Math 563 Lecture Notes
Initial value problems (part II)

Spring 2020

The point: One last class of methods (Runge-Kutta) are introduced. The main focus here is on techniques for improving methods and practical concerns - adjusting step sizes, error estimates and deciding when to choose one method over another.

Related reading: For more detail on RK methods, see Ascher & Petzold, Chapter 4. To see how these methods are implemented in software, Matlab's guide `https://www.mathworks.com/help/matlab/math/choose-an-ode-solver.html` is a good illustration.

1 Runge-Kutta methods

Recall that **Euler's method** was derived by expanding y_{n+1} in a Taylor series:

$$y_{n+1} = y_n + hy'_n + O(h^2) \implies y_{n+1} = y_n + hf_n + O(h^2)$$

since $y'_n = f(t_n, y_n)$ from the ODE. A higher-order one-step method (involving only data at t_n) can be derived by using more terms in the Taylor series. With p terms,

$$y_{n+1} = y_n + hy'_n + \cdots + \frac{h^p}{p!} y_n^{(p)} + O(h^{p+1}).$$

To complete the formula, we need computable expressions for the derivatives. Conveniently, all derivatives of y are available in terms of f by differentiating the ODE.

Shorthand: The expressions involved get rather long. For short, we use subscripts t and y to denote partial derivatives, e.g. $f_t = \partial f/\partial t$ and $f_{yy} = \partial^2 f/\partial^2 y$.

Do not confuse the partials with the total derivative d/dt (denoted with a $'$) for $y(t)$; note that $(f(t, y(t)))'$ involves a chain rule, while the partial $\partial f/\partial y$ does not.

Take d/dt of the ODE $y' = f(t, y(t))$ to obtain

$$y'' = \frac{d}{dt}(f(t, y(t))) = \frac{\partial f}{\partial t} + y' \frac{\partial f}{\partial y} = f_t + ff_y,$$

which can be repeated any number of times to obtain

$$y^{(p)}(t) = \text{nasty expression involving } f \text{ and its partials at } (t, y(t)).$$

That is, the p-th derivative is obtained from the $p-1$ formula by

$$y^{(p)}(t) = \frac{\partial}{\partial t} + f\frac{\partial}{\partial y} \text{ applied to the formula for } y^{(p-1)}(t).$$

While the expressions become complicated, it is always possible to write them out involving only f (known) and its partial derivatives (known, but tedious to compute).

Taylor series method: It follows that the Taylor series yields a p-th order method

$$y_{n+1} = y_n + hy'_n + \cdots + \frac{h^p}{p!}y_n^{(p)} + O(h^{p+1}).$$

where $y'_n, \cdots$ are all replaced by the appropriate expressions involving f. The method, while the right order, is undesirable in general because we need to compute partial derivatives in advance, and the expressions are long (expensive to compute!). Thankfully, a better general method exists!

1.1 The Runge-Kutta approach

Note that $y'_n = f(t_n, y_n)$ is reasonable to compute, but y''_n and so on are not. Recall that when approximating functions, we got around the problem by using an **interpolant** - using function data at several points. The same can be done here; however, the catch is that for a one step method, only $y(t_n)$ is known.

The goal of a **Runge-Kutta method** is to use function evaluations

$$f(t + ha, y + hb)$$

to replace the derivatives $y''_n, y'''_n, \cdots$ and get a method with the desired order of accuracy. The number of **stages** is the number of function evaluations in the formula.

To illustrate, we derive a 'two-stage Runge-Kutta method' of the form

$$\begin{aligned} f_1 &= f(t_n, y_n) \\ f_2 &= f(t_n + ch,\ y_n + hbf_1) \\ y_{n+1} &= y_n + h(w_1 f_1 + w_2 f_2) + O(h^3) \end{aligned}$$

for constants b, c, w_1, w_2.

Analogy (integrals): Before proceeding, it is worth noting that if

$$y' = f(t)$$

then the formula has the form

$$y_{n+1} = y_n + \underbrace{h(w_1 f(t_n) + w_2 f(t_n + ch))} + O(h^3).$$

The bracketed term is an approximation to

$$\int_{t_n}^{t_{n+1}} f(s)\,ds.$$

Thus, the Runge-Kutta method for ODEs is analogous to a **composite rule** for integrals, and each step is like an integration rule on that sub-interval using some set of f-values.

Back to the derivation: The goal is to choose the constants so that

$$\frac{1}{h}(y_{n+1} - y_n) = w_1 f_1 + w_2 f_2 + \tau_{n+1}, \qquad \tau_{n+1} = O(h^2). \tag{1.1}$$

The strategy is to **expand in a Taylor series** around (t_n, y_n), then **write in terms of** f and its derivatives and match terms. For shorthand, we use the convention that all quantities are evaluated at (t_n, y_n) unless otherwise noted (e.g. f_t means $\partial f/\partial t$ at (t_n, y_n)).

For the LHS:

$$\begin{aligned} \frac{1}{h}(y_{n+1} - y_n) &= y_n' + \frac{h}{2}y_n'' + O(h^2) \\ &= f + \frac{h}{2}(f_t + f f_y) + O(h^2) \end{aligned} \tag{1.2}$$

With two function evaluations on the RHS, we expect to be able to cancel out the first two terms of the LHS, leaving an $O(h^2)$ error.[1]

For the RHS: first expand f_2 using a Taylor series[2] around (t_n, y_n):

$$\begin{aligned} f_2 &= f_n + (ch)\frac{\partial f}{\partial t}(t_n, y_n) + (hbf_1)\frac{\partial f}{\partial y}(t_n, y_n) + O(h^2) \\ &= f + chf_t + hbff_y + O(h^2) \end{aligned}$$

using the 'at t_n is implied' shorthand. Now plug this into the RHS:

$$\text{RHS} = w_1 f + w_2(f + chf_t + hbff_y) + \tau_{n+1} + O(h^2). \tag{1.3}$$

[1]One might hope to do more because there are more coefficients to choose, but that is not the case

[2]Note: this is a 2d Taylor series, so $f(t_n + A, y_n + B) = f(t_n, y_n) + A(\partial f/\partial t)_n + B(\partial f/\partial y)_n + \cdots$.

Both sides of the proposed formula (1.1) computed; the LHS (1.2) and RHS (1.3) are

$$\begin{aligned} \text{LHS} &= f + \frac{h}{2}(f_t + f f_y) + O(h^2) \\ \text{RHS} &= (w_1 + w_2)f + h(cw_2 f_t + bw_2 f f_y) + \tau_{n+1} + O(h^2). \end{aligned}$$

For the formula to be second order, the $O(1)$ and $O(h)$ terms must match so that τ is $O(h^2)$ (note that one could be more precise here and get an actual expression for τ, but the process is messy).

There are three terms to match: f, f_t and $f f_y$. It follows that

$$w_1 + w_2 = 1, \quad cw_2 = bw_2 = \frac{1}{2}.$$

Letting $w_2 = \theta$, there is a solution for each θ:

$$w_1 = 1 - \theta, \quad b = c = 1/(2\theta).$$

In conclusion, for any $\theta \neq 0$, the method

$$\begin{aligned} f_1 &= f(t_n, y_n) \\ f_2 &= f(t_n + \frac{1}{2\theta}h,\, y_n + \frac{1}{2\theta}h f_1) \\ y_{n+1} &= y_n + h((1-\theta)f_1 + \theta f_2) + O(h^3) \end{aligned}$$

is second-order (for stability, see subsection 1.5).

Note there is a free parameter. We may hope, then, that one more term in the Taylor series can cancel. However, there is no way to match the RHS (1.3) with the LHS term

$$h^2 y_n'''/2$$

no matter how the coefficients are chosen (exercise); the best possible order is two.

For example, if $w_1 = 0$ and $w_2 = 1$ then we get the **modified Euler method**

$$\begin{aligned} f_1 &= f(t_n, u_n) \\ f_2 &= f(t_n + h/2, u_n + h f_1/2) \\ u_{n+1} &= u_n + h f_2 \end{aligned}$$

One can interpret modified Euler as using the midpoint rule to estimate

$$y(t_{j+1}) = y(t_n) + \int_{t_j}^{t_{j+1}} f(t, y(t))\, dt \approx y(t_j) + h f(t_j + h/2, y(t_j + h/2))$$

but using Euler's method to estimate the midpoint value:

$$y(t_j + h/2) \approx y_n + \tfrac{h}{2} f(t_j, y_j).$$

When $f = f(t)$, this reduces exactly to the composite midpoint rule.

The classical RK-4 method: There are many free parameters for ≥ 3 stage methods, leading to several families of methods of practical use. One four stage method of note is the classical 'RK-4' method

$$\begin{aligned}
f_1 &= f(t_n, y_n)\\
f_2 &= f(t_n + \frac{1}{2}h, y_n + \frac{h}{2}f_1)\\
f_3 &= f(t_n + \frac{1}{2}h, y_n + \frac{h}{2}f_2)\\
f_4 &= f(t_n + h, y_n + f_3)\\
y_{n+1} &= y_n + \frac{h}{6}(f_1 + 2f_2 + 2f_3 + f_4) + O(h^5).
\end{aligned}$$

(Exercise: this formula reduces to an integration rule when $y' = f(t)$ - what is it?).

This method has a good balance of efficiency and accuracy - only four function evaluations per step, and fourth-order accuracy. For solving a general ODE with a fixed time step, there is no better simple method to try.

However, it is not used in practice much, because there is a different RK method (same idea, different coefficients) that is better for non-constant step size (see subsection 2.2.

1.2 Higher-order formulas

The process can be used to build methods of any order, assuming one is willing to suffer through the complicated algebra. Hopefully, it is clear that the process of deriving equations for the coefficients is straightforward, even if the details are messy. Because they are so popular, some discussion is worthwhile:

Definition (RK methods) A general '**explicit Runge-Kutta** (RK) method' uses m 'stages' to get from u_n to u_{n+1} and has the form

$$\begin{aligned}
f_1 &= f(t_n, u_n)\\
f_2 &= f(t_n + c_2h, u_n + ha_{21}f_1)\\
f_3 &= f(t_n + c_3h, u_n + ha_{31}f_1 + ha_{32}f_2)\\
&\vdots\\
f_m &= f(t_n + c_mh, u_n + ha_{m1}f_1 + \cdots + ha_{m,m-1}f_{m-1})\\
u_{n+1} &= u_n + h(w_1f_1 + \cdots + w_mf_m).
\end{aligned}$$

Each stage f_j is an evaluation of f at an intermediate point $(t_n + c_jh, \tilde{u}_j)$ where $\tilde{u}_j$ involves a linear combination of previous f's. The next value u_{n+1} is a weighted average of the f's.

The coefficients are typically written in an array called a **Butcher Tableau**:

$$\begin{array}{c|ccccc}
c_1 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
c_m & a_{m1} & a_{m2} & \cdots & a_{m,m-1} & \\
\hline
 & w_1 & w_2 & \cdots & w_{m-1} & w_m
\end{array}$$

An **implicit** Runge-Kutta method has the same form, but the f's can depend on any of the others (not just previous ones):

$$f_k = f(t_n + c_n h,\, y_n + h \sum_{j=1}^{m} a_{kj} f_j).$$

That is, the Butcher table can be square (with a_{kj} non-zero for all k, j, not just $j < k$).

The tables for modified Euler,classical RK4 and the trapezoidal rule (implicit) are

$$\text{Trap.: } \begin{array}{c|cc}
0 & & \\
1 & & \\
\hline
 & 1/2 & 1/2
\end{array}, \quad \text{M. Euler: } \begin{array}{c|cc}
0 & & \\
1/2 & 1/2 & \\
\hline
 & 0 & 1
\end{array}, \quad \text{RK4: } \begin{array}{c|cccc}
0 & & & & \\
1/2 & 1/2 & & & \\
1/2 & 0 & 1/2 & & \\
1 & 0 & 0 & 1 & \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}$$

Despite the fact that there are more than m coefficients for an m-stage method, the order of accuracy is constrained. The result:

Theorem (Butcher): An m-stage RK method can be at best m-th order.

- For $m \leq 4$, the best possible order is exactly m.
- For $m > 4$, the best possible order is strictly less than m. For instance, $m = 6$ stages are required to get a fifth-order method.

For this reason, the 'sweet spot' for RK methods is around four stages, since it provides an optimal gain in efficiency per step. For each number of stages, there are a large number of valid methods of each possible order; which one to use depends on what other properties are desired.

1.3 Systems

The RK formulas extend trivially to first order systems

$$\mathbf{y}' = F(t, \mathbf{y}).$$

The formulas are exactly the same, which is a nice feature. The simplicity of implementation (evaluate F at some points, add things together) makes RK methods quite popular for general ODE solving routines. If vector arithmetic is used (e.g. in Matlab) than the formulas can even be written in the same form, such as

$$\mathbf{f}_2 = f(t_n + c_2 h, \mathbf{y}_n + h a_{21} \mathbf{f}_1)$$

if the code allows for vectors to be scaled/added.

1.4 Convergence

n explicit RK method can be written in the form

$$u_{n+1} = u_n + h\psi(h, t_n, y_n)$$

since the step only depends on t_n, y_n and intermediate quantities derived from these values. From this formula, one can prove zero-stability in the same was as Euler's method if one has

$$\psi(h, t, y) \text{ is Lipschitz in } y.$$

It is not too hard to show that if $f(t, y)$ is Lipschitz in y than so is ψ; then the rest of the zero-stability proof is the same as for Euler (but with f replaced by ψ).

1.5 Absolute stability

Observe that, applied to the test equation, (with $z = h\lambda$)

$$\begin{aligned} hf_1 &= h\lambda u_n = z u_n \\ hf_2 &= h\lambda(u_n + a_{21}(hf_1)) = (z + a_{21} z^2) u_n \\ \vdots &= \vdots \\ hf_j &= (\text{poly. of deg. } j \text{ in } z) u_n \end{aligned}$$

so it follows that

$$u_{n+1} = u_n + \sum_{j=1}^{m} w_j (hf_j) = q(z) u_n$$

where the 'growth factor' $q(z)$ is a polynomial of degree m. It follows that the stability region for any explicit RK method is bounded, since

$$|q(z)| \sim C|z|^m \text{ as } |z| \to \infty,$$

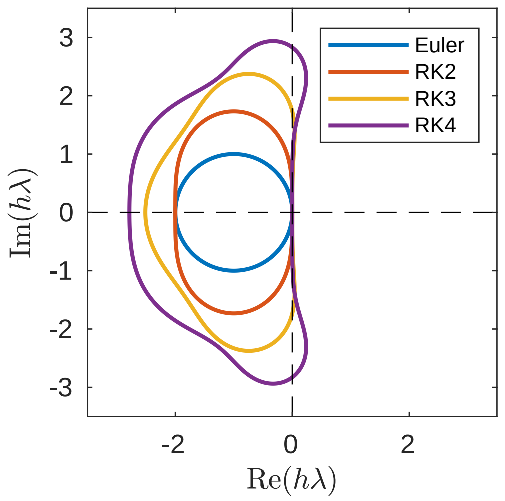


Figure 1: Stability regions for explicit m-stage, order m RK methods (see Remark below). The regions are inside the curves.

i.e. outside a large enough circle in the complex plane, $|q(z)| > 1$ so u_n grows in magnitude at each step. This means that **all explicit RK methods have stability constraints**.

For example, one can calculate (left as an exercise) that for the classical RK4 method,

$$u_{n+1} = \left(1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24}\right) u_n, \qquad z = h\lambda.$$

It follows that

$$R = \{z \in \mathbb{C} : \left|1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24}\right| < 1\}.$$

The interval of absolute stability is about $(-2.7, 0)$, which is not really any better than Euler's method. Thus, while RK4 is much more accurate, it is not more stable than Euler (and has the same issues).

Remark (why the exponential-ness?): The form of the stability region is not a coincidence. We know that $u_{n+1} = q(z)u_n$ for a degree four polynomial q and the method has an order of accuracy of 4, so

$$u_1 = y_1 + O(h^5).$$

But $y_1 = u_0 e^{h\lambda}$ so

$$q(z)u_0 = e^z u_0 + O(z^5)$$

so $q(z)$ and e^z must match up to z^4 terms. (Note: the argument here does not generalize to any RK method, only methods with m stages that are order m).

In fact, it follows from this that all m-stage methods of order m have the same stability region, which is why the legend in Figure 1 can refer to 'RKm' without ambiguity.

2 Adaptive time stepping

Notation: For this section, we denote by

$$e_n = h\tau_n$$

the **local error** in the n-th step. The reason is that we need to discuss the actual error incurred at each step, so it is useful to switch from the 'τ_n-matches-the-global-order' convention to the actual local error.

Some of the calculations here are informal; they are used to get practical estimates and may not be rigorous (in fact, some may even fail in bad cases!). The hand-waving is part of the subject here - we sacrifice rigor to get convenient estimates (for code).

Notice that one step methods use only the values at the previous time step n to update to time $n+1$. Thus, at each step, a new value of h can be chosen.

To be efficient, the algorithm should be able to choose an h just small enough to meet the desired needs for error. The simplest approach is to request a tolerance ϵ and attempt **local error control** by requiring that

$$|e_n| < \epsilon \text{ for all } n.$$

A more sophisticated algorithm might attempt to instead control the global error, e.g.

$$\max_n |y(t_n) - u_n| < \epsilon.$$

However, **global error control** is much more difficult, and will not be pursued here. In practice, one typically settles for the local variant and accepts that there may not be a rigorous guarantee on the global error.

Remark (relative error): Both relative and absolute error can be the right measure to control. Good algorithms take both into account, so the user may input both a relative and absolute tolerance ϵ_R and ϵ_A, and then impose that

$$|e_n| \leq \epsilon_A + |u_n|\epsilon_R$$

or, if you prefer, two separate bounds on $|e_n|$ and $|e_n|/|u_n|$. For simplicity, the treatment below assumes only an absolute error bound, but you should consider putting the combined version in your own codes.

2.1 The simple way: two methods of different order

Suppose that a step size h is taken to get from u_{n-1} to u_n and we have:

i) Method A: Order p, producing u_n from u_{n-1}, local error $e_n(h)$

ii) Method B: Order $p+1$, producing $\tilde{u}_n$ from u_{n-1} with an $O(h^{p+2})$ local error

The goal is to find a step size h^* such that

$$e_n(h^*) < \epsilon$$

for a given tolerance ϵ (i.e. to bound the local error in Method A).

To derive the estimate, suppose the value at time $n-1$ is exact. Then

$$y(t_n) = u_n + e_n(h) \tag{2.1}$$
$$y(t_n) = \tilde{u}_n + O(h^{p+2}). \tag{2.2}$$

Now assume that $e_n(h)$ has a 'leading term', so that

$$e_n(h) = Ch^{p+1} + O(h^{p+2}).$$

First, note that using this estimate, our objective is to find h^* so that

$$\epsilon > e_n(h^*) \approx C(h^*)^{p+1} = (h^*/h)^{p+1} e_n(h)$$

so it suffices to get an estimate for $e_n(h)$, then solve for h^* above.

Next, by subtracting the two formulas for Methods A and B, (2.1) and (2.2), we get

$$0 = u_n - \tilde{u}_n + Ch^{p+1} + O(h^{p+2}) \implies e_n(h) \approx |u_n - \tilde{u}_n|.$$

Finally, all the pieces are computable. We should choose h^* so that

$$(h^*/h)^{p+1} < \frac{\epsilon}{|u_n - \tilde{u}_n|}$$

which provides a soft (non-rigorous) guarantee that the local error is bounded by ϵ.

In practice, to be safe, one adds a 'safety factor' between 0 and 1 and sets

$$h_{\text{new}} = (\text{safety}) \cdot h \left(\frac{\epsilon}{|u_n - \tilde{u}_n|} \right)^{1/(p+1)}. \tag{2.3}$$

In summary, the algorithm proceeds (from u_{n-1} to u_n) as follows:

- Use 'the method' to compute u_n from u_{n-1} and an 'auxiliary' method of higher order to get another approximation $\tilde{u}_n$
- Find h^* such that the local error would be $< \epsilon$ (roughly) using (2.3)
- Accept u_n as the next value and go to the next step (or, in practice, use $\tilde{u}_n$ instead; or, to be careful, reject the step and try again)

The recipe can be modified as needed, depending on how reliable the estimate must be.

2.2 Embedded RK methods

In general, 'two methods' approach to error control costs about twice as much as using a fixed step. However, with a judicious choice, we can create some overlap in the computations, saving work.

One of the main approaches is to use an **embedded pair** of Runge-Kutta methods, which is a pair of RK formulas where most of the f_i's are the same for both.

For example, consider an adaptive version of Euler's method

$$u_{n+1} = u_n + hf(t_n, u_n)$$

using an order 2 method for the error estimate. We can tack on the Euler step for free to the modified Euler method (recall this has order 2):

$$\begin{aligned} f_1 &= f(t_n, u_n) \\ f_2 &= f(t_n + \frac{1}{2}h, u_n + \frac{h}{2}f_1) \\ \tilde{u}_{n+1} &= u_n + hf_2 \text{ (Mod. Euler)} \\ u_{n+1} &= u_n + hf_1 \text{ (Euler)} \end{aligned}$$

Now only two function evaluations total are done, because f_1 is shared between both methods (we say Euler's method is **embedded** here).

Following (2.3), the timestep would then be chosen to be something like

$$h_{\text{new}} \approx (\text{safety}) \cdot h(\epsilon/|u_{n+1} - \tilde{u}_{n+1}|)^{1/2}.$$

The popular method: Recall that fourth-order RK methods had a good balance of efficiency/accuracy. There are also good embedded pairs of fourth/fifth order methods that allow the adaptive step size selection to be implemented.

Two popular pairs are the **Runge-Kutta-Fehlberg** (RKF) and **Dormand-Prince** pairs (the latter is used by Matlab's `ode45`, for example), the tables for which are easy to find. The adaptive step sizes allow the method to be quite efficient on non-stiff problems, and it is both easy to use and implement (both for scalar equations and systems). For these reasons, RKF and its variants are a good first choice when trying to solve an ODE (the 'default').

3 Newton's method (the quick version)

Solving implicit systems of ODEs requires solving non-linear systems

$$\mathbf{F}(\mathbf{x}) = 0, \qquad \mathbf{F} : \mathbb{R}^m \to \mathbb{R}^m$$

for a solution $\mathbf{x}^*$. We will not delve into the extensive array of methods here, instead introducing only **Newton's method**, the most useful for solving ODEs (and in general).

3.1 Scalar version

In one dimension, Newton's method solves a non-linear equation

$$f(x) = 0$$

by generating a sequence of 'iterates' $\{x_n\}$ converging to a solution x^* such that $f(x^*) = 0$.

The method: Given a point x_n and data $f(x_n)$ and $f'(x_n)$, we can estimate the solution by using a linear approximation. Via Taylor series:

$$f(x) \approx \underbrace{f(x_n) + f'(x_n)(x - x_n)}_{\ell(x)} + \frac{f''(\xi_n)}{2}(x - x_n)^2. \tag{3.1}$$

The line $\ell(x)$ is the tangent line through x_n. Its intersection with the x-axis (where $\ell(x) = 0$) gives an approximation to x^*. We take this value as the next iterate:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{3.2}$$

This iteration is computed, in practice, until $|x_{n+1} - x_n|$ is sufficiently small - that is, the update from x_n to x_{n+1} is small enough. Note that this is not a bound on the actual error, but due to the rapid convergence (see below), it tends to be good enough.

Convergence: To show that it converges, let $e_n = x^* - x_n$ be the error. Plug $x = x^*$ into the Taylor series (3.1) to get

$$0 = f(x_n) + f'(x_n)e_n + \frac{f''(\xi_n)}{2}e_n^2.$$

By the definition of the method, $e_{n+1} = e_n + f(x_n)/f'(x_n)$ so

$$e_{n+1} = \frac{f''(\xi_n)}{2f'(x_n)}e_n^2. \tag{3.3}$$

We can show from here that if x_n is close enough to x^* then the error is decreasing, hence x_n converges to x^*. Then, since ξ_n is between x_n and x^*, we have $\xi_n \to x^*$ so

$$\lim_{n\to\infty} \frac{e_{n+1}}{e_n^2} = \frac{f''(x^*)}{2f'(x^*)} := C.$$

This shows **quadratic convergence** - the next error looks like the square of the previous:

$$e_{n+1} \approx Ce_n^2.$$

This error decays quite rapidly (e.g. with $C = 1$ and $e_0 = 10^{-2}$, we get $10^{-4}, 10^{-8}, 10^{-16}$, reaching machine precision in four steps!). Contrast with **linear convergence** where

$$e_{n+1} \approx Ce_n.$$

Here, a fixed number of significant digits is added per iteration (e.g. $10^{-1}, 10^{-2}, 10^{-3}, \cdots$).

Theorem (simplified convergence result): Suppose $f(x^*) = 0$ and $f'(x^*) \neq 0$ (a simple zero) and $f \in C^2$ in a neighborhood of x^*. Then if the initial value x_0 is close enough to x^*, Newton's method converges to x^* and the convergence is quadratic as given by (3.3).

Precisely, convergence holds if, in an interval $I = (x^* - \epsilon, x^* + \epsilon)$ containing x_0,

$$\left(\frac{\max_I |f''|}{2\min_I |f'|}\right) |x_0 - x^*| < 1$$

so that $|e_1| < |e_0|$ and $x_1 \in I$. Better results exist; see the **Newton-Kantorovich theorem**.

Key point (accurate, but not reliable): Newton's method tends to converge extremely fast - typically reaching machine precision in only a few steps. However, the downside is that one needs a good initial guess - it has to be close to the true zero, and 'close enough' can be hard to identify. Thus, Newton's method is best used when the starting point is already close.

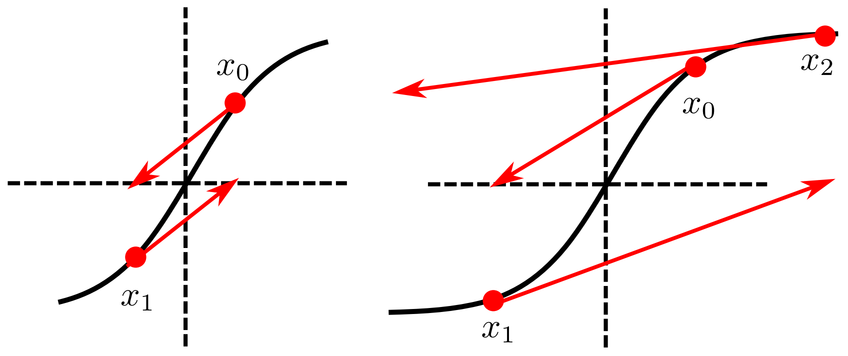
Example: There are three typical behaviors for a Newton iteration. Consider

$$f(x) = \tanh(x)$$

and its zero $x^* = 0$. There is a value a such that

- If $|x_0| < a$ then $|x_{n+1}| < |x_n|$, and the sequence will converge quadratically to x^*
- If $|x_0| = a$ then $x_{n+1} = -x_n$; the sequence will 'bounce' around zero (no convergence).
- If $|x_0| > a$ then $|x_{n+1}| > |x_n|$ and the sequence diverges (rapidly).

If the zero is simple, this means that typically, Newton's method will either converge very quickly, blow up, or oscillate wildly, so the 'failure' cases are easy to spot. The bad cases are shown below (using the 'follow the tangent line' interpretation to sketch):



3.2 For systems

Now, we turn to the problem of solving a system of m equations

$$\mathbf{F}(\mathbf{x}) = 0, \qquad \mathbf{F} : \mathbb{R}^m \to \mathbb{R}^m.$$

Let $\mathbf{J}(\mathbf{x})$ be the Jacobian of $\mathbf{F}$ at $\mathbf{x}$, i.e. the matrix with entries $\partial F_i / \partial x_j$.

Let us assume that $\mathbf{x}^*$ is a solution, $\mathbf{F}$ is C^2 and

$$J(\mathbf{x}^*) \text{ is invertible.} \tag{3.4}$$

The method produces a sequence $\{\mathbf{x}_n\}$ of approximations, hopefully converging to $\mathbf{x}^*$. Given $\mathbf{x}_n$, the next step is derived by estimating $\mathbf{F}$ with a Taylor series around $\mathbf{x}_n$:

$$\mathbf{F}(\mathbf{x}_n) = \underbrace{\mathbf{F}(\mathbf{x}_n) + J(\mathbf{x}_n)(\mathbf{x} - \mathbf{x}_n)} + O(\|\mathbf{x} - \mathbf{x}_n\|^2).$$

The bracketed term is a linear approximation to $\mathbf{F}$. As in the scalar case, choose the 'next' iterate $\mathbf{x}_{n+1}$ as the point where this approximation is zero:

$$0 = \mathbf{F}(\mathbf{x}_n) + J(\mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n).$$

Rearranging yields Newton's method for systems,

$$\mathbf{x}_{n+1} = \mathbf{x}_n - J_n^{-1}\mathbf{F}(x_n), \qquad J_n := \text{ Jacobian at } \mathbf{x}_n. \tag{3.5}$$

In practice, it is best to solve a linear system instead of inverting J, so the actual steps are:

- Compute J_n and solve the linear system $J_n \mathbf{v} = -\mathbf{F}(\mathbf{x}_n)$ for an 'update' $\mathbf{v}$
- Update $\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}$.
- If $\|\mathbf{v}\|$ is small enough, stop.

The convergence results transfer to the n-dimensional case:

Theorem (convergence for systems, roughly) Suppose $\mathbf{x}^*$ is a zero of $\mathbf{F}$ and $J(\mathbf{x}^*)$ is invertible and $\mathbf{F}$ is C^2. If the initial value $\mathbf{x}_0$ is close enough to $\mathbf{x}^*$ then Newton's method converges quadratically.

A more precise result is given by the **Newton-Kantorovich theorem.**

Key point: A good guess is even more valuable in $\mathbb{R}^n$ since there are more dimensions to work with - which means more possibilities for failure. Without a good guess, Newton's method will likely not work.

Moreover, constructing a guess is much harder with more dimensions - one typically needs some intuition from the problem to find a reasonable guess.

3.3 Newton's method example

(Also a preview of BVPs for later). The system of N equations

$$\begin{aligned} 2y_1 - y_2 &= \cos(y_1) \\ -y_{i-1} + 2y_i - y_{i+1} &= \cos(y_i), \qquad i = 2, \cdots N-1 \\ -y_{N-1} + 2y_N &= \cos(y_N) \end{aligned} \tag{3.6}$$

are a finite difference approximation for the **boundary value problem**

$$-y'' = \cos(y), \quad y(0) = y(1) = 0$$

for equally spaced data $\{y_j\}$ at points $x_0, \cdots, x_N$ in $[0,1]$. The set of equations is a non-linear system for the data $\mathbf{y} = (y_1, \cdots, y_N$ of the form

$$\mathbf{F}(\mathbf{y}) = 0$$

where the i-th component of $\mathbf{F}$ is

$$\mathbf{F}_i(\mathbf{y}) = -y_{i-1} + 2y_i - y_{i+1} - \cos(y_i), \qquad i = 2, \cdots N-1.$$

and the exceptional cases

$$\begin{aligned} \mathbf{F}_1(\mathbf{y}) &= 2y_1 - y_2 - \cos(y_1), \\ \mathbf{F}_N(\mathbf{y}) &= -y_{N-1} + 2y_N - \cos(y_N). \end{aligned}$$

To apply Newton's method, we first compute the Jacobian $\mathbf{J}(\mathbf{y})$ (letting $c_k = \cos(y_k)$)

$$\mathbf{J}(\mathbf{y}) = \begin{bmatrix} 2+c_1 & -1 & 0 & \cdots & 0 \\ -1 & 2+c_2 & -1 & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & -1 & 2+c_{N-1} & -1 \\ 0 & \cdots & 0 & -1 & 2+c_N \end{bmatrix}.$$

Note that the Jacobian of $\mathbf{F}$ is a tridiagonal matrix. At step k, we solve the linear system

$$(\mathbf{J}(\mathbf{y}_k))\mathbf{v}_k = -\mathbf{F}(\mathbf{y}_k)$$

which is efficient to solve because the matrix is tri-diagonal (it requires $O(N)$ operations). Then, update

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \mathbf{v}_k$$

and iterate until convergence; the vector $\mathbf{y}_k$ will approach a solution to the system (3.6).

Picking an initial guess is non-trivial: a good guess at the solution to the boundary value problem would be best. Here, approximating $\cos(y) \approx 1$ yields a solution $y(x) = \frac{1}{2}x(1-x)$, which would provide an initial $\mathbf{y}_0$ close to the solution.

4 Implicit methods (implementation)

Suppose the first order IVP

$$\mathbf{y}'(t) = \mathbf{F}(t, \mathbf{y}), \qquad \mathbf{y}(0) = \mathbf{y}_0$$

is to be solved using Backward Euler,

$$\mathbf{u}_n = \mathbf{u}_{n-1} + h\mathbf{F}(t_n, \mathbf{u}_n).$$

A backward Euler solver will have the user input $\mathbf{F}$ and, if Newton's method is used, also input the Jacobian with respect to $\mathbf{y}$, i.e. the matrix $J_F(t, \mathbf{y})$ such that

$$(J_F)_{ij} = \frac{\partial \mathbf{F}_i}{\partial \mathbf{y}_j}.$$

We re-formulate 'solving for $\mathbf{u}_n$' in the formula as finding a zero of

$$\mathbf{G}(\mathbf{z}) = \mathbf{z} - \mathbf{u}_{n-1} - h\mathbf{F}(t_n, \mathbf{z}).$$

This sub-problem of 'find a zero of $\mathbf{G}$' must be done at each step. The Jacobian of $\mathbf{G}$ is

$$J_G(\mathbf{z}) = I - hJ_F$$

so it can be computed from the user input (the user does not have to worry about the internal workings of Newton's method).

To obtain $\mathbf{u}_n$, we apply Newton's method to get iterates

$$\mathbf{z}_{k+1} = \mathbf{z}_k - (J_G(\mathbf{z}_k))^{-1}\mathbf{G}(z)$$

until some tolerance is reached (usually very small, near machine precision so it doesn't spoil the accuracy). Then the result becomes the new value $\mathbf{u}_n$.

Because $\mathbf{u}_n \approx \mathbf{u}_{n-1}$ if the step size is small, we have an obvious initial guess of the previous value, i.e. $\mathbf{z}_0$ should be $\mathbf{u}_{n-1}$. Taking h small enough will ensure convergence.

Practical note (modularity): The solver should consist of an outer loop over times $t_0, t_1, \cdots, t_N$ in which each step is taken (to get to the approximation at the next time). This 'outer loop' is the same as for an explicit method.

At each step, the implicit sub-problem must be solved using Newton's method. It's helpful to think of this sub-part as its own isolated problem to be completed before moving onto the next time.

You could write a sub-routine (e.g. a generic Newton's method solver) to be called at each step, thus separating the implementation into two separate pieces that can be tested separately ('modular' code).

Example: Consider the pendulum equation

$$\theta'' + \sin\theta = f(t)$$

which can be written as the system

$$\mathbf{x}' = \mathbf{F}(t,\mathbf{x}), \qquad \mathbf{F}(t,\mathbf{x}) = \begin{bmatrix} x_2 \\ -\sin x_1 + f(t) \end{bmatrix}$$

and $\mathbf{x} = (\theta, \theta')$. The Jacobian (to be input to the solver) is

$$J_F = \begin{bmatrix} 0 & 1 \\ -\cos z_1 & 0 \end{bmatrix}$$

Suppose we apply Backwards Euler to this system:

$$\mathbf{x}_n = \mathbf{x}_{n-1} + h\mathbf{F}(t_{n-1}, \mathbf{x}_n)$$

and have values up to time $n-1$. To obtain $\mathbf{x}_n$, construct

$$\mathbf{G}(\mathbf{z}) = \mathbf{z} - \mathbf{x}_{n-1} - h\mathbf{F}(t_n, \mathbf{z}).$$

Then set $\mathbf{z}_0 = \mathbf{x}_{n-1}$ and iterate using Newton's method. Given $\mathbf{z}_k$, compute

$$\begin{aligned} (I - hJ_F(t_n, \mathbf{z}_k))\mathbf{v} &= -\mathbf{G}(\mathbf{z}_k) \quad \text{(for the update } \mathbf{v} \text{)} \\ \mathbf{z}_{k+1} &= \mathbf{z}_k + \mathbf{v} \end{aligned}$$

Then stop once $\|\mathbf{v}\|$ is small (say, around 10^{-12} or less) and set $\mathbf{u}_n$ to the output $\mathbf{z}$-vector.

5 Operator splitting

The goal here is to introduce a key idea in numerical analysis (broadly applicable) by way of a simple example. The idea of **operator splitting** is to take a process that involves the sum of two operators, like the PDE

$$\frac{\partial u}{\partial t} = L_1 u + L_2 u$$

and approximate it by applying each part separately (step forward by Δt with just L_1, then just L_2 and so on).

For example, consider a 'convection-diffusion' process

$$\frac{\partial u}{\partial t} = -c\frac{\partial u}{\partial x} + k\frac{\partial^2 u}{\partial x^2},$$

which describes a distribution $u(x,t)$ of stuff transported at speed c and spreading out. The two processes are coupled together, which can complicate taking a time step Δt forward. However, one could try to approximate the process by taking one step of the convection part (transport it to the right by c), then one step of the diffusion part. Then, one only has to numerically solve simpler equations

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0, \qquad \frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2}$$

each of which has only one effect. As another example, consider diffusion in 2d,

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}, \qquad 0 < x, y < 1$$

in a square domain. We can think of this as two processes: diffusion in the x-direction and in the y-direction. One might hope, then, to solve numerically by taking a step of x-diffusion along lines $y =$ const. (which are simpler 1d problems), then a step of y-diffusion along lines $x =$ const.. This reduces a two-dimensional problem with N^2 points to two sets of N one-dimensional problems (with N points).

The subtlety is that there is an error associated with splitting, since taking the steps separately is not the same as all at once. The numerical properties can lead to both quantitative and qualitative differences (stability, long term behavior).

5.1 The example equation

The idea is best understood by studying a simple example. Consider the ODE

$$\frac{d\mathbf{y}}{dt} = A\mathbf{y} + B\mathbf{y}$$

where A and B are $n \times n$ matrices. Recall (from ODE theory) that the solution is

$$\mathbf{y}(t) = e^{(A+B)t}\mathbf{y}_0$$

where e^A is the matrix exponential $e^A = I + A + A^2/2 + \cdots = \sum_{j=0}^{\infty} A^j/j!$.

We can try to 'split' the equation into an A part and a B part. Suppose we first apply the B part, then the A part as follows:

$$\mathbf{w} = \text{solution to } \mathbf{y}' = B\mathbf{y},\ \mathbf{y}(0) = \mathbf{y}_0,$$

$$\mathbf{y} = \text{solution to } \mathbf{y}' = A\mathbf{y},\ \mathbf{y}(0) = \mathbf{w}(t).$$

The 'solution operator' of $\mathbf{y}' = A\mathbf{y}$ is e^{At}; it advances the solution by a time t. We can think of the splitting as approximating the solution operator for the full ODE with the operators for $\mathbf{y}' = A\mathbf{y}$ and $\mathbf{y}' = B\mathbf{y}$.

In the scalar case, the splitting is exact. The ODE solution is

$$y(t) = e^{(a+b)t}y_0.$$

Using the splitting, we first solve the b-part to get

$$w(t) = e^{bt}y_0,$$

(i.e. apply the solution operator e^{bt} to y_0). Then, solve the a-part, given this result (i.e. apply the solution operator e^{at}):

$$y(t) = e^{at}(e^{bt}y_0) = e^{(a+b)t}y_0.$$

However, the splitting is **not exact** for systems. The splitting gives

$$\mathbf{w}(t) = e^{Bt}\mathbf{y}_0, \quad \mathbf{y}(t) = e^{At}e^{Bt}\mathbf{y}_0.$$

In comparison:

$$\begin{aligned} \text{actual solution:} \quad & \mathbf{y}(t) = e^{(A+B)t}\mathbf{y}_0, \\ \text{splitting solution:} \quad & \mathbf{y}(t) = e^{At}e^{Bt}\mathbf{y}_0, \end{aligned}$$

However, matrix multiplication is not commutative, and in particular,

$$e^{(A+B)t} \neq e^{At}e^{Bt} \text{ unless } A \text{ and } B \text{ commute.}$$

Fortunately, they are equal up to an error of size $O(t^2)$. It is straightforward to show that

$$e^{(A+B)t} = e^{At}e^{Bt} + O(t^2).$$

This suggests the splitting approximation may work. In practice, we should not just jump from 0 to t. Instead, we take time steps of size h, alternating between the A and B parts:

$$\begin{aligned} \mathbf{w}_n &= e^{hB}\mathbf{w}_{n-1} \\ \mathbf{u}_n &= e^{hA}\mathbf{w}_n \end{aligned}$$

Since e^{hB} means 'solve $\mathbf{y}' = B\mathbf{y}$ up to time h', we can approximate this operator by a regular ODE method. This yields

$$\mathbf{u}_n = e^{hA}e^{hB}e^{hA}e^{hB}\cdots e^{hA}e^{hB}\mathbf{u}_0$$

and since

$$e^{hA}e^{hB} = e^{h(A+B)} + O(h^2)$$

the method can be expected to be first order (local error of $O(h^2)$).

Note that if Euler's method is used for this example ODE then

$$\mathbf{u}_n = \mathbf{w}_n + hA\mathbf{w}_n = (I + hA)\mathbf{w}_n$$

so we see that using Euler's method can be thought of as approximating the solution operator e^{hA} by $I + hA$ (its first two terms in the series). The splitting can be checked by noting that

$$\underbrace{(I + hA)(I + hB)}_{\text{split Euler}} = \underbrace{I + h(A + B)}_{\text{Euler}} + O(h^2).$$

A slight variation can improve the order by 1. The observation is that

$$e^{hA/2}e^{hB}e^{hA/2} = e^{h(A+B)} + O(h^3).$$

Then we can proceed by taking split steps as above. Notice that combining them, the first term of a step merges with the last term of the next:

$$\mathbf{y}_n = e^{hA/2}e^{hB}e^{hA}\cdots e^{hB}e^{hA/2}\mathbf{y}_0$$

so one proceeds in practice by taking a half step at t_0, then full steps alternating between A and B, then a half step of A at the end. This is called **Strang splitting**.

5.2 More generally

Splitting is commonly used in solving PDEs, where dealing with several types of terms at once can be challenging (even if each part is simpler). While the solution structure is different, it is sometimes close enough that the process still works.

A version of splitting is a key idea in optimization - for instance, minimizing a complicated problem by minimizing parts one at a time and iterating (see **alternating direction** methods).

Splitting can be used to separate a 'stiff' part of a DE from a 'non-stiff' part, allowing an implicit method to be used on one part and an explicit method on the other. This isolation of stiff terms can help the solver and improve efficiency.

As a simple example (just to illustrate the point), to solve

$$y' = -100y + \sin(y),$$

we are required to solve the implicit equation at each step (taking a few Newton iterations). One could try, instead, to solve

$$\tilde{y}_{n+1} = y_n - h\lambda\tilde{y}_{n+1}, \quad y_{n+1} = \tilde{y}_{n+1} + h\sin(\tilde{y}_{n+1}).$$

Then the implicit equation is **linear** so it is easy to solve. Of course, more work is required to improve the order of accuracy, but it does greatly simplify the implicit part. More generally,

$$\mathbf{y}' = A\mathbf{y} + g(t, y)$$

can be split if the $A\mathbf{y}$ part is stiff; then one has to solve one linear system per step instead of a handful with Newton's method.

5.3 Step doubling

Now suppose instead that we have only a single method that can take a step size h (for any h) and wish to choose a new step size h_{new} such that the local error is at most ϵ. By using (Richardson) extrapolation, an error estimate and a new step size can be obtained.

The idea is to use **step doubling**. Suppose, for the sake of example, that our method is a one step method of the form

$$y_{n+1} = y_n + h\psi(t_n, y_n) + O(h^{p+1}).$$

Assume y_n is exact; then the next step is

$$u_{n+1} = y_n + h\psi(t_n, y_n).$$

Now take two steps of size $h/2$ (see **??**, right) to get a new approximation:

$$\hat{y}_{n+1/2} = y_n + \frac{h}{2}\psi(t_n, y_n),$$
$$\hat{y}_{n+1} = \hat{y}_{n+1/2} + \frac{h}{2}\psi(t_n, \hat{y}_{n+1/2}).$$

Assume that each application of a step creates a LTE

$$\tau \approx Ch^{p+1}$$

and that C is a single constant.[3] Then, if y_n is exact, we have

$$y_{n+1} \approx y(t_{n+1}) + Ch^{p+1}$$
$$\hat{y}_{n+1} \approx y(t_{n+1}) + 2C(h/2)^{p+1}$$

i.e. the 'doubled' method accumulates two truncation errors from a step size $h/2$. Subtracting the two approximations gives

$$|y_{n+1} - \hat{y}_{n+1}| \approx (1 - 2^{-p})|Ch^{p+1}|.$$

Thus the error estimate is

$$|Ch^{p+1}| \approx \frac{|y_{n+1} - \hat{y}_{n+1}|}{1 - 2^{-p}}$$

from which we can choose a new time step h_{new} such that $C(h_{\text{new}})^{p+1} < \epsilon$.

The advantage of step doubling is its simplicity. Given a 'black box' solver of a known order, we may still use step doubling to get an error estimate. The disadvantage is that it requires three times as much work as a single step of size h.

6 Additional notes

Continuation and multiple paths, e.g. for

$$f(x; a) = (x^2 - 1)^2 - a$$

[3]This can be made more precise by assuming that the LTE for a step of size h starting at t is $\tau(h; t) \approx C(t)h^{p+1}$ where $C(t)$ is a smooth function of t.