Numerical Solution of Differential Equations

Liz Bradley

Department of Computer Science
University of Colorado
Boulder, Colorado, USA 80309-0430

lizb@cs.colorado.edu

Research Report on Curricula and Teaching CT003-98

1 Ordinary Differential Equations

A differential equation expresses a set of constraints among the derivatives of an unknown function. Here's a simple example:

$$\frac{d}{dt}x(t) = ax(t) \tag{1}$$

What this means is that the derivative of the unknown function is equal to a times the unknown function itself. For compactness, the independent variable (t, here) is often omitted and the notation $\frac{dx}{dt}$ is often abbreviated with a dot, like so: $\dot{x}$ or with a prime, like so: x'. Using these two notations, equation (1) becomes $\dot{x} = ax$ or $x' = ax$, respectively.

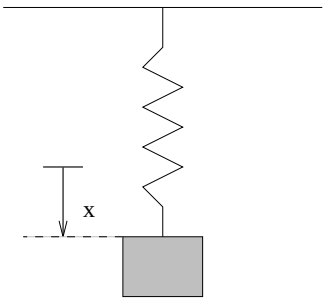
To solve a differential equation, you (generally) can't just integrate both sides. Rather, you have to find some function that satisfies the constraints expressed in that equation — in the example above, some function $x(t)$ whose derivative $x'(t)$ is equal to a constant multiple of the function itself[1]. There is no general, foolproof way to do this unless the equation is very simple. Differential equations textbooks are cookbooks that give you lots of suggestions about approaches, but there are lots of differential equations (DEs) that simply don't *have* analytic solutions — that is, solutions that you can write down. These equations can only be solved numerically, using the kinds of methods that are described in these notes.

Differential equations are interesting and useful to scientists and engineers because they "model" the physical world: that is, they capture the physics of a system, and their solutions emulate the behavior of that system. The class of differential equations that have no analytic solutions has a highly specific and interesting analog in the physical world: they model so-called *chaotic* systems. This means that numerical methods for solving ODEs are an essential tool for anyone (like me) who studies chaos.

An *ordinary* differential equation (ODE) has only one independent variable, and all derivatives in it are taken with respect to that variable. Most often, this variable is time t, but some books use x as the independent variable; pay careful attention to what the derivative is taken with respect to so you don't get confused.

Here's an example of where ODEs come from. Consider a mass m on a spring with spring constant k:

[1] The answer is $x(t) = be^{at} + c$, where a, b and c are constants.



If you define x as the position of the end of the spring, as shown in the figure, and set $x = 0$ at the point where the end of the spring would naturally rest if it were unloaded (i.e., if $m = 0$), you can write the following force balance at the mass:

$$\begin{aligned} \Sigma F &= ma \\ mg - kx &= ma \end{aligned} \tag{2}$$

Acceleration a is the second derivative of position: $a = x''$, so the equation becomes:

$$mg - kx = mx''$$

The mg force is gravity; kx is Hooke's law for the force exerted by a spring. The signs of mg and kx are opposite because gravity pulls in the direction of positive x and the spring pulls in the direction of negative x. Gathering terms and dividing through by m gives you the following ODE for the spring-mass system:

$$x''(t) + \frac{k}{m}x(t) - g = 0$$

Note that this ODE model does not include the effects of friction; if friction plays an important role in the system, this model is inadequate.

The independent variable t only appears *implicitly* in this equation — that is, hidden in the time dependence of the function $x(t)$. This means that the *physics* (the governing equations) of the system is not a function of time. If, on the other hand, someone were holding the top of the spring and moving it up and down in a sinusoidal pattern, the apparent gravity acting on the mass would change sinusoidally (much as the gravity changes in an elevator that is starting or stopping), and the equation would look like this:

$$x''(t) + \frac{k}{m}x(t) - g_0 \sin t = 0$$

The variable t appears *explicitly* in this ODE — by itself, and not wrapped in the brackets of a function like $x(t)$. Systems where this occurs are called *nonautonomous*.

The *order* of an ODE is the degree of the highest derivative in that equation. $x' = ax$ is a first-order ODE, $x''' - \tan x' = 2$ is a third-order ODE, and the spring-mass equation above is second order. An n^{th}-order ODE can be transformed into n first-order ODEs (an n^{th}-order "ODE system"), and vice versa[2]. This transformation requires the introduction of "helper" variables. I'll illustrate with the third-order ($n = 3$) example

$$x_1'''(t) + 36 \log x_1'(t) - x_1^2(t) + \sin 2t = 14$$

1. The first step is to rewrite the ODE with the highest-order term by itself on the left-hand side:

$$x_1''' = 14 + x_1^2 - 36 \log x_1' - \sin 2t$$

[2]This means that the order of an ODE system is equal to the number of first-order ODEs in the corresponding ODE system.

2. The second step is to define $n-1$ helper variables, like so:

$$x_1' = x_2, x_2' = x_3$$

3. The third step is to rewrite the equation from step 1 using the helper variables, with no derivative signs at all on the right-hand side and only one on the left-hand side:

$$x_3' = 14 + x_1^2 - 36\log x_2 - \sin 2t$$

4. Finally, one appends the equations that define the helper variables to that rewritten equation, obtaining the n^{th}-order system:

$$\begin{aligned} x_1' &= x_2 \\ x_2' &= x_3 \\ x_3' &= 14 + x_1^2 - 36\log x_2 - \sin 2t \end{aligned}$$

This set of first-order ODEs is equivalent to $x_1''' = 14 + x_1^2 - 36\log x_1' - \sin 2t$, as you can see by substituting the first two equations into the third. The variables that appear on the left-hand side of an ODE system are termed the *state variables* of the system. The *state vector* $\vec{x}$ of this system is $(x_1\ x_2\ x_3)^T$ and the ODE system is of the form

$$\vec{x}' = \vec{f}(\vec{x}, t)$$

I will use this standard form throughout these notes. For autonomous systems, $\vec{f}$ doesn't actually depend on t at all, so the standard form collapses to

$$\vec{x}' = \vec{f}(\vec{x})$$

Here's a famous third-order ODE system, derived by Edward Lorenz, that models the physics of a fluid that is being heated from below:

$$\vec{x}' = \vec{f}(\vec{x}) = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a(y-x) \\ rx - y - xz \\ xy - bz \end{bmatrix} \tag{3}$$

This system of ODEs is autonomous (no explicit ts on the right-hand side). It is also *nonlinear*, as you can see by looking at equation (3) above; if any of right-hand sides of any of these equations contain any terms that are nonlinear (e.g., products, transcendentals, powers, ...) in the state variables, then the whole system is nonlinear. Here, the product terms xz and xy are the culprits; a, r, and b are constants. The simple example $x' = ax$, in contrast, is *linear*; $x_1''' + 36\log x_1' - x_1^2 + \sin 2t = 14$ is nonlinear because of the log and x_1^2 terms. This terminology can be a little confusing; the individual equations in the system (3) — like the $x' = a(y-x)$ one — may look linear by themselves, but you need to remember that the whole thing is a package: a *single* (albeit vector-valued) function of a vector-valued argument. You might also think that the derivative operator makes the equations nonlinear, but it doesn't work that way. The *solutions* to the ODE are another matter: an ODE that is linear in its dependent variables can have solutions that are nonlinear in its independent variable (e.g., $x' = ax$ and its solution $x(t) = e^{at}$).

The order of the ODE affects the "width" of the solution. The solution to the first-order ODE $x' = ax$, for example, is the single function $x(t) = be^{at}$. The solution to an n^{th}-order ODE system is a set of n functions, each of which both obeys the rules of the ODE and describes the evolution of one state variable: $x_1(t)$, $x_2(t)$, $x_3(t)$ in the $x_1''' + 36\log x_1' - x_1^2 + \sin 2t = 14$ example[3], or $x(t)$, $y(t)$, and $z(t)$ for the Lorenz system. These n functions define a trajectory in the *state space* of that system: the space whose axes are the state variables. One can plot the three pieces of the ODE solution individually, against time, or against one another, as shown in figure 1.

[3]Strictly speaking, in a nonautonomous system like this, one must consider t to be another state variable, but this won't be a factor in our study of these kinds of system.

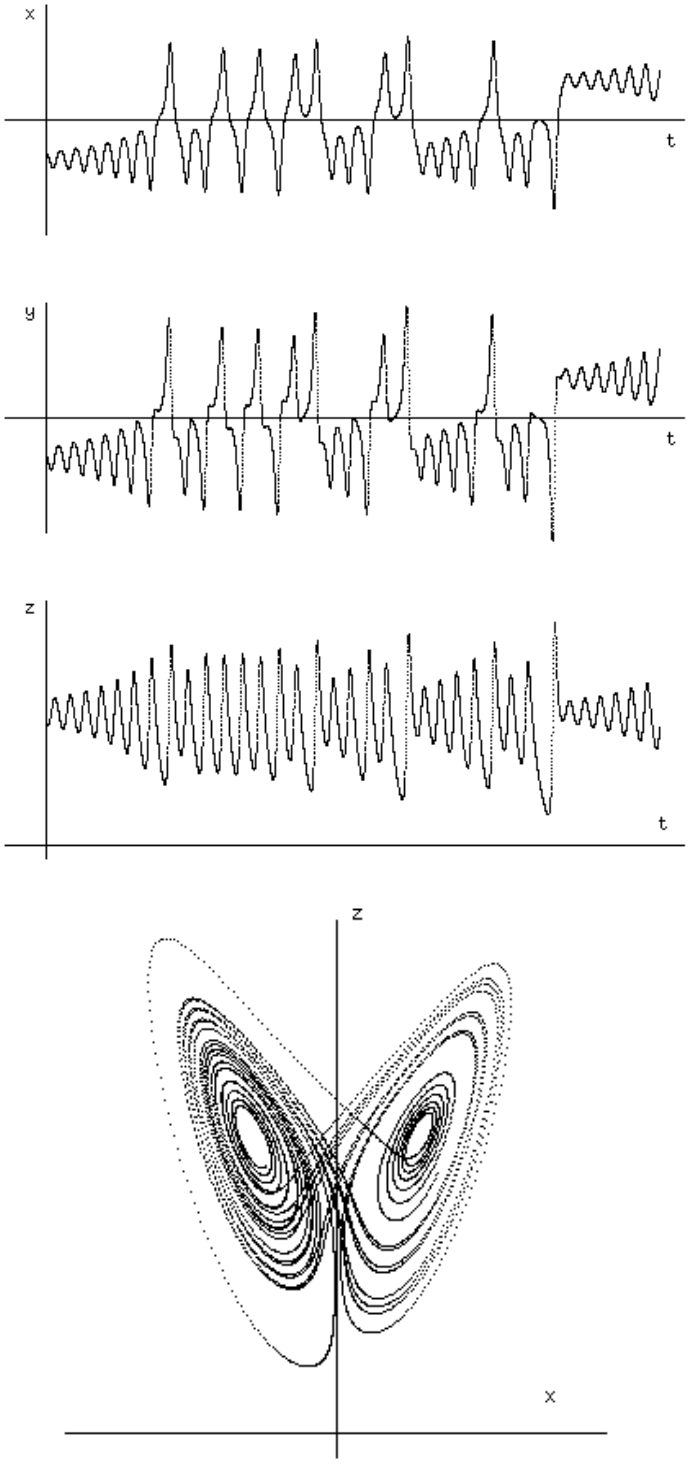


Figure 1: The Lorenz system. From top to bottom: $x(t)$ versus t, $y(t)$ versus t, $z(t)$ versus t, and $z(t)$ versus $x(t)$. A 3D plot of all three pieces of the solution against one another — with $x(t)$ on the x axis, $y(t)$ on the y axis, and $z(t)$ on the z axis — is called a *state-space* plot.

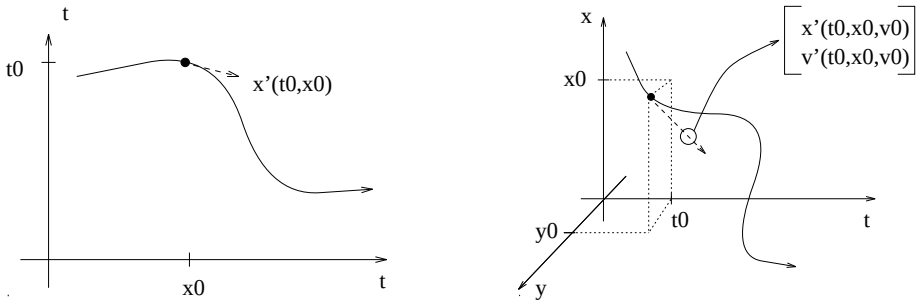


Figure 2: (a) A solution to a first-order ODE (b) A solution to a second-order ODE. Solid lines show the state-space trajectories and dashed lines show the derivative vectors at example state-space points — (x0,t0) in part (a) and (x0,y0,t0) in part (b).

2 Numerical Solution of ODEs

2.1 Single-Step Methods

Solving an ODE numerically means computing the *trajectory* $\vec{x}(t)$ from some initial condition $\vec{x}(t_0)$. This is different from the general, analytic solution, which tells you $\vec{x}(t)$ *everywhere*. A useful metaphor for ODE solvers is to think of figuring out where a ball will roll on a landscape, given the slope of the landscape and the initial position of the ball. An ODE is a machine that takes a point and gives you the slope at that point, and an ODE solver is a numerical method that uses these slopes to compute the trajectory from some initial condition. A first-order ODE describes the slope of a 2D landscape (x vs. t, for instance); see figure 2(a). A second-order ODE describes the slope of a 3D landscape, as shown in figure 2(b). Note that the slope vector in this image in Figure 2 has two pieces: an x-direction slope and a y-direction slope. The ball rolling on that landscape reacts to *both* of them.

The rest of this section describes four basic numerical ODE solution algorithms: Forward Euler, Backward Euler, Trapezoidal, and fourth-order Runge-Kutta. All four of these methods take an ODE in the standard form $\vec{x}' = \vec{f}(\vec{x}, t)$, an initial condition $\vec{x}(t_0)$, and a step size h, and generate an approximation of the solution $\vec{x}(t)$ for $t > t_0$. There are **lots** of other numerical ODE solvers; this is only a small sample. There are different names for these methods, by the way; many people call Backward Euler "Modified Euler" or "Implicit Euler."

2.1.1 Forward Euler

Forward Euler (FE) is the simplest and most obvious numerical ODE solver. It uses the slope at each point, computed using the ODE, to extrapolate and find the next point:

$$\vec{x}_{n+1} = \vec{x}_n + h\vec{x}'_n$$

Note that $\vec{x}'$ is a derivative, so its units are in distance per time. Multiplying it by h—an increment of time—puts the $h\vec{x}'_n$ term in units of distance.

Example: solve the first-order ODE system $x'(t) = -2x(t) + t$ from $x(0) = 1$ for two steps using Forward Euler with a time step $h = 0.1$.

First step:

$$\begin{aligned} x(0.1) &= x(0) + h(x'(0)) \\ &= 1 + (0.1)(-2(1) + 0) \\ &= 0.8 \end{aligned}$$

Second step:

$$\begin{aligned} x(0.2) &= x(0.1) + h(x'(0.1)) \\ &= 0.8 + (0.1)(-2(0.8) + 0.1) \\ &= 0.65 \end{aligned}$$

Example: solve the Lorenz system on page 3 of these notes with $a = 16$, $r = 50$, and $b = 4$ from the initial condition $[x, y, z] = [0, 1, 2]$ for one step using Forward Euler with a time step $h = 0.001$.

The arithmetic here is identical to that in the first-order system above, except that all the xs are now three-vectors instead of scalars:

$$\begin{aligned} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{t=0.001} &= \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{t=0} + h \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}_{t=0} \\ \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{t=0.001} &= \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} + 0.001 \begin{bmatrix} 16(1-0) \\ 50(0) - 1 - (0)(2) \\ (0)(1) - (4)(2) \end{bmatrix} \\ \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{t=0.001} &= \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} + \begin{bmatrix} .016 \\ -.001 \\ -.008 \end{bmatrix} \\ &= \begin{bmatrix} .016 \\ .999 \\ 1.992 \end{bmatrix} \end{aligned}$$

Here is an example of how one would write Scheme code that implements the Lorenz ODEs. Scheme vectors look like `#(3 2 5)`; the selector `(vector-ref vec n)` grabs the `n`th element of `vec`.

```
(define *lorenz-a* 16)
(define *lorenz-r* 50)    ;; constant values that yield the
(define *lorenz-b* 4)     ;; classic Lorenz butterfly

;; state vector = [x y z]
(define (lorenz state-vect)
  (let* ((x (vector-ref state-vect 0))
         (y (vector-ref state-vect 1))
         (z (vector-ref state-vect 2))
         (new-state (make-vector 3 0)))
    (vector-set! new-state 0 (* *lorenz-a* (- y x)))
    (vector-set! new-state 1 (- (* *lorenz-r* x) y (* x z)))
    (vector-set! new-state 2 (- (* x y) (* *lorenz-b* z)))
    new-state))
```

This ODE function takes a state vector $\vec{x}$ and returns its derivative $\vec{x}'$:

```
 (lorenz #(0 1 2))
;Value 1: #(16 -1 -8)
```

Here is a chunk of Scheme code that implements a single Forward Euler step, together with an example of how you apply it.

```
;x = state vector       h = stepsize
;f = system deriv   dx/dt = value of f at x
```

```
(define (forward-euler f dx/dt x h)
    (add-vectors x (scale-vector h dx/dt)))

(forward-euler lorenz (lorenz #(0 1 2)) #(0 1 2) 0.002)
;Value 4: #(.032 .998 1.984)
```

The function `add-vectors` adds two vectors:

```
 (add-vectors #(1 2 3) #(1 1 1)) ;Value 2: #(2 3 4)
```

and `scale-vector` multiplies a vector (e.g., `dx/dt`) by a scalar (e.g., `h`):

```
(scale-vector 2 #(1 2 3))
;Value 3: #(2 4 6)
```

Here is a simple recursive function that calls a stepper function like `forward-euler` `n` times on a system derivative `f` to generate an `n`-point trajectory, feeding back the result of each step as the next initial condition.

```
(define (generate-trajectory stepper f initial-condition h n)
  (if (> 0 n) '()
      (let* ((new-state (stepper f (f initial-condition) initial-condition h)))
         (cons new-state
               (generate-trajectory stepper f new-state h (- n 1))))))
```

`let*` is one way to define local variables in Scheme.

Here's how you would combine all of these functions to solve the Lorenz system from the initial condition $(x, y, z) = (0, 1, 2)$ for 10 steps using a time step of 0.001. Pretty-print (`pp`) is just a nicely formatted version of print:

```
(pp (generate-trajectory forward-euler lorenz #(0 1 2) .001 10))

(#(.016 .999 1.992)
 #(.031728 .998769128 1.984047984)
 #(.047200658048 .9992938089975636 1.9761434810108933)
 #(.06243414846319302 1.0005612728182651 1.9682860744122177)
 #(.07744418245287417 1.0025595307036186 1.9604753993056225)
 #(.09224602802488607 1.0052773528810406 1.9527111401116157)
 #(.10685452922258455 1.0087042470828487 1.944993028394036)
 #(.12128412470834878 1.0128304379825048 1.9373208408979068)
 #(.13554886572073527 1.0176468475174723 1.9296943977874639)
 #(.14966243342948307 1.0231450760691838 1.9221135610721993)
 #(.16363815571171828 1.029317384471711 1.9145782332097465))
```

These ten three-vectors are coordinates of the first ten points in the trajectory that starts from (0, 1, 2) in the state space of the Lorenz system.

2.1.2 Backward Euler

Backward Euler (BE) is somewhat more complicated. It takes a FE step to check out the landscape ahead, and then uses the slope at *that* point (*not* the slope at the original point, as FE does) to extrapolate from the original point:

$$\vec{x}_{n+1} = \vec{x}_n + h\vec{x}'_{n+1,FE}$$

where $x_{n+1,FE}$ is the result of taking a FE step from $\vec{x}_n$ and $\vec{x}'_{n+1,FE}$ is the slope at that point.

Here is a BE stepper function, written in Scheme:

```
(define (backward-euler f dx/dt x h)
  (let* ((xp (add-vectors x (scale-vector h dx/dt))))
    (add-vectors x (scale-vector h (f xp)))))
```

Example: solve the system $x' = -2x + t$ from $x(0) = 1$ for two steps using Backward Euler with a time step $h = 0.1$.

First step:

$$\begin{aligned} x(0.1)_{FE} &= x(0) + h(x'(0)) \\ &= 1 + (0.1)(-2(1) + 0) \\ &= 0.8 \\ x(0.1) &= x(0) + h(x'(0.1)_{FE}) \\ &= 1 + (0.1)(-2(0.8) + 0.1) \qquad [*] \\ &= 0.85 \end{aligned}$$

Note that $t = 0.1$ in the next-to-last line above, marked with a [*]! Remember that the act of taking a FE step moves the ball's position on the landscape *and* increments the timestamp, and you have to take that into account when you're computing the slope at that new point.

Second step:

$$\begin{aligned} x(0.2)_{FE} &= x(0.1) + h(x'(0.1)) \\ &= 0.85 + (0.1)(-2(0.85) + 0.1) \\ &= 0.69 \\ x(0.2) &= x(0.1) + h(x'(0.2)_{FE}) \\ &= 0.85 + (0.1)(-2(0.69) + 0.2) \\ &= 0.732 \end{aligned}$$

Example: The Lorenz system.

```
(pp (generate-trajectory backward-euler lorenz #(0 1 2) .001 10))
(#(.015728 .999769128 1.992047984)
 #(3.1216828318292922e-2 1.0002807917147354 1.9841430596248)
 #(.04648174930954169 1.0015237714022671 1.976284867040263)
 #(.06153762214563069 1.0034875632375726 1.9684730944643098)
 #(.07639891815243106 1.0061623605516097 1.9607074762065357)
 #(.09107973759143692 1.0095390358679155 1.9529877909018014)
 #(.10559382590715283 1.0136091237117053 1.94531385992661)
 #(.11995458946040508 1.0183648041673417 1.9376855459879665)
 #(.13417511076706598 1.0237988871609087 1.9301027518752438)
 #(.14826816326102132 1.02990479744548 1.9225654193663486)
 #(.16224622559957547 1.0366765602674808 1.915073528280206))
```

2.1.3 Trapezoidal

The Trapezoidal method is the obvious next step: an average of Forward and Backward Euler. It takes a FE step to get $x_{n+1,FE}$, uses the ODE to compute the slope at that point, averages that slope with the slope at the current point, and uses the averaged slope to move forward:

$$\vec{x}_{n+1} = \vec{x}_n + \frac{h}{2}(\vec{x}'_n + \vec{x}'_{n+1,FE})$$

Trapezoidal is a better method than FE and BE; it has $O(h^3)$ error, compared to their $O(h^2)$. Here is a Trapezoidal method stepper function:

```
(define (trapezoidal f dx/dt x h)
  (let* ((xp (add-vectors x (scale-vector h dx/dt)))
         (avg-slope (scale-vector 0.5
                                  (add-vectors (f xp) dx/dt))))
    (add-vectors x (scale-vector h avg-slope))))
```

Example: solve the system $x' = -2x + t$ from $x(0) = 1$ for two steps using Trapezoidal with a time step of $h = 0.1$.

First step:

$$\begin{aligned} x(0.1)_{FE} &= x(0) + h(x'(0)) \\ &= 1 + (0.1)(-2(1) + 0) \\ &= 0.8 \\ x(0.1) &= x(0) + h/2(x'(0) + x'(0.1)_{FE}) \\ &= 1 + (0.05)[(-2(1) + 0) + (-2(0.8) + 0.1)] \\ &= 0.825 \end{aligned}$$

Second step:

$$\begin{aligned} x(0.2)_{FE} &= x(0.1) + h(x'(0.1)) \\ &= 0.825 + (0.1)(-2(0.825) + 0.1) \\ &= 0.670 \\ x(0.2) &= x(0.1) + h/2(x'(0.1) + x'(0.2)_{FE}) \\ &= 0.825 + (0.05)[(-2(.825) + 0.1) + (-2(0.670) + 0.2)] \\ &= 0.690 \end{aligned}$$

Example: the Lorenz system.

```
(pp (generate-trajectory trapezoidal lorenz #(0 1 2) .001 10))
(#(.015864 .999384564 1.992023992)
 #(3.1472536103547125e-2 .9995247571602539 1.9840955143172878)
 #(4.6841554216401654e-2 1.000408199698714 1.9762141524393404)
 #(.06198655681804697 1.0020232707455323 1.9683795429272173)
 #(.07692262187350225 1.0043590873379908 1.9605913711251455)
 #(.09166442111877579 1.007405484300651 1.952849369087303)
 #(.10622623774371602 1.0111529949819287 1.9451533137085624)
 #(.12062198349519176 1.01559283281934 1.9375030250469871)
 #(.1348652152227369 1.0207168737066725 1.929898364826848)
 #(.14896915088802673 1.0265176391373139 1.9223392351118225)
 #(.16294668505881293 1.0329882800989165 1.914825577138889))
```

2.1.4 Runge-Kutta

The Runge-Kutta method is one of the most widely used numerical ODE solvers. It is based on taking test steps (à la Backward Euler) of different lengths, using the results to kludge together a good approximation of the power series of the function, and then using that power series to extrapolate. Fourth-order Runge-Kutta (RK4, henceforth), the most common member of the RK family, uses a four-term version of the power series; it has $O(h^5)$ error (per step) and its algebra is fairly complex.

For autonomous ODEs, the RK formulae look like this:

1. compute $\vec{v}_1 = \vec{f}(\vec{x})$

2. compute $\vec{v}_2 = \vec{f}(\vec{x} + \frac{h}{2}\vec{v}_1)$
3. compute $\vec{v}_3 = \vec{f}(\vec{x} + \frac{h}{2}\vec{v}_2)$
4. compute $\vec{v}_4 = \vec{f}(\vec{x} + h\vec{v}_3)$
5. use the $\vec{v}_i$ to compute the next $\vec{x}$:

$$\vec{x}(t_0 + h) = \vec{x}(t_0) + \frac{h}{6}(\vec{v}_1 + 2\vec{v}_2 + 2\vec{v}_3 + \vec{v}_4)$$

The $\vec{v}_i$ are somewhat like the $\vec{x}_{FE}$ in BE and Trapezoidal: they're test steps that look ahead on the landscape. (In fact, $\vec{v}_0$ is exactly the increment added during a FE step from $\vec{x}$.)

There are variations on this algorithm that use slightly different algebra to get the same answer; the most-common one uses $\vec{k}$s instead of $\vec{v}$s:

1. compute $\vec{k}_1 = h\vec{f}(\vec{x})$
2. compute $\vec{k}_2 = h\vec{f}(\vec{x} + \frac{1}{2}\vec{k}_1)$
3. compute $\vec{k}_3 = h\vec{f}(\vec{x} + \frac{1}{2}\vec{k}_2)$
4. compute $\vec{k}_4 = h\vec{f}(\vec{x} + \vec{k}_3)$
5. use the $\vec{k}_i$ to compute the next $\vec{x}$:

$$\vec{x}(t_0 + h) = \vec{x}(t_0) + \frac{1}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)$$

If the ODE is nonautonomous (that is, if the right-hand side contains an explicit reference to the independent variable), the RK4 procedure has to take into account that numerical ODE solution *both* evolves the spatial position *and* advances time, so when you're evaluating the system derivative `f` at a point like $(\vec{x} + \frac{1}{2}\vec{k}_0)$, you have to use the appropriate timestamp (just as you do in Backward Euler; see my comments about the line marked [*] on page 8.)

1. compute $\vec{v}_1 = \vec{f}(\vec{x}, t)$
2. compute $\vec{v}_2 = \vec{f}(\vec{x} + \frac{h}{2}\vec{v}_1, t + h/2)$
3. compute $\vec{v}_3 = \vec{f}(\vec{x} + \frac{h}{2}\vec{v}_2, t + h/2)$
4. compute $\vec{v}_4 = \vec{f}(\vec{x} + h\vec{v}_3, t + h)$
5. use the $\vec{v}_i$ to compute the next $\vec{x}$:

$$\vec{x}(t_0 + h) = \vec{x}(t_0) + \frac{h}{6}(\vec{v}_1 + 2\vec{v}_2 + 2\vec{v}_3 + \vec{v}_4)$$

The $\vec{k}$ version of the algorithm changes as follows for nonautonomous systems:

1. compute $\vec{k}_1 = h\vec{f}(\vec{x}, t)$ [2]
2. compute $\vec{k}_2 = h\vec{f}(\vec{x} + \frac{1}{2}\vec{k}_1, t + h/2)$
3. compute $\vec{k}_3 = h\vec{f}(\vec{x} + \frac{1}{2}\vec{k}_2, t + h/2)$
4. compute $\vec{k}_4 = h\vec{f}(\vec{x} + \vec{k}_3, t + h)$

5. use the $\vec{k}_i$ to compute the next $\vec{x}$:

$$\vec{x}(t_0 + h) = \vec{x}(t_0) + \frac{1}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)$$

Here is a 4th-order Runge-Kutta stepper and an example of its application to the Lorenz system. I've modified this version to print out the $\vec{k}_i$ so you can use these numbers to check your answers:

```
(define (runge-kutta-4 f dx/dt x h)
  (let* ((k1 (scale-vector h dx/dt))
         (k2 (scale-vector h (f (add-vectors x (scale-vector 0.5 k1)))))
         (k3 (scale-vector h (f (add-vectors x (scale-vector 0.5 k2)))))
         (k4 (scale-vector h (f (add-vectors x k3)))))
; take the following line out to omit printing of the ki
    (pp (list 'k1 k1 'k2 k2 'k3 k3 'k4 k4))
    (add-vectors x
                 (scale-vector (/ 1 6)
                               (add-vectors
                                (add-vectors k1 (scale-vector 2 k2))
                                (add-vectors k4 (scale-vector 2 k3)))))))
```

Example: solve the Lorenz system on page 3 of these notes with $a = 16$, $r = 50$, and $b = 4$ from the initial condition $[x, y, z] = [0, 1, 2]$ using fourth-order Runge-Kutta with a time step $h = 0.001$.

First, I'll do one step by hand (ouch ouch ouch):

$$\begin{aligned}
\vec{k}_1 &= \begin{bmatrix} k_{1x} \\ k_{1y} \\ k_{1z} \end{bmatrix} = h\vec{f}(\vec{x}) = h\vec{f}\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{t=0} = h\vec{f}\begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \\
&= 0.001\begin{bmatrix} 16(1-0) \\ 50(0) - 1 - (0)(2) \\ (0)(1) - (4)(2) \end{bmatrix} = \begin{bmatrix} .016 \\ -.001 \\ -.008 \end{bmatrix} \\
\vec{k}_2 &= \begin{bmatrix} k_{2x} \\ k_{2y} \\ k_{2z} \end{bmatrix} = h\vec{f}(\vec{x} + \frac{1}{2}\vec{k}_1) = h\vec{f}\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{t=0} + \frac{1}{2}\begin{bmatrix} k_{1x} \\ k_{1y} \\ k_{1z} \end{bmatrix}\right) \\
&= h\vec{f}\begin{bmatrix} 0 + 0.008 \\ 1 - 0.0005 \\ 2 - 0.004 \end{bmatrix} = h\vec{f}\begin{bmatrix} 0.008 \\ 0.9995 \\ 1.9996 \end{bmatrix} \\
&= 0.001\begin{bmatrix} 16(0.9995 - 0.008) \\ 50(0.008) - 0.9995 - 0.008(1.9996) \\ 0.008(0.9995) - 4(1.9996) \end{bmatrix} = 0.001\begin{bmatrix} 15.864 \\ -0.6155 \\ -7.990 \end{bmatrix} \\
&= \begin{bmatrix} .015864 \\ -.0006155 \\ -.00798 \end{bmatrix} \\
\vec{k}_3 &= \begin{bmatrix} k_{3x} \\ k_{3y} \\ k_{3z} \end{bmatrix} = h\vec{f}(\vec{x} + 1/2\vec{k}_2) = h\vec{f}\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{t=0} + \frac{1}{2}\begin{bmatrix} k_{2x} \\ k_{2y} \\ k_{2z} \end{bmatrix}\right) \\
&= h\vec{f}\begin{bmatrix} 0 + 0.007932 \\ 1 - .0003077 \\ 2 - .003988 \end{bmatrix} = 0.001\vec{f}\begin{bmatrix} 0.007932 \\ .9996923 \\ 1.996012 \end{bmatrix} \\
&= 0.001\begin{bmatrix} 16(.9996923 - 0.007932) \\ 50(0.007932) - .9996923 - 0.007932(1.996012) \\ 0.007932(.9996923) - 4(1.996012) \end{bmatrix} = 0.001\begin{bmatrix} 15.86816 \\ -.61892 \\ -7.976 \end{bmatrix}
\end{aligned}$$

$$
\begin{aligned}
&= \begin{bmatrix} .01586816 \\ -6.1892e-4 \\ -.007976 \end{bmatrix} \\
\vec{k}_4 &= \begin{bmatrix} k_{4x} \\ k_{4y} \\ k_{4z} \end{bmatrix} = h\vec{f}(\vec{x}+\vec{k}_3) = h\vec{f}\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{t=0} + \begin{bmatrix} k_{3x} \\ k_{3y} \\ k_{3z} \end{bmatrix}\right) \\
&= h\vec{f}\begin{bmatrix} 0+0.01586816 \\ 1-6.1892e-4 \\ 2-0.007976 \end{bmatrix} = 0.001\vec{f}\begin{bmatrix} 0.015868 \\ 0.99938 \\ 1.992024 \end{bmatrix} \\
&= 0.001\begin{bmatrix} 15.73619 \\ -.237589 \\ -7.9522 \end{bmatrix} = \begin{bmatrix} 0.01573619 \\ -2.37589e-4 \\ -.0079522 \end{bmatrix} \\
\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{t=0.001} &= \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \\
&+ \frac{1}{6}\left(\begin{bmatrix} .016 \\ -.001 \\ -.008 \end{bmatrix} + 2\begin{bmatrix} .015864 \\ -.0006155 \\ -.00798 \end{bmatrix} + 2\begin{bmatrix} .01586816 \\ -6.1892e-4 \\ -.007976 \end{bmatrix} + \begin{bmatrix} 0.01573619 \\ -2.37589e-4 \\ -.0079522 \end{bmatrix}\right) \\
&= \begin{bmatrix} 1.58667e-2 \\ .999382 \\ 1.99202 \end{bmatrix}
\end{aligned}
$$

Incidentally, I was being very cavalier about significant figures in the calculations above; your mileage may vary.

Here are the results of a three-step `runge-kutta-4` run:

```
(pp (generate-trajectory runge-kutta-4 lorenz #(0 1 2) .001 3))

(k1 #(.016 -.001 -.008)
 k2 #(.015864 -.000615468 -.007976004)
 k3 #(.015868164256 -6.18924633168136e-4 -7.976118432946088e-3)
 k4 #(.01573620657777331 -2.3758262472141256e-4 -7.952237183209957e-3))
(k1 #(1.5736248258717785e-2 -2.3765143678056546e-4 -7.952238724124705e-3)
 k2 #(1.5608457061153798e-2 1.402944766530796e-4 -7.928473803221701e-3)
 k3 #(1.5612502958041778e-2 1.370374699923347e-4 -7.928580708423843e-3)
 k4 #(1.5488640810908996e-2 5.119857801574603e-4 -7.904917228774927e-3))
(k1 #(.01548868076306085 5.119189592231575e-4 -7.904918706871421e-3)
 k2 #(.01536886666863015 8.836695331569069e-4 -7.881358203662625e-3)
 k3 #(1.5372799185977067e-2 8.806064686823121e-4 -7.881457939012697e-3)
 k4 #(1.5256805679584137e-2 1.249546464021753e-3 -.00785798619111313))
(k1 #(.01525684396112769 1.2494815519405566e-3 -.00785798761163514)
 k2 #(1.5144785061854195e-2 1.6154165591077195e-3 -7.834606112490058e-3)
 k3 #(1.5148609013105718e-2 1.6125419889319837e-3 -7.834699005059483e-3)
 k4 #(1.5040266888740909e-2 1.9758483035414564e-3 -7.811394140896604e-3))
(#(1.5866755848295548e-2 .9993822720181571 1.992023919658483)
 #(3.1477890699631875e-2 .9995204383909351 1.9840953754957846)
 #(.04684936039160845 1.000402107962089 1.9762139526318954)
 #(6.1996676891573184e-2 1.0020156491206826 1.9683792873006236))
```

2.2 Multistep Methods

The methods described in the previous section — Forward and Backward Euler, Trapezoidal, and Runge-Kutta — are all *single-step* methods: they take *one* initial condition and figure out where the ODE solution

goes from there. The conceptual basis of these methods is to move along the slope of a landscape, as given by an ODE. They may take lots of test steps to check out the slope ahead, but they never look further back than the current point. Note that every one of these test steps requires an evaluation of the ODE.

Another class of ODE solvers, called *multistep methods*, takes a different approach; instead of moving along a slope, they integrate the area *under* the slope and use that to advance the solution to the next timestep. Recall that an ODE tells you the derivative of each state variable:

$$\vec{x}'(t) = \vec{f}(\vec{x}, t)$$

and that the integral of a derivative is simply an increment in the value of the associated variable, which you can derive by integrating both sides of the equation above with respect to the independent variable, t:

$$\int_{t_n}^{t_{n+1}} \vec{x}'(t)dt = \int_{t_n}^{t_{n+1}} \vec{f}(\vec{x}, t)dt \tag{4}$$

The left-hand side of equation (4) evaluates to $\vec{x}(t_{n+1}) - \vec{x}(t_n)$. $\vec{x}(t_n)$ is what we know — the jumping-off point — and $\vec{x}(t_{n+1})$ is what we want to compute, so if we can evaluate the right-hand side, we're done.

Not surprisingly, given their name, multistep methods use more than one initial condition to accomplish this. In particular, they take a bunch of points $x(t_n)$, $x(t_{n-1})$, $x(t_{n-2}), \ldots$), evaluate the ODE ($\vec{f}$) at each of those points, fit a polynomial to those values, extrapolate that polynomial to estimate the value of $\vec{f}$ out at t_{n+1}, then integrate that polynomial from t_n to t_{n+1} and add the result to $x(t_n)$.

This can be confusing upon first encounter, but it boils down to yet another subscript-laden formula, so you don't have to deal with all of the details of all of those methods as you crunch through the process, but you do need to understand the basic ideas. Below are two such formulae, the first of which — called second-order Adams(-Bashforth)[4] — fits a line to the slope at two points (the last one and the next-to-last one that you have) and uses that to extrapolate:

$$\vec{x}(t_{n+1}) = \vec{x}(t_n) + \frac{h}{2}\left[3\vec{f}(\vec{x}(t_n), t_n) - \vec{f}(\vec{x}(t_{n-1}), t_{n-1})\right] + O(h^3)$$

Third-order Adams-Bashforth, as you might expect, uses three initial conditions, fits a parabola to them, and uses that parabola to extrapolate and integrate:

$$\vec{x}(t_{n+1}) = \vec{x}(t_n) + \frac{h}{12}\left[23\vec{f}(\vec{x}(t_n), t_n) - 16\vec{f}(\vec{x}(t_{n-1}), t_{n-1}) + 5\vec{f}(\vec{x}(t_{n-2}), t_{n-2})\right] + O(h^4)$$

Because these methods need more than one point to make progress, firing them up can require some extra work. In particular, if you only have one initial condition, you may have to run a single-step solver like Forward Euler & Co. in order to concoct enough information to feed those formulae.

The big advantage of these methods is that increasing the order of the method, and thus the quality of the answer, does not increase the number of times you have to evaluate $\vec{f}$ (which is generally assumed to be much more work than the arithmetic of the formula). Note that both of these formulae require only a *single* evaluation of the ODE function for each step, as long as you're smart about saving the previous $\vec{f}$ values that you'll need for the formula. This is also true of higher-order Adams-Bashforth methods, but you'll obviously have to save n old f values for nth-order Adams-Bashforth, so there *is* an $O(n)$ storage cost. Note, too, that the error term of the second formula is better[5] — $O(h^4)$ vs. $O(h^3)$ — but the number of function evaluations (that is, calculations of $\vec{f}$) is the same. This is a major win.

Adams-Bashforth is a *predictor* method: it extrapolates forward from some known points. Adams-*Moulton* is a bit smarter: it predicts and then corrects. In particular, it uses an Adams-Bashforth scheme to compute $x(t_{n+1})$, and then does a *backward* "prediction" from that newly computed point (also using several of the

[4]Some textbooks call this simply the "Adams" method

[5]In general, the error order improves linearly with the method order (until you get bitten by roundoff, of course)

previous values), then adjusts $\vec{x}(t_{n+1})$ until that backward "prediction" falls in the right place. Here are the associated formulae:

Predictor:

$$\vec{x}(t_{n+1}) = \vec{x}(t_n) + \frac{h}{24}\left[55\vec{f}(\vec{x}(t_n), t_n) - 59\vec{f}(\vec{x}(t_{n-1}), t_{n-1}) + 37\vec{f}(\vec{x}(t_{n-2}), t_{n-2}) - 9\vec{f}(\vec{x}(t_{n-3}), t_{n-3})\right] + O(h^5)$$

Corrector: Watch out for the subscripts!

$$\vec{x}(t_{n+1}) = \vec{x}(t_n) + \frac{h}{24}\left[9\vec{f}(\vec{x}(t_{n+1}), t_{n+1}) + 19\vec{f}(\vec{x}(t_n), t_n) - 5\vec{f}(\vec{x}(t_{n-1}), t_{n-1}) + \vec{f}(\vec{x}(t_{n-2}), t_{n-2})\right] + O(h^5)$$

Again, note that you only have to evaluate the ODE once per step, regardless of the order of the method, and that the error decreases with the order of the method. As in the case of Adams-Bashforth, this is a big win over single-step methods, where a decrease in error requires an increase in how much work is required to obtain the answer.

2.3 Error and Stability in ODE Solvers

Error in numerical ODE solvers can be classified in several ways:

- absolute vs. relative
- local vs. global
- roundoff vs. truncation
- observational ("additive") vs. dynamical

The first distinction simply lies in how one measures things: in absolute units like volts or meters or in relative units like percentages, decibels, or parts per million.

Local error is what happens in one step. Global error is what accrues over a multi-step run.

Roundoff error comes from finite-precision arithmetic: it's the figures that get dropped after the significant ones (e.g., if 1.4825349 is stored in a four-decimal-place calculator, there's a roundoff error of 0.0000349). Truncation error comes from the method. If, for instance, some numerical method is based on a Taylor series but it only uses the first two terms, those "higher-order terms" that are omitted give rise to truncation error. The truncation error of RK4, for instance, is $O(h^5)$ (since it's essentially a fourth-order Taylor series); FE and BE are $O(h^2)$, and Trapezoidal is $O(h^3)$.

The final distinction — observational vs. dynamical — is perhaps the most important. Observational error is introduced *between* you and the system that you're looking at: it's like dirt or imperfections on a telescope lens. When the error actually gets *coupled back into the system*, it is called dynamical error — when, for instance, you step forward using RK4 and then feed that extrapolated state (complete with its local error) back into the solver as the new jumping-off point. This kind of error can snowball and wreak havoc on your solution. Among other things, the snowballing error can cause the solution of physically stable systems to blow up; for obvious reasons, methods that do this are called *numerically unstable*[6].

Numerical dynamics are a pernicious problem: the effects of the solver method, the time step, the computer arithmetic, etc., can mess with the system's behavior *in ways that look exactly like real physical effects*. This is akin to the Hubble Space Telescope turning stars into novas. You should always distrust your results and do some basic belief checks on them: change the time step, use a different method, use double-precision arithmetic instead of single, etc., and see if your results change. If they don't, it's safe(r) to trust them.

[6]The standard way to quantify the stability of a method is to try it out on the first-order linear system $\dot{x} = \lambda x,\ x(t) = x_0$, vary λ, and see whether the numerical solution does the right thing. The ranges of λ for which the algorithm is stable are called the *stability regions* for that algorithm.

2.4 Adaptive ODE Solvers

Choosing good time step values is hard; you have to use a small-enough h to get a good picture of the dynamics: it makes no sense to use one much smaller than that, or you're doing extra work. Often, the only solution is to do it by Braille: try some random time step, try a smaller one, try a bigger one, etc., until you zero in on a value where the solution quits changing.

Consider a *stiff* system: one that has two very different natural frequencies (e.g., a pulsar that moves once every ten years around its companion star, but wobbles 1000 times a second around its axis). Picking a good time step is even harder in cases like this. Either you use one that's small enough to capture the fast dynamics — and your solution takes forever — or you use a big time step and miss seeing the wobble.

The obvious solution to this is to change the time step on the fly: *adaptive time-step* ODE solvers. (Another, less-common, idea is to change the *order* of the method on the fly.) One simple idea behind adaptive time step solver algorithms is to take a full (h) step from $x(t)$ to get *one* estimate of $x(t+h)$, take two half ($h/2$) steps from $x(t)$ to get *another* estimate of $x(t+h)$, and then compare the two estimates. If they are within some (specified) tolerance of each other, the full time step (h) was certainly small enough — and perhaps even too small. If the two estimates are very different, h was too big. Many algorithms essentially do binary search on h using this strategy, halving h if the difference Δ_1 between the two estimates exceeds the tolerance, doubling it if the difference is lower than the tolerance, and settling on an h value when halving is unnecessary and doubling is unsuccessful.

There is a sneaky way to jump directly to the "right" h value — one that (like Aitken acceleration, Richardson extrapolation, Romberg integration, etc.) uses knowledge about the order of the truncation error to figure out exactly what time step to use to meet a given tolerance. Here's the idea. Remember that both estimates are accurate to a certain order in h, but in different time step values: $O(h^5)$ vs. $O(\frac{h}{2})^5$ in RK4, for instance. Inverting and solving the equation

$$better\ estimate = more - accurate + \frac{1}{2^n - 1}(more - accurate\ -\ less - accurate)$$

equation for this situation ($n = 5$) yields the following formula for the step size that you'd have to use to produce some *desired* accuracy, Δ_0:

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2}$$

If $|\Delta_1| > |\Delta_0|$, this equation tells you how much to decrease h. If $|\Delta_1| < |\Delta_0|$, it tells you how much to increase h. The only complicating factor is that the Δs are really vectors, and the common solution is to use the worst element of each Δ in the computation – the "worst offender" algorithm, according to Bill Press *et al.* in *Numerical Recipes*. Any other vector norm is fine, too.

Sometimes people also add a couple of "fudge factors" to this algorithm:

$$\begin{aligned} h_0 &= S h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2} \quad if \Delta_0 \geq \Delta_1 \\ &= S h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.25} \quad if \Delta_0 < \Delta_1 \end{aligned}$$

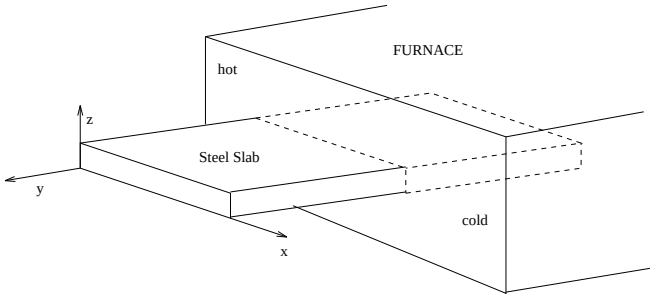
See section 15.2 of *Numerical Recipes* — on reserve for CSCI 3656 — for some handwaving about these fudge factors. All of these tricks work equally well for other ODE solvers, but you have to adapt the formula accordingly, by looking at the exponent in the $O(h^n)$ error term.

3 Understanding and Solving Partial Differential Equations

Many real-world systems cannot be described by ODEs. Anything that involves the flow of gas, fluid, or heat, for example, can only be accurately modeled by a *partial* differential equation (PDE), a more-complicated kind

of differential equation that involves more than one independent variable. Because of this, designers of things like airplanes, cars, racing yachts, weather models, and chemical reactors have to solve a lot of PDEs to get all the shapes, pressures, concentrations, etc. right. Solving PDEs analytically is almost always impossible, so numerical solvers are critical to this kind of endeavor.

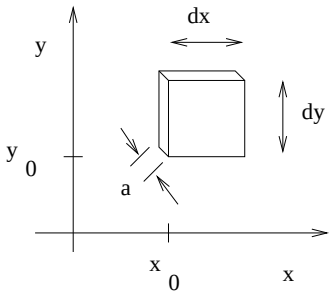
In this section, I'll cover one of the simplest examples of this kind of problem. Consider a slab of steel stuck partway into a furnace. The furnace is malfunctioning, so it's hotter (say, 1000 degrees C) on the left side and cooler (say, 850 degrees C) on the right:



If the slab is initially at room temperature (20 degrees C), heat will flow around in the steel for a while, and the system will eventually settle down to some thermal equilibrium. *(Thought experiment: which points on the slab will be the hottest? The coolest?)* The temperature evolution of each tiny patch of steel in the slab depends on four variables:

- time t
- the "across-the-slab" position x of the patch
- the "along-the-slab" position y of the patch
- the vertical position z of the patch

The "state variables" of this system[7] are the values of the heat u at every point (x, y, z) in the slab. The u at each point also depends on t, so it is written $u(x, y, z, t)$. If the slab is much thinner than it is long or wide, it is safe to assume that it's the same temperature all the way across in that dimension, so the z variable doesn't matter and the heat can be written $u(x, y, t)$. Here's a picture of such a patch:



...where I've called the slab thickness a. If any of its neighbor patches is hotter — i.e., its u is higher — then heat will flow into this patch across the corresponding boundary, and u in this patch will increase. Conversely, if the neighbor patch is cooler, heat will flow out across that boundary and u will decrease.

[7] I have quotes there because there are an infinite number of points in the slab, so the concept of a finite number of discrete state variables doesn't really make sense.

The amount of heat flow is proportional to the *gradient*: a big heat difference makes for a lot of flow. The gradient can be different in the x and y directions (consider the steel slab in the asymmetrical furnace), and so we have to distinguish them: $\frac{\partial u}{\partial x}$ for the former and $\frac{\partial u}{\partial y}$ for the latter.

Finally, the amount of heat flow is also proportional to the cross-sectional area involved, as well as to the *thermal conductivity* k of the boundary. (Potholders, for example, are devices with very low k.) The cross-sectional area on the right and left sides of the patch are $a\,dy$; the areas on the top and bottom are $a\,dx$.

To put this all together, you first write down how much heat is flowing in the positive x direction across the left-hand side of the box (that is, *into* the box): [***]

$$-k(a\,dy)\frac{\partial u}{\partial x}$$

The minus sign is necessary to get the flow direction right: here, I've defined flow *out of* a volume as positive and flow *into* a volume as negative. It works the other way too, but you do have to use the sign to keep track of direction. Next we need the relationship between the heat gradients on the different sides of the box. If the gradient on the left side (at $x = x_0$) is $\frac{\partial u}{\partial x}$, then the gradient on the right side (at $x = x_0 + dx$) is

$$\frac{\partial u}{\partial x} + \frac{\partial}{\partial x}(\frac{\partial u}{\partial x})dx$$

Using this fact, you write down how much heat is flowing *out* across the right-hand face of the box:

$$k(a\,dy)\left[\frac{\partial u}{\partial x} + \frac{\partial}{\partial x}(\frac{\partial u}{\partial x})dx\right]$$

Note the positive sign; since this is outward flow. Finally, you combine the inflow and outflow to get the net change in u due to heat flow in the x-direction:

$$-k(a\,dy)\left[\frac{\partial u}{\partial x} - \left(\frac{\partial u}{\partial x} + \frac{\partial}{\partial x}(\frac{\partial u}{\partial x})dx\right)\right]$$

The first two terms inside the brackets cancel, leaving

$$k\,a\,dx\,dy\frac{\partial^2 u}{\partial x^2}$$

If you repeat the derivation starting from [***], but using y instead, you'll obtain

$$k\,a\,dx\,dy\frac{\partial^2 u}{\partial y^2}$$

for the change in u due to heat flow in the y-direction.

If we're thinking about the equilibrium situation, the total amount of heat flow had better be zero, so the sum of the x heat flow and the y heat flow should be

$$k\,a\,dx\,dy\left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right] = 0$$

Dividing through by $-k\,a\,dx\,dy$, you get

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

or

$$\nabla^2 u = 0 \tag{5}$$

This is called the *steady-state heat equation*. This kind of equation — known as *Laplace's equation* — turns up all over the place in electromagnetics as well as in flow problems.

If there's a small internal heat source of size Q in the box, you have to add another term to this equation:

$$k\,a\,dx\,dy\left[\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right]+Q\,a\,dx\,dy=0$$

or

$$\nabla^2 u=-\frac{1}{k}Q$$

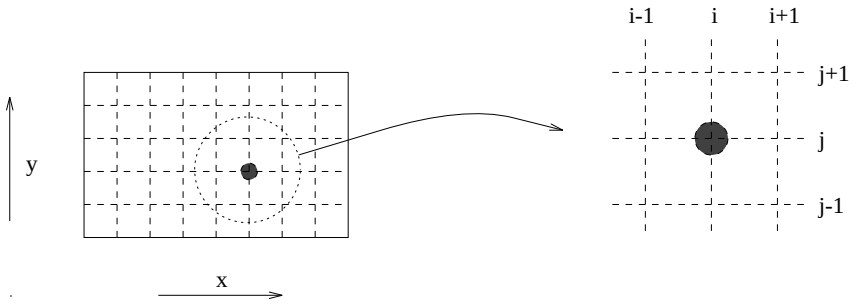
This is called *Poisson's equation*.

So how do you go from $\nabla^2 u=0$ to C++ code that figures out the temperature at the various points in the slab in the furnace?

Numerical solution of the heat equation:

1. divide the slab up on a grid
2. use differences to approximate the second derivatives $\frac{\partial^2 u}{\partial x^2}$ and $\frac{\partial^2 u}{\partial x^2}$ at each grid point
3. write down $\nabla^2 u=0$ at each grid point using those difference approximations, formulate the whole mess as a big matrix, and solve it with some $A\vec{x}=\vec{b}$ solver

Here's a picture of a slab divided on a 4-by-7 grid, and a close-up of a small section of the grid:



If you know the values for u_{ij} at some grid point (e.g., the one with the dot on it) and at all eight of its neighbors, it's easy to figure out what $\frac{\partial^2 u}{\partial x^2}$ and $\frac{\partial^2 u}{\partial y^2}$ are; you can just use any of the standard difference formulas (e.g., first-order forward, center, *et al.*). In this case, it makes intuitive sense to use the center-difference formula, rather than one of the lopsided ones like forward or backward, because it's a better match to the flow physics. Here's the formula, again:

$$\frac{\partial^2 u}{\partial x^2}=\frac{u(x_{i+1},y_j)-2u(x_i,y_i)+u(x_{i-1},y_i)}{(\Delta x)^2} \tag{6}$$

in the x direction and

$$\frac{\partial^2 u}{\partial y^2}=\frac{u(x_i,y_{j+1})-2u(x_i,y_i)+u(x_i,y_{j-1})}{(\Delta y)^2} \tag{7}$$

in the y direction. The steady-state heat equation says that the sum of these two second derivatives must be zero:

$$\frac{u(x_{i+1},y_j)-2u(x_i,y_i)+u(x_{i-1},y_i)}{(\Delta x)^2}+\frac{u(x_i,y_{j+1})-2u(x_i,y_i)+u(x_i,y_{j-1})}{(\Delta y)^2}=0 \tag{8}$$

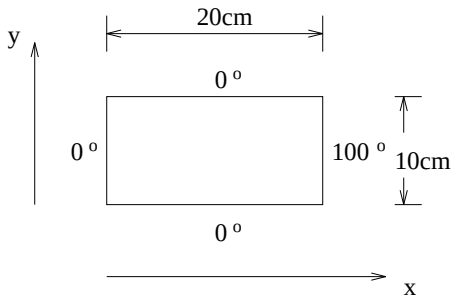
This looks like a bit of a mess, but if $\Delta x=\Delta y=h$ and you write $u_{i,j}$ instead of $u(x_i,y_i)$, things become a little simpler:

$$\nabla^2 u_{i,j}=\frac{1}{h^2}[u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}-4u_{i,j}]=0 \tag{9}$$

This is much easier to remember if you think about it geometrically: $u_{i+1,j}$ is the patch to the right of the one you're considering, $u_{i,j+1}$ is the one above it, and so on.

$$\nabla^2 u_{i,j} = \frac{1}{h^2}[u_R + u_L + u_A + u_B - 4u_0] = 0 \tag{10}$$

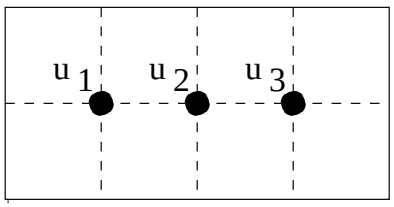
An example will make this much clearer. Consider a thin steel plate, 10cm by 20cm, with the following boundary conditions:



If there are no internal heat sources and all the transients have died out, the physics is described accurately by the steady-state heat equation — equation (5). The boundary conditions shown above translate to the following mathematical conditions on u:

- $u(x, 0) = 0$
- $u(x, 10) = 0$
- $u(0, y) = 0$
- $u(20, y) = 100$

To solve this PDE numerically — that is, to find $u(x, y)$ — you must first choose the grid. This is easily the hardest, subtlest, and most time-consuming part of the whole procedure; doing it well requires years of practice (and sometimes the help of an automatic gridding program). For the purposes of this example, let's assume that $\Delta x = \Delta y = 5cm$, so the grid looks like this:



There are three gridpoints, so we'll end up writing down equation (10) three times: once for each point.

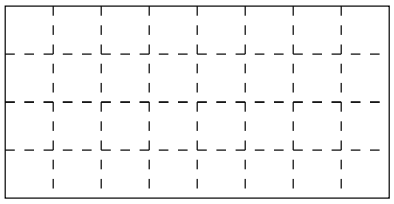
$$at\ u_1 :\ \frac{1}{5^2}[u_2 + 0 + 0 + 0 - 4u_1] = 0$$

$$at\ u_2 :\ \frac{1}{5^2}[u_3 + u_1 + 0 + 0 - 4u_2] = 0$$

$$at\ u_3:\ \frac{1}{5^2}[100 + u_2 + 0 + 0 - 4u_3] = 0$$

This is just a system of three linear equations in three unknowns — a problem that you can solve with any $A\vec{x} = \vec{b}$ solver.

If you use a finer grid, like this one:



...everything is the same, except that there are more unknowns $u_{i,j}$ and more equations — 21 of each for the picture above, and hence a 21×21 $A\vec{x} = \vec{b}$ solve. In general, halving the grid spacing *quadruples* the height and width of the matrix solve. Given that many $A\vec{x} = \vec{b}$ solvers are ($O(n^3)$ in the size of the matrix, this is obviously a concern.

Again, if there's an internal heat source at one or more grid points, you have to use Poisson's equation. The only effect on the equations is that the right-hand side of the system is no longer all zeroes. Each grid point's equation is $\frac{1}{h^2}[u_R + u_L + u_A + u_B - 4u_0] = -Q/k$, where Q is the size of the heat source at each grid point.

Accurate solution of a nasty PDE (e.g., turbulent flow in a viscous fluid, as modeled by the Navier-Stokes equations) can require fine resolution and hence lots of grid points, so these matrices can get really big really fast. This is one of the primary driving forces behind the development of really good $A\vec{x} = \vec{b}$ solvers and techniques for parallelizing them.