stable than the forward Euler method when simulating Hamiltonian systems. But is this property alone sufficient to *guarantee* the long-term energy conservation we desire?

9.2.2 The Velocity Verlet algorithm

Before answering that question, it's worth noting that while the symplectic Euler method is an improvement on forward Euler, there is an immediate, essentially "free" improvement we can make to it. Knowing that the most computationally expensive part of a particle-based system is usually the calculation of forces, we can do a very mild version of the RK trick and split the timestep in a way that looks very symmetric. The algorithm is now commonly called the "velocity Verlet" (after Loup Verlet's work on molecular dynamics in the 1960's [30]) or "Størmer-Verlet" algorithm (after Størmer's work in 1907 studying particles moving in a magnetic field [31]), but it was used by Delambre in 1791 [32] to calculate astronomical tables, and (!) by Newton in his proof of Kepler's second law [33].

Here's what that algorithm looks like. We perform the following three-step waltz for each particle in every timestep:

$$(1) \quad \vec{p}_i(t + \frac{\Delta t}{2}) = \vec{p}_i(t) + \frac{\Delta t}{2}\vec{F}_i(\mathbf{q}(t)) \tag{9.10}$$

$$(2) \quad \vec{q}_i(t + \Delta t) = \vec{q}_i(t) + \frac{\Delta t}{m}\vec{p}_i(t + \Delta t/2) \tag{9.11}$$

$$(3) \quad \vec{p}_i(t + \Delta t) = \vec{p}_i(t + \frac{\Delta t}{2}) + \frac{\Delta t}{2}\vec{F}_i(\mathbf{q}(t + \Delta t)) \tag{9.12}$$

Notice that the result of the force calculation at step (3) of this update is the same force needed in step (1) of the next iteration. Thus, as long as we set up a stateful integrator and initialize it properly, this pattern still only requires one force calculation per Δt.

```
struct VerletIntegrator{D,T} <: AbstractIntegrator
    accelerations::Vector{SVector{D,T}}
end
function VerletIntegrator(system::System{D,T},force_calculator) where
{D,T}
    initial_accelerations = zeros(SVector{D,T},
length(system.particles))
    compute_acceleration!(initial_accelerations, system,
force_calculator)
    return VerletIntegrator(initial_accelerations)
end
```

The combination of simplicity, computational efficiency, and (as we'll see) excellent stability has made this the go-to algorithm for simulating particles and planets evolving under Hamiltonian dynamics for literally centuries.

9.3 Symplectic integrators and energy conservation

To understand the origin of that excellent stability, we turn for a moment to a more formal operator formulation of Hamiltonian dynamics.

9.3.1 The Liouvillian

For any function $\rho(\mathbf{q}, \mathbf{p})$ of classical phase space variables, the evolution of that function governed by a Hamiltonian $\mathcal{H}$ is given by the Poisson bracket:

$$\frac{d\rho}{dt} = \{\rho, \mathcal{H}\} = \sum_i^N \frac{\partial \rho}{\partial \vec{q}_i}\frac{\partial \mathcal{H}}{\partial \vec{p}_i} - \frac{\partial \rho}{\partial \vec{p}_i}\frac{\partial \mathcal{H}}{\partial \vec{q}_i} \tag{9.13}$$

For the purposes of writing down a formal solution, we define the Liouvillian operator, $L_{\mathcal{H}}\rho = \{\rho, \mathcal{H}\}$. This lets us cast the time evolution as a simple equation with the formal solution:

$$\frac{d\rho}{dt} = L_{\mathcal{H}}\rho \quad \Rightarrow \rho(t) = e^{tL_{\mathcal{H}}}\rho(t=0). \tag{9.14}$$

Restricting our attention to Hamiltonians of the form $\mathcal{H} = T(\mathbf{p}) + V(\mathbf{q})$. The properties of the Poisson bracket mean that we can decompose the Liouvillian:

$$L_{\mathcal{H}}\rho = (L_T + L_V)\rho = \{\rho, T\} + \{\rho, V\}. \tag{9.15}$$

Each of these parts are exactly solvable, albeit only accounting for part of the system dynamics.

Sadly, the evolution operator isn't $L_{\mathcal{H}}$ but $e^{tL_{\mathcal{H}}}$, and the Baker-Campbell-Hausdorff (BCH) formula tells us that we are not allowed to just split the evolution into a part that talks only to the positions and a part that talks only to the momenta. In particular, [34]:

$$e^X e^Y = e^Z, \text{ where } Z = X + Y + \frac{1}{2}[X,Y] + \frac{1}{12}\left([X,[X,Y]] - [Y,[X,Y]]\right) + \cdots. \tag{9.16}$$

This formula tells us, for instance, that evolving the system by $Y = \Delta t L_T$ and then $X = \Delta t L_V$ (a la Eq. (9.6)) is different from evolving the system by the true $\Delta t L_{\mathcal{H}}$ by an amount proportional to Δt^2 and the commutator of L_T and L_V.

In this language, we can write our velocity Verlet algorithm as this propagator:

$$\phi(\Delta t) = e^{\frac{\Delta t}{2}L_V} e^{\Delta t L_T} e^{\frac{\Delta t}{2}L_V}. \tag{9.17}$$

An explicit calculation – applying BCH first with $X = A\Delta t/2$, $Y = B\Delta t$ and then with the result of that as the new X and $Y = A\Delta t/2$ – shows that this symmetric splitting of the operator cancels the coefficient of the Δt^2 term, approximating the true evolution operator to $\mathcal{O}(\Delta t^3)$ in the exponential. This fact, in combination with still only needing one force computation per timestep, is why velocity Verlet is essentially universally preferred to the symplectic Euler approach.

9.3.2 Backward error analysis and the shadow Hamiltonian

We've now seen that the velocity Verlet algorithm is both time-reversible and second-order accurate... but it still corresponds to an evolution operator which is only an approximation of the true operator, so you might complain that we still haven't explained why the algorithm should do a particularly good job at conserving energy. The final piece of the puzzle comes from an area of study known as backward error analysis [24].

The core idea is to ask: if our algorithm doesn't perfectly solve the original Hamiltonian, might it perhaps *perfectly solve a different but related Hamiltonian?* For a symplectic integrator like Verlet, the answer is, indeed, yes. It can be shown that the algorithm exactly conserves a nearby "shadow Hamiltonian[88]."

The math is mildly tedious, and each different integration scheme requires a separate analysis, but it is straightforward to mechanically carry out. In the case of the velocity Verlet algorithm, one finds [24, 35] that it rigorously conserves

$$\tilde{\mathcal{H}} = \mathcal{H} + \Delta t^2 \mathcal{H}_2 + \mathcal{O}(\Delta t^4). \tag{9.18}$$

That is: it conserves a Hamiltonian which is *very close* to the actual Hamiltonian of interest. The error term is

$$\mathcal{H}_2 = \frac{1}{12}\{T,\{T,V\}\} + \frac{1}{24}\{V,\{T,V\}\}. \tag{9.19}$$

We can evaluate these terms – the Poisson bracket $\{T,V\} = \sum_i (\vec{p}_i/m)\vec{F}$, and the Poisson brackets of T and V with that are, in turn

$$\{T,\{T,V\}\} = -\sum_i^N \frac{p_i^2}{m^2}\frac{d\vec{F}_i}{d\vec{q}_i} \tag{9.20}$$

$$\{V,\{T,V\}\} = -\sum_i^N \frac{F_i^2}{m} \tag{9.21}$$

These have a nicely interpretable physical meaning. Equation (9.20) involves gradients of the forces, capturing how they vary across space; Eq. (9.21) is proportional to the square of the forces, representing the effect of strong interactions. The shadow Hamiltonian is thus perturbed away from the true one by both the magnitude and curvature of the potential energy landscape.

The proof of the pudding is in the eating [36]. Figure 9.2 once again shows a simulation of the Solar System – this time over ten millenia – comparing the workhorse RK4 algorithm with the symplectic Euler and velocity Verlet algorithms. As before, the RK4 simulation has small relative errors in the total energy of the system, but those errors grow linearly, inexorably, with time. In contrast, both symplectic methods have some error in the total energy, but the energy of the system stays consistently *close* to that of the true system: we've devised schemes that rigorously conserve *a* Hamiltonian, and results like Eq. (9.18) tell us that the conserved Hamiltonian is closely related to the Hamiltonian we actually care about.

[88] Sounds like the original Hamiltonian's evil doppleganger.

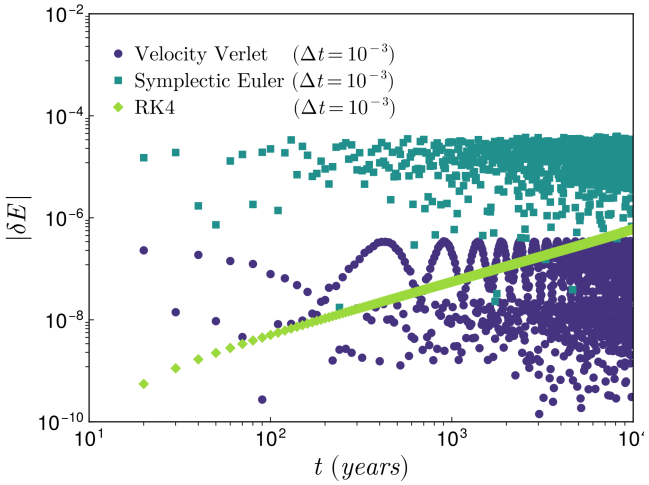

Figure 9.2: The relative error, δE, in numerically computing the total energy of our solar system simulation over the course of ten thousand years relative to its state when the simulation was started. Data correspond to using the symplectic Euler, velocity Verlet, and RK4 algorithms, all using the same Δt.

Chapter 10

Molecular Dynamics

Chapters 8 and 9 have given us a solid foundation for studying ordinary differential equations. We have our toolbox of different integration methods with their different pros and cons, and we have a modular computational structure that will let us apply that toolbox to a breathtaking range of problems in the physical sciences. Indeed, the language of ODEs is ubiquitous: we could study classical mechanical systems governed by Newton's laws, charges in an electrical circuit, solutions to the Schrodinger equation in time-independent potentials, the ecosystem dynamics of competing species, networks of chemical reactions, the expansion equations governing the entire universe... It's all at our fingertips!

In this coda to Module II, we'll focus on a particular example –classical simulations of the movements of atoms, molecules, and course-grained "particle" more generally – to see how the specific details of a problem can shape the specific algorithms and approaches we use to adapt our more general structure. We will, again, just scratch the surface of the full complexity and power of these "molecular dynamics" simulations; I recommend Ref. [5] as a good place to start diving into increasingly interesting details.

10.1 An N-body problem in a box

Our goal will be to simulate and study the properties of a macroscopic, "bulk" material – a liquid or gas containing something like Avogadro's number of particles. Doing this directly would be absurd: our computers run at only gigahertz speeds, so even just asking the computer to look at (let alone store in memory) the position of so many particles would take of order a million years. Instead we will simulate much smaller systems containing perhaps $10^3 - 10^6$ particles[89]. But how could such a tiny system replicate the behavior of one that is, in comparison, effectively infinite?

The validity of this approach relies on a key physical principle: in most bulk systems, interactions are *local*. That is, the behavior of a given particle is dominated by its interactions with

[89] An early but important paper in the literature on molecular dynamics was published in 1959 [13], which optimistically noted that "[c]omputers now being planned should be able to handle ten thousand molecules..." We can, obviously, simulation much larger systems, but it is important to remember that we can often extract all of the physical information we want from these smaller systems. Brute force and simply scaling up to ever larger sizes is not always the winning strategy.

its immediate neighbors, with far away particles typically having a negligible influence. This gives rise to a characteristic correlation length, ξ, which the distance over which particle positions and motions are meaningfully correlated. As long as we make our "simulation box" much larger than this correlation length, the particles in the center of the box will behave almost precisely as they would in the true bulk system, blissfully oblivious to the distant boundaries. This leaves of with a critical, immediate problem to solve: what to do about those particles at the boundaries of our (relatively) small simulation?

10.1.1 Periodic boundary conditions

We could let them interact with an artificial wall we build into the simulation to keep everything contained, or let them interact with an empty vacuum, effectively simulating a tiny isolated droplet rather than the behavior of a bulk system. If our interest is at small systems at the nanoscale this may well be the correct thing to do, but not if we are interested in understanding bulk properties: in such systems a huge fraction of the particles are reasonably close to the surface, where their physical behavior (their structural arrangements, their dynamics, their pressure, etc) is completely different from the particles in the interior.

The standard solution is to eliminate either free or confining surfaces entirely by imposing periodic boundary conditions (PBCs) on the simulation box [37], as illustrated in Fig. 10.1. The idea is to imagine our simulation box (our "primary unit cell") surrounded on all sides by an infinite lattice of identical copies of itself. Each particle in our simulation represents not a single entity, but an infinite set of periodic images. For example, when a particle leaves the primary unit cell by traveling across one face, an image of it is simultaneously entering the primary unit cell through the opposite face with the same velocity. This creates a system that is finite in size – and for which we only need to actually keep track of the finite set of particles in the primary unit cell – but which has no edges or surfaces.

Using PBCs has a crucial algorithmic consequence for how we compute the distance between particles, known as the *minimum image convention*. Since each particle represents an infinite set, when we compute the distance (or the forces) between " particles i and j" what do we actually mean? Rather than summing an infinite set of forces and calculating an infinite set of interparticle distances[90], we only need to find the distance between the *closest perdiodic image of the pair,*calculating the forces based on that interaction alone.

Implementing the minimum image convention requires "wrapping" the separation vector between two particles into the dimensions of the central box – along any coordinate it is impossible for particles to be separated by more than the linear size of the simulation box in that direction. Thus, for a cubic box of side length L, the raw separation vector $\Delta x = x_i - x_j$ between particles in the unit cell is adjusted, as we can see visually in Fig. 10.1. If $\Delta x > L/2$, the closest image of particle j is actually in the neighboring box in the negative direction, and the minimum image separation is actually $\Delta x - L$. Similarly, if $\Delta x < -L/2$ the minimum image separation is $\Delta x - L$. In code, this might look like

[90] A bit impractical, at least in this real-space formulation

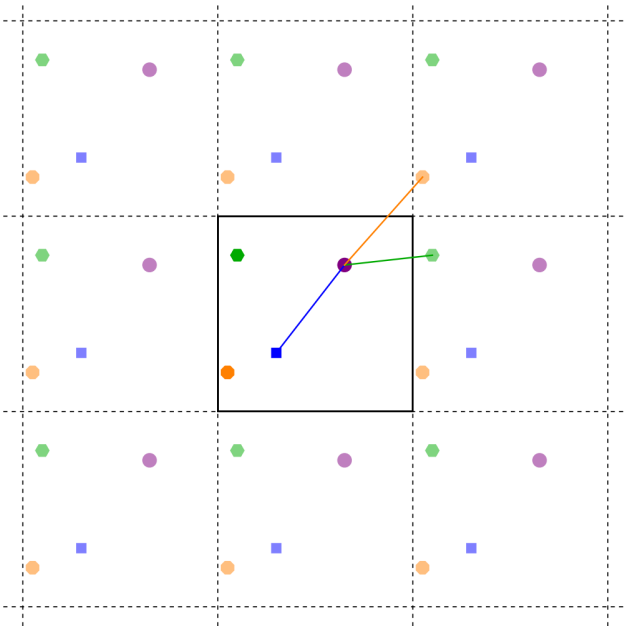

Figure 10.1: A representation of periodic boundary conditions and the minimum image convention in a two dimensional system. Different particles in the primary unit cell are indicated with shapes and full opacity colors, whereas their periodic images have reduced opacity. The minimum image distance between the purple circular particle and its nearest neighbors is shown with solid lines: the shortest separation vector may be contained entirely within the primary unit cell, or it may cross any number of faces.

```
# A simple, explicit implementation
function minimum_image_distance(dx, L)
    if dx > L / 2
        return dx - L
    elseif dx < -L / 2
        return dx + L
    else
        return dx
    end
end

# A vectorized version that works for a vectors `dr` and `L`
function minimum_image_vector(dr, L)
    return dr .- L .* round.(dr ./ L)
end
```

So: when calculating forces we first compute this minimum image separation vector, and then use that to determine distances and directions/magnitudes of forces. This ensures that every particle interacts with its “true” nearest neighbor in the infinite system we are modeling.

10.1.2 Interparticle potentials

With our simulation box defined, we now need to specify the physical interactions that will govern our particles. We will focus on central pairwise interactions, for which we can write an interparticle potential that depends only on the relative separation between two particles, $V(r)$, from which the force is $\vec{F} = -\nabla V(r)$. The choice of V is a modeling decision that depends on the physical system. For charged particles like ions, we might use the long-range Coulombic potential; for gravitationally interacting planets we might use Newton's universal law of gravitation. For many neutral atoms and molecules, and for sterically interacting mesoscale "particles" like colloids, the interactions are effectively short ranged. This is the case we'll focus on, and we'll explore the algorithmic optimizations that can be applied when one knows the interactions act only over a finite distance.

The canonical model for neutral atoms and molecules is the Lennard-Jones potential [38],

$$V_{LJ}(r) = 4\varepsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right). \tag{10.1}$$

This potential, illustrated in **??**, is a simple but extremely effective model for noble gases, but it also more generically captures two characteristic features of atomic interactions: a harsh short-ranged repulsion (due, e.g., to Pauli exclusion) and a weaker but more long-ranged attraction (due, e.g., to van der Waals forces). The parameters ε and σ characterize the energy and length scales in the interaction, respectively. Because the LJ potential decays rapidly, the force between distance particles is negligible. This allows us to make a very pragmatic approximation: we introduce a cutoff radius, r_c, and write

$$\tilde{V}_{LJ}(r) = \begin{cases} V_{LJ}(r) & \text{if } r < r_c \\ 0 & \text{if } r > r_c \end{cases} \tag{10.2}$$

This harsh truncation introduces small discontinuities in both the energy and the force, and one can implement models that smooth these discontinuities over [5], but for our purposes it will be sufficient.

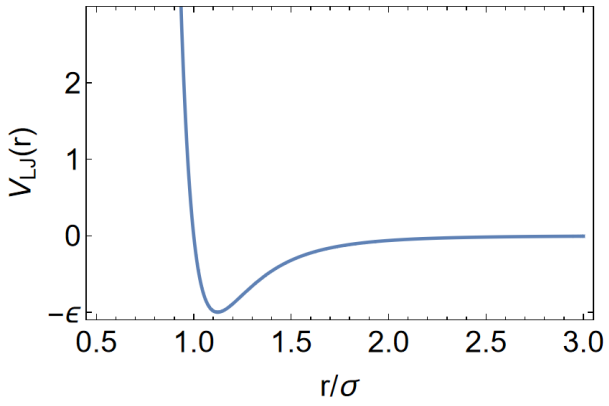

Figure 10.2: A (placeholder) plot of the Lennard-Jones potential.

An alternate short-ranged family of models often used in the study of sterically interacting

particles is based on the overlap between particles of some size:

$$\tilde{V}_s(r) = \begin{cases} \varepsilon\left(1 - \frac{r}{\sigma}\right)^\alpha & \text{if } r < \sigma \\ 0 & \text{if } r > \sigma \end{cases} \tag{10.3}$$

Here $\sigma = r_c$ and ε again set the length and energy scales, and α parameterizes the steepness of the interaction; $\alpha = 2$ corresponds to harmonic repulsions, $\alpha = 5/2$ corresponds to Hertzian repulsion, etc.

10.1.3 Initial conditions

Our simulation also requires an initial state: how will we assign the positions and velocities of our N particles? Unlike the Solar System in Chapter 8, we cannot simply look up the positions of particles in a fluid in a convenient NASA database; instead we must generate plausible starting configurations that represent the system at a desired density and temperature.

This can be surprisingly fussy. The velocities are relatively straightforward: we can generate random numbers so that each component of the velocity of each particle is drawn from a Maxwell-Boltzmann distribution corresponding to our target temperature. Assuming we have the facilities to generate Gaussian numbers with zero mean and unit variance, we simply take those results and scale them so that each velocity component has zero mean and variance $k_B T/m$. After assigning these random velocities, we should be careful to calculate the total momentum of the system and subtract the center-of-mass velocity from each particle – this ensures the system as a whole has zero net momentum and will not drift through the periodic box during our simulation.

The positions require a bit more care. One approach is to place particles at positions that are uniformly randomly distributed through throughout the box. This is easy to do, but will almost certainly result in pairs of particles that happen to be placed very close to each other[91]. If we are using steric repulsions this won't cause too many issues, but if we are using LJ interactions this would seed the system in an initial state of massively high potential energy and hence subject our system to enormous repulsive forces. Alternate strategies include Poisson disk sampling [39] or "soft push-off" initialization [40] – these allow the initial state to be disordered but not uniformly random – or seeding particles on the sites of a regular crystal lattice (cubic, FCC, etc) at the desired density. All of these other approaches have benefits and drawbacks, depending on the eventual target of the simulation.

10.2 Force calculations for short-ranged interactions

With our physical setup defined, we can turn to the computational heart of every molecular dynamics simulation: the calculation of the forces. At every time step we must calculate the net force on every particle; we implemented the most direct method in the `BruteForceCalculator` of Chapter 9. It simply loops through all unique pairs of particles, calculates their (minimum

[91] The expectation value of the minimum distance between all pairs of N points placed uniformly randomly in a d-dimensional hypercube of side length L is $\langle r_{\text{min}} \rangle \sim LN^{-2/d}$ – much smaller than the expectation value of the distances themselves.

image) separation, and sums the forces. This approach is both simple and guaranteed to be correct, but for a system of N particles it requires checking each of the $N(N-1)/2$ pairs. That is, it's complexity scales as $\mathcal{O}\left(N^2\right)$.

Is that scaling a real problem, or just a theoretical concern? The answer depends on precisely what we want to do, but let's perform a quick Fermi estimate. A single Lennard-Jones force calculation involves subtracting vectors, computing the magnitude of the minimum image separation, and then some divisions and multiplications. It can be hard to reason precisely about floating point operations per second on modern hardware[92], but we can run a quick benchmark on a modern CPU core and find that it takes, say, 50 nanoseconds to execute a single LJ force calculation. Our processors run at gigahertz speeds; if we want to calculate the interactions for all pairs of forces between, say, $N = 10^5$ particles, it will take of order $\sim (5 \times 10^{-8}$ s/pair$) \times (5 \times 10^9$pairs$) \approx 250$ s. For *one* timestep.

A simulation long enough to observe the diffusion of molecules might require a million timesteps, and at this rate our "modest" simulation would take almost eight years to complete. The brute-force approach is not just inefficient – for non-trivial systems it is simply not viable. Fortunately, the short-ranged nature of our potentials unlocks much faster algorithms for our use.

10.2.1 Neighbor list structures

While the brute-force approach may be a dead end, our physical intuition might provide an escape. For the short-ranged potentials described above, each particle only feels a force from its nearby neighbors, and so we spend the overwhelming majority of our computational effort checking the distance between the $\mathcal{O}\left(N^2\right)$ pairs that are too far apart to interact. This is a practical application of the complexity analysis from Section 7.3: by identifying an algorithms inefficiency, we can try to find a better one.

A solution in this case is to build a *neighbor list*, a data structure that will allow us to more quickly identify which particles are close enough to potentially interact with each other. It involves a classic trade-off: we accept a more complex implementation and an increase in memory usage in exchange for a dramatic improvement in time complexity. One of the most common (and intuitive) neighbor list structures is the cell list [41]. It an its corresponding algorithm work in two stages.

First is the *binning* stage. We divide up the d-dimensional simulation box into a grid of smaller hyperrectangular cells, where crucially every side length of these small cells is at least as large as the force cutoff radius r_c. This guarantees that the neighbors of a particle can only reside in the same cell as the particle itself or one of the immediately adjacent $3^d - 1$ cells. We then perform an $\mathcal{O}(N)$ sweep over the particles, placing each particle's index into the appropriate cell.

Second is the actual force calculation. For each of the N particles, we refrain from checking the distance to each of the other $N-1$ particles; we instead only check particles in the same or adjacent cells of the target particle. For a system of roughly uniform density, the number of

[92] Modern CPUs can execute multiple instructions at once, use micro-operations to effectively amortize the high cost of otherwise "slow" operations light division, can vectorize similar operations as we compute multiple particle pairs at the same time, and so on.

particles per cell cells will on average be a constant. Since the number of neighboring cells is also a constant, we suddenly find that we can reduce the complexity of the force calculation from $\mathcal{O}\left(N^2\right)$ to $\mathcal{O}\left(N\right)$. The prefactor is, of course, important – for sufficiently small systems it can be slower to use a cell list than the brute force system – but for large systems this is an inevitable improvement.

To implement this in the context of our general, d-dimensional system, we can first create a `CellList` object that stores the grid of cells, along with a function that does the necessary assignment of particles at each step.

With this data structure in hand, we can define a new `CellListCalculator` the plugs directly into our existing simulation framework. The `compute_acceleration!` method for this new type will first update the cell list, and then use that structure to perform the more algorithmically efficient force calculation. This is part of the continuing payoff of our modular design: by creating a new `AbstractForceCalculator` type, we can swap out the fundamental force calculation without changing a line of the other parts of our code. Moreover, we have a built in set of natural tests: we can compare the results of this newer method directly with our older `BruteForceCalculator` approach.

10.3 Thermodynamic ensembles and equilibration

We're making serious progress in our efforts to simulate the properties of bulk physical systems: we have symplectic integration methods that will conserve energy, we have relevant force laws for inter-particle interactions, and we have boundary conditions and algorithmic solutions that will let us simulate reasonably large numbers of particles in a reasonable amount of time. The is a set up that corresponds to conserving the number of particles, N, the volume of our simulation domain V, and the energy of our system, E – in the language of thermodynamics this is the "NVE ensemble".

Perhaps a small, niggling doubt enters your mind at this point, as you realize that you've never actually carried out an experiment in which you have specified *precisely*, down to the electron volt, exactly how much energy is contained in the system you are studying. Much more commonly, you have down your best to control the *temperature* of your system – you do this by connecting your system to a much large "reservoir" whose temperature you know. By doing so, you let the energy of the system itself fluctuate as it trades energy back and forth with your system, even as it maintains a constant temperature. This corresponds to the "NVT" ensemble – can we accommodate this kind of physics in our simulations?

10.3.1 Computational thermostats

Of course we can. We'll mimic the effects of a thermal reservoir by introducing "thermostatting algorithms." Thermostats are modifications to our simulation that allow energy to flow into or out of the system, steering towards a target temperature while trying to preserve the correct statistical properties of the NVT ensemble. There are two fundamentally different approaches we can take here, differing both in how they are implemented and what properties of the system under study they want to preserve.

Stochastic thermostats

Stochastic thermostats try to explicitly model the random interactions the system might have with the heat bath – a molecule in the surface of the system has a chance collision with the reservoir, suddenly experiencing a change in momentum. The most intuitive example of such a thermostat was proposed by Andersen [42]: it imagines that every particle occasionally undergoes such a chance collision with a fictitious particle from the reservoir, which instantly thermalizes the particle to the bath's temperature.

This is straightforward to implement: we perform standard integration steps (e.g., with the velocity Verlet algorithm) for some number of steps. At regular intervals, we select a small fraction of the total number of particles at random; for each selected particle we discard its current velocity and replace it with one drawn from the Maxwell-Boltzmann distribution corresponding to the target temperature. This can be as simple as code block 10.1.

```
using Random
function andersen_thermostat!(sys::System{D,T}, temperature::T,
                probability::Float64) where {D,T}
    for i in eachindex(system.particles)
        if rand() < probability
            p = system.particles[i]
            sigma = sqrt(temperature / p.mass)
            new_v = randn(SVector{D,T}) * sigma
            system.particles[i] = Particle(p.position, new_v, p.mass)
        end
    end
    return nothing
end
```

Code block 10.1: A simple implementation of an Andersen thermostat. Note that `randn` generates random numbers drawn from a zero mean unit variance Gaussian, and that we are working in a system of units where $k_B = 1$.

With a reasonable choice of the probability of selecting particles and the frequency of performing this operation, the Andersen thermostat is a robust way of driving the system to equilibrium and maintaining a constant temperature. Its fundamental drawback is that it is *non-deterministic* and it breaks the true dynamics of the system. By occasionally assigning random velocities, the trajectories we observe are no longer continuous solutions to Newton's equations. This means that while this approach (and those using other stochastic thermostats) are excellent for sampling the static equilibrium properties of our system – what are its bulk elastic properties? how are particles typically arranged with respect to each other at the microscopic scale? – it is unsuitable for measuring dynamical properties.

Deterministic thermostats

An alternative to the stochastic approach is a deterministic, "extended Hamiltonian" thermostat, and perhaps the most famous and widely used version is the Nosé-Hoover (NH) thermostat

[43, 44]. The core idea is a clever abstraction of a physical heat bath Rather than simulating 10^{23} particles in an actual reservoir, the basic NH thermostat models its entire effect with a single extra degree of freedom. The "thermal piston" has a fictitious mass Q that represents the reservoir's thermal inertia and a velocity ζ that acts as a time-dependent "friction coefficient" that acts on the particles in the system.

The equations of motion for the real system are modified to include this friction term:

$$\frac{d\mathbf{r}_i}{dt} = \frac{\mathbf{p}_i}{m_i}$$
$$\frac{d\mathbf{p}_i}{dt} = \mathbf{F}_i - \zeta \mathbf{p}_i$$

The equations of motion for the thermostat degrees of freedom are:

$$\frac{d\zeta}{dt} = \frac{1}{Q}\left(\sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{m_i} - gk_BT\right).$$

Here g is the number of degrees of freedom in the system, and the mass Q determines the timescale of the coupling between the system and the reservoir. A large value of Q corresponds to a slow, weakly-coupled thermostat that lets the system's temperature fluctuate over long times. A small Q corresponds to a fast thermostat that very tightly controls the system's temperature, but which can introduce artificial high-frequency oscillations into the system. Choosing an appropriate value of this parameter is a key part of setting up a stable NVT simulation.

If the system gets too hot (i.e., its kinetic energy grows too large), ζ increases, and this increased "friction" cools the system; if the system is too cold, ζ becomes negative and accelerates the particles in the system, heating everything back up. The result is a set of *deterministic*, time-reversible equations for the motion of this extended system-plus-thermostat. Implementing the NH thermostat is certainly more complex than implementing the Andersen thermostat[93], but it is a proper tool for studying dynamical properties in the NVT ensemble. This is because the extended system not only samples the correct equilibrium structure of our physical system; it samples the correct dynamical trajectories that particles at constant temperature might take.

Other Ensembles

The idea of extending the system you are simulating with fictitious degrees is a quite general and powerful one. A very similar approach can be used to construct not only sophisticated thermostats but also *barastats*, which control the system's pressure, P, by allowing the entire shape and volume of the simulation box to fluctuate. Combining thermostats and barostats allows for simulations in the NPT ensemble, which most closely mimics standard lab experiments.

The Nosé-Hoover equations may seem cleverly constructed, but they are not ad hoc. They are a practical reformulation of the dynamics derived from a conserved Hamiltonian for an extended system. The original formulation by Nosé was

$$\mathcal{H}_{\text{Nosé}} = \sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2s^2 m_i} + V(\{\mathbf{q}_\mathbf{i}\}) + \frac{p_s^2}{2Q} + gk_BT\log(s). \tag{10.4}$$

[93] Especially when wants to include it in an explicitly reversible, symplectic time evolution scheme [45].

In that expression, s is a dynamic time-scaling variable (the thermostat's "position") and p_s is its conjugate momentum. While working with this Hamiltonian and its "virtual time" is more complex, the existence of this conserved quantity is the fundamental reason that the NH thermostat is stable, generates the correct NVT ensemble, and possesses a Hamiltonian structure that allows it to be integrated with a symplectic algorithm.

10.4 Observables and measurements

We have finally assembled the components for a complete, efficient[94], and flexible simulation engine. We can model a physical system, choose an appropriate force law, an integrate its equation of motion using physically motivated ODE solvers in multiple thermodynamic ensembles. All that is left for us to do? In the immortal words of XKCD author Randall Munroe: "Stand back – I'm going to try SCIENCE."

How do we bridge the gap between the microscopic trajectories of individual particles that we can trace in our simulations and the macroscopic, measurable properties of the material we are simulating? Much of the answer to that is the focus of statistical physics, but below we will cover the fundamental methods for extracting physical meaning from our simulations.

10.4.1 Equilibration

Before we measure *anything*, we have to address a crucial artifact of our simulation's setup. Our initial conditions – whether those were on a perfect crystalline lattice or a completely random placement of particles – is an artificial state. It is not representative of the natural, equilibrium configuration of, e.g., a fluid that has been sitting at constant temperature for a long time. If we start measuring the properties of our system immediately, our results will be contaminated with the relaxation of our simulation from the initial artificial state to its more typical steady state.

The solution is the same as in a laboratory experiment: after having prepared our system in an unusual way, we must wait for it to settle down and equilibrate. This involves integrating the system forward in time during an "equilibration" or "burn-in" time – this allows the particles to interact, exchange energy, and eventually settle into a statistical steady state that is characteristic of the target ensemble. After this initial transient period is over we can begin a "production run," during which we continue simulating our system, collecting data to be included in our eventual analyses.

How long is long enough for this equilibration phase? The standard answer is to monitor the various quantities you eventually hope to measure – perhaps the total potential energy, or the diffusion constant of the particles. You will observe that as you start measuring from the very start of the simulation, these quantities will show a systematic drift as the system relaxes to equilibrium. As an example, if you start a simulation of a fluid with uniformly random particle positions (i.e., the kind of positions you would expect to see in an ideal gas), you will find that

[94] We have deliberately avoided one of the key topics in modern computational research: writing code that makes efficient use of parallel computing resources. There have been other places where we prioritized clarity and pedagogy over a maximally performant implementation, but for the most part we have written very solid albeit single-threaded code.

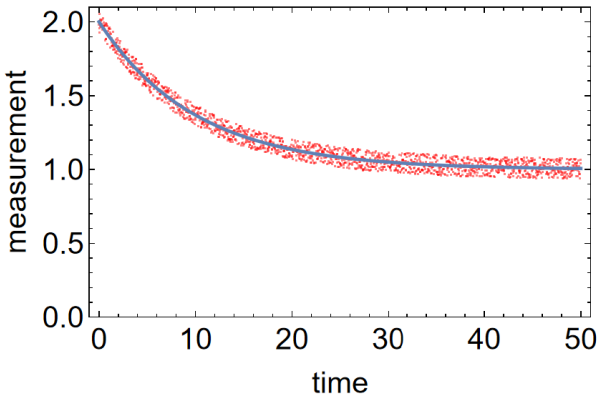

Figure 10.3: A (placeholder) plot showing the equilibration of a system. We have a noisy measurement of some property, and we are able to fit its decay to some plateau value with an exponential function. Our "equilibration" phase is many multiples of the time constant in that exponential fit. This plot is an idealization, and will be replaced with an actual example from a simulation.

the total potential energy starts at a very high value but then steadily decreases, eventually plateauing and then fluctuating around a steady-state value. Ideally, these quantities show an exponential decay from the initial values to their steady state values: this exponential decay gives you a time scale over which the system relaxes to equilibrium, and you want to make sure your equilibration phase is several multiples of this timescale. An example of this is shown in Fig. 10.3

10.4.2 From microscopic trajectories to macroscopic properties

With our system in a statistically steady state, we can begin our production runs. A core principle of statistical mechanics tells us that by averaging observables over a long enough trajectory, we can calculate the macroscopic thermodynamic properties of the system.

Thermodynamic properties

The most fundamental observables are the state variables (or "thermodynamic coordinates") that define the ensemble we are working in. Consider, for instance, our simulations of the NVT ensemble – how could we verify that our thermostat is doing its job? One definition [46] of the temperature of a system relates T with the instantaneous kinetic energy:

$$\langle KE \rangle = \left\langle \sum_{i=1}^{N} \frac{p_i^2}{2m} \right\rangle = \frac{g}{2} k_B T, \tag{10.5}$$

where g is the total number of momentum degrees of freedom (e.g., dN for N point particles in d dimensions). By calculating an "instantaneous kinetic temperature" at each time step, we could determine T for our system and verify that our thermostat is working as desired.

We can also measure state variables that are conjugate to the ones we control. For example, if we are working at fixed V, we could measure the pressure, P, of the system. This pressure

arises from the complex interplay between the motion of the particles and the forces between them; in equilibrium it can be calculated via the virial theorem:

$$P = \frac{Nk_BT}{V} + \frac{1}{V}\left\langle \frac{1}{d}\sum_{i<j} \mathbf{F}_{ij} \cdot \mathbf{r}_{ij} \right\rangle. \tag{10.6}$$

Here the first term is the familiar ideal gas pressure; the second "virial" term is the contribution due to interparticle interactions, and it can be calculated by averaging the dot product of the force and separation vectors for all interacting pairs of particles[95].

Structural properties

Beyond measuring simple thermodynamic variables, MD simulations give us a direct window into the microscopic structure of matter. One of the most important ways of quantifying that structure is the "radial distribution function," $g(\vec{r})$, which describes how the density of particles varies as a function of relative separation from a reference particle.

For an isotropic system, $g(r)$ is the ratio of the average local density at a distance r from a reference particle, $\rho(r)$, to the bulk density $\rho = N/V$. In three dimensions:

$$g(r) = \frac{\rho(r)}{\rho} = \frac{V}{N}\frac{\langle N(r, \Delta r)\rangle}{4\pi r^2 \Delta r}. \tag{10.7}$$

Here, $\langle N(r, \Delta r)\rangle$ is the average number of particles found in a thin spherical shell of radius r and thickness Δr around any given particle. This can be computed in a simulation by building a histogram of all pairwise distances[96]

The importance of $g(r)$ is not just in the simple structural picture it gives us. It's Fourier transform is directly related to the static structure factor $S(k)$, which is precisely what is measured in x-ray and neutron scattering experiments:

$$S(k) = 1 + \rho \int e^{-i\mathbf{k}\cdot\mathbf{r}}(g(r) - 1), d^d\mathbf{r} \tag{10.8}$$

Measure testable observables

The fact that the radial distribution function is related to the structure factor reinforces a general important rule: Simulations correspond closely to experiments – ones we happen to running on a computer rather than with fancy physical instrumentation, but experiments nonetheless. As a consequence, the simulator should in general report and measure things that correspond to experiments and to testable hypotheses.

The shape of $g(r)$ is a direct structural fingerprint of the material's phase. In an ideal gas there are no correlations between particle positions, and $g(r) = 1$ for all r. For a crystal, $g(r)$

[95] Some care must be taken, as always, to account for periodic boundaries.

[96] Or by numerically differentiating the cumulative probability of observing particles separated by some distance – a method which is sometimes more robust.

is characterized by a series of sharp, well-defined peaks that correspond to the crystal lattice spacings. As shown in TODO, for a liquid, $g(r)$ shows a relatively sharp peak at the average nearest-neighbor distance, followed by decaying oscillations that represent subsequent "solvation shells" – an example of this is shown in Fig. 10.4

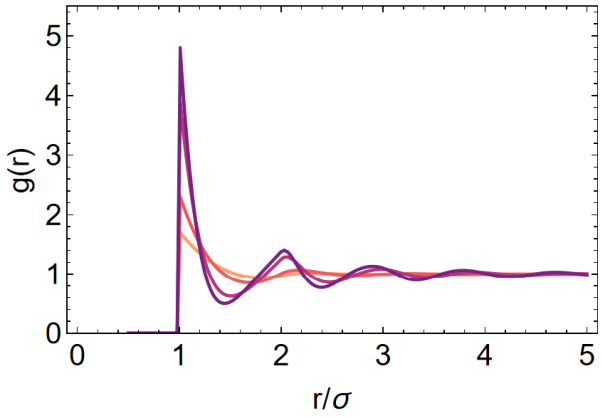

Figure 10.4: A (placeholder) plot of the radial distribution function of a fluid. Current data is for the Percus-Yevick solution of a hard sphere fluid at four different volume fractions (0.2, 0.3, 0.45, 0.5).

Dynamical properties

Finally, because MD generates particle trajectories, we can directly study how particles move over time. One of the most straightforward properties to measure is the *mean-squared displacement* (MSD). This measure how far, on average, particles have moved from a position at an earlier time:

$$\mathrm{MSD}(\Delta t) = \left\langle \frac{1}{N} \sum_{i=1}^{N} |\mathbf{r}_i(t_0 + \Delta t) - \mathbf{r}i(t_0)|^2 \right\rangle_{t_0}, \tag{10.9}$$

where the average, $\langle \ldots \rangle_{t_0}$, is taken over all starting times in the equilibrated trajectory.

Just as $g(r)$ is a fingerprint of the structure, the MSD is a fingerprint of the dynamics. In a solid, particles vibrate around fixed lattice sites, so the MSD plateaus at a small value after a short time. In a fluid, particles are free to move, and the MSD eventually grows linearly with time. This is described by the Einstein relation [47]:

$$\mathrm{MSD}(t) = 2dDt, \tag{10.10}$$

where d is the dimensionality of space and D is the diffusion coefficient. By calculating the MSD from our trajectories and finding its slope in the linear regime, we can directly measure this transport property of our simulated material.

10.5 Coda

Having built our simulation engine on the foundation of Chapters 8 and 9, and having filled it with the tools from this chapter, our basic toolbox is now complete. Our simulations are no long

just generic solutions to N-body problems: we can use them as a virtual laboratory to probe the properties of a material. We can set the thermodynamic coordinates (N, V, T), we can prepare a sample, let it reach equilibrium, and perform measurements of its fundamental properties. By measuring structure like $g(r)$ – which is itself connected to *many* other material properties – and transport coefficients like D, we can bridge the gap between the microscopic laws of motion and the macroscopic world of material science. By turning these tools on specific problems, we could already find ourselves pushing the boundaries of science outward.

And the code we have is already better than what I wrote as a postdoc.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

[2] Cristopher Moore and Stephan Mertens. *The nature of computation*. Oxford University Press, 2011.

[3] Werner Krauth. *Statistical mechanics: algorithms and computations*, volume 13. OUP Oxford, 2006.

[4] Daan Frenkel. Simulations: The dark side. *The European Physical Journal Plus*, 128:1–21, 2013.

[5] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*. Elsevier, 2023.

[6] Alex Gezerlis. *Numerical methods in physics with Python*, volume 1. Cambridge University Press Cambridge, UK, 2023.

[7] Kyle Novak. *Numerical Methods for Scientific Computing: The Definitive Manual for Math Geeks*. Equal Share Press, 2022.

[8] William Jones. *Synopsis Palmariorum Matheseos: Or, a New Introduction to the Mathematics*. J. Matthews for Jeff. Wale at the Angel in St. Paul's Church-Yard, 1706.

[9] William Oughtred. *Clavis Mathematicae denuo limita, sive potius fabricata*. Lichfield, 1631.

[10] Florian Cajori. *A history of mathematical notations*, volume 1. Courier Corporation, 1993.

[11] Tom Kwong. *Hands-On Design Patterns and Best Practices with Julia: Proven solutions to common problems in software design for Julia 1. x*. Packt Publishing Ltd, 2020.

[12] Lee Phillips. *Practical Julia: A Hands-on Introduction for Scientific Minds*. No Starch Press, 2023.

[13] Berni J Alder and Thomas Everett Wainwright. Studies in molecular dynamics. i. general method. *The Journal of Chemical Physics*, 31(2):459–466, 1959.

[14] Gregorii Aleksandrovich Galperin. Playing pool with π (the number π from a billiard point of view). *Regular and chaotic dynamics*, 8(4):375–394, 2003.

[15] F Chiappetta, C Meringolo, P Riccardi, R Tucci, A Bruzzese, and G Prete. Boyle, huygens and the 'anomalous suspension'of water. *Physics Education*, 59(4):045026, 2024.

[16] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.

[17] Greg Wilson, Dhavide A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven HD Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, et al. Best practices for scientific computing. *PLoS biology*, 12(1):e1001745, 2014.

[18] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K Teal. Good enough practices in scientific computing. *PLoS computational biology*, 13(6):e1005510, 2017.

[19] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

[20] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[21] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835. SIAM, 2024.

[22] Tony Freeth, Yanis Bitsakis, Xenophon Moussas, John H Seiradakis, Agamemnon Tselikas, Helen Mangou, Mary Zafeiropoulou, Roger Hadland, David Bate, Andrew Ramsey, et al. Decoding the ancient greek astronomical calculator known as the antikythera mechanism. *Nature*, 444(7119):587–591, 2006.

[23] Arieh Iserles. *A first course in the numerical analysis of differential equations*. Number 44. Cambridge university press, 2009.

[24] Ernst Hairer, Christian Lubich, and Gerhard Wanner. Structure-preserving algorithms for ordinary differential equations. *Geometric numerical integration*, 31, 2006.

[25] Leonhard Euler. *Institutiones calculi integralis*, volume 1. Impensis Academiae Imperialis Scientiarum, 1768.

[26] Carl Runge. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895.

[27] Wilhelm Kutta. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner, 1901.

[28] John C Butcher. Implicit runge-kutta processes. *Mathematics of computation*, 18(85):50–64, 1964.

[29] Hermann Weyl. *The classical groups: their invariants and representations*, volume 1. Princeton university press, 1939.

[30] Loup Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.

[31] Carl Størmer. Sur les trajectoires des corpuscules électriques dans l'espace sous l'action du magnétisme terrestre. *Archives des Sciences Physiques et Naturelles*, 24:5–18, 113–158, 221–247, 317–364, 1907. Published in four parts.

[32] Robert I McLachlan and G Reinout W Quispel. Splitting methods. *Acta Numerica*, 11:341–434, 2002.

[33] Isaac Newton. *Philosophiæ Naturalis Principia Mathematica*. Jussu Societatis Regiæ ac Typis Josephi Streater, Londini, 1687.

[34] Eugene Borisovich Dynkin. Calculation of the coefficients in the campbell-hausdorff formula. In *Dokl. Akad. Nauk. SSSR (NS)*, volume 57, pages 323–326, 1947.

[35] Mark E Tuckerman. *Statistical mechanics: theory and molecular simulation*. Oxford university press, 2023.

[36] William Camden. *Remaines of a greater worke, concerning Britaine, the inhabitants thereof, their languages, names, surnames, empreses, wise speeches, poësies, and epitaphes*. Printed by G. Eld for Simon Waterson, London, 1605.

[37] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[38] John Edward Jones. On the determination of molecular fields.—ii. from the equation of state of a gas. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 106(738):463–477, 1924.

[39] Robert Bridson. Fast poisson disk sampling in arbitrary dimensions. *SIGGRAPH sketches*, 10(1):1, 2007.

[40] Kurt Kremer and Gary S Grest. Dynamics of entangled linear polymer melts: A molecular-dynamics simulation. *The Journal of Chemical Physics*, 92(8):5057–5086, 1990.

[41] Roger W Hockney and James W Eastwood. *Computer simulation using particles*. IOP Publishing Ltd., 1988.

[42] Hans C Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *The Journal of chemical physics*, 72(4):2384–2393, 1980.

[43] Shuichi Nosé. A unified formulation of the constant temperature molecular dynamics methods. *The Journal of chemical physics*, 81(1):511–519, 1984.

[44] William G Hoover. Canonical dynamics: Equilibrium phase-space distributions. *Physical review A*, 31(3):1695, 1985.

[45] Glenn J Martyna, Mark E Tuckerman, Douglas J Tobias, and Michael L Klein. Explicit reversible integrators for extended systems dynamics. *Molecular Physics*, 87(5):1117–1157, 1996.

[46] Owen G Jepps, Gary Ayton, and Denis J Evans. Microscopic expressions for the thermodynamic temperature. *Physical Review E*, 62(4):4757, 2000.

[47] Albert Einstein. Über die von der molekularkinetischen theorie der wärme geforderte bewegung von in ruhenden flüssigkeiten suspendierten teilchen. *Ann. d. Phys.(Leipzig)*, 17:549, 1905.

[48] Stanley J Farlow. *Partial differential equations for scientists and engineers*. Courier Corporation, 1993.

[49] Sandro Salsa. *Partial differential equations in action*. Springer, 2016.

[50] Zhuoqiang Guo, Denghui Lu, Yujin Yan, Siyu Hu, Rongrong Liu, Guangming Tan, Ninghui Sun, Wanrun Jiang, Lijun Liu, Yixiao Chen, et al. Extending the limit of molecular dynamics with ab initio accuracy to 10 billion atoms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 205–218, 2022.

[51] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[52] John Wallis. *Opera mathematica*, volume 2. E Theatro Sheldoniano, Oxonii, 1693.

[53] Leonhard Euler. *Introductio in analysin infinitorum*, volume 2. MM Bousquet, 1748.

[54] Eduard Study. *Geometrie der Dynamen. Die Zusammensetzung von Kräften und verwandte Gegenstände der Geometrie*. B. G. Teubner, Leipzig, 1903.

[55] RWHT Hudson. Geometrie der dynamen. die zusammensetzung von kräften, und verwandte gegenstände der geometrie. von e. study.(leipzig, teubner, 1903.) pp. 603. m. 21. *The Mathematical Gazette*, 3(44):15–16, 1904.

[56] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153):1–43, 2018.

[57] William Kingdon Clifford. Preliminary sketch of biquaternions. *Proceedings of the London Mathematical Society*, 1(1):381–395, 1873.

[58] Robert Edwin Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

[59] Isaac Newton. *The Method of Fluxions and Infinite Series; With Its Application to the Geometry of Curve-Lines*. Henry Woodfall, London, 1736. Originally written in Latin as 'De Methodis Serierum et Fluxionum' c. 1671.

[60] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-mode automatic differentiation in julia. *arXiv preprint arXiv:1607.07892*, 2016.

[61] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[62] Bert Speelpenning. *Compiling fast partial derivatives of functions given by algorithms*. University of Illinois at Urbana-Champaign, Urbana-Champaign, 1980.

[63] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[64] Adrian Hill, Guillaume Dalle, and Alexis Montoison. An illustrated guide to automatic sparse differentiation. In *ICLR Blogposts 2025*, 2025.