Solving differential equations is one of the most important tasks in applied math. Many practical situations, e.g. the behavior of water waves, are modeled by systems of differential equations and finding solutions to these differential equations is important in understanding these situations. The most common problem is the *initial-value problem* which specifies a system of differential equations

$$\frac{dy_i}{dt} = f_i(t, y_1, \ldots, y_n)$$

for $a \leq t \leq b$ along with initial conditions

$$y_i(a) = \alpha_i$$

for each $1 \leq i \leq n$. There are a few general approaches we can use for this kind of problem.

1. We can try to solve the equation exactly. There are few situations in which this approach is appropriate given the complexity of most situations. (Obtain exact solutions to exact equations.)

2. Modify/relax the equation in order to be able to solve it exactly. This often requires unjustified assumptions about the situation, limiting the usefulness of the solution obtained this way. (Obtain exact solutions to approximate equations.)

3. Approximate solutions to the differential equation using numerical techniques. This typically gives the most accurate results and the most realistic error information. (Obtain approximate solutions to the exact equations.)

Rather than finding a functional description of the solution, we instead obtain approximate values at certain specified (and equally spaced) points and then use an appropriate type of interpolation to obtain other values if needed.

1 Existence, Uniqueness, and Well-posedness

Existence and Uniqueness of solutions is crucial to the success of numerical methods.

Example 1.1. Consider the initial value problem

$$y'(t) = y(t)^2 \qquad y(0) = y_0.$$

We can solve this problem by separation of variables

$$\int_0^t \frac{y'(s)ds}{y(s)^2} = \int_0^t 1ds$$

to obtain the solution

$$-\frac{1}{y(s)}\bigg|_0^t = t \implies y(t) = \frac{1}{\frac{1}{y_0} - t}.$$

In this situation, we will have no solution at $t = \frac{1}{y_0}$ and no hope of solving this equation numerically.

Example 1.2. Consier the initial value problem

$$y'(t) = 2\sqrt{y(t)} \qquad y(0) = y_0 \geq 0.$$

We can solve this problem again by separation of variables

$$\int_0^t \frac{y'(s)ds}{2\sqrt{y(s)}} = \int_0^t 1ds$$

to obtain the solution

$$\sqrt{y(s)}\Big|_0^t = t \implies y(t) = (t + \sqrt{y_0})^2.$$

However, if $y_0 = 0$, then consider the further solutions

$$y_c(t) = \begin{cases} (t-c)^2 & c \leq t \\ 0 & 0 \leq t \leq c \end{cases}.$$

This function is 0 for a while before suddenly starting to increase quadratically at some time $c \geq 0$. From the perspective of watching time increase, we won't know when this function starts to increase, so we can't expect to solve this numerically.

These examples demonstrate two important aspects of differential equations – existence and uniqueness of solutions – that we require in order to be able to develop meaningful numerical methods. Let's investigate what conditions we can impose on initial value problems to guarantee existence and uniqueness of solutions.

1.1 Picard iteration

Let's restate the initial value problem

$$y'(t) = f(t, y) \qquad y(a) = y_0$$

by first integrating both sides

$$\int_a^t y'(s)ds = \int_a^t f(s, y(s))ds.$$

To simplify the left hand side, we recall the fundamental theorem of calculus.

Theorem 1.3 (Fundamental theorem of calculus)**.** *Let $f(x)$ be a differentiable function on the interval $[a, b]$. Then for any $t \in [a, b]$,*

$$f(t) = f(a) + \int_a^t f'(s)ds.$$

Applying this to our situation, we obtain

$$y(t) = y_0 + \int_a^t f(s, y(s))ds$$

This equation is known as the *Picard integral equation.*

Question 1.4. Does this remind you of any other method we've used so far in class?

If we know the function y up to time t, then we can figure out $y(t)$ by solving this integral. We will come back to (a discretized version of) this idea later. For now, observe the similarity between this equation and fixed point iteration. First, if we know the function $y(t)$, then plugging it into the integral equation returns the actual function itself. Otherwise, if we only have a guess $y_0(t)$ for the function itself, we can hope that iterating the equation will yield a good approximation of the desired solution

$$y_{n+1}(t) = y_0 + \int_0^t f(s, y_n(s))ds.$$

To phrase this more generally, let V denote the vector space of differentiable functions in one variable and $F : V \to V$ the transformation given by

$$y \mapsto y_0 + \int_a^t f(s, y(s))ds.$$

Then the Picard integral equation gives a literal fixed point iteration on V.

Question 1.5. Perform iterations using the Picard integral equation for the initial value problem

$$y'(t) = y(t)^2 \qquad y(0) = y_0.$$

Example 1.6. The Picard integral form of this initial value problem is

$$y(t) = y_0 + \int_0^t y(s)^2 ds.$$

Let's guess $y_0(t) = y_0$. What is $y_1(t)$?

$$y_1(t) = y_0 + \int_0^t y_0(s)^2 ds = y_0 + \int_0^t y_0^2 ds = y_0 + y_0^2 t = y_0(1 + y_0 t).$$

$$y_2(t) = y_0 + \int_0^t y_0^2(1 + y_0 s)^2 ds = y_0 + y_0^2(t + y_0 t^2 + \frac{y_0^2 t^3}{3}) = y_0(1 + y_0 t + y_0^2 t^2) + \frac{y_0^4 t^3}{3}.$$

$$y_3(t) = y_0 + \int_0^t y_0^2(1 + y_0 s + y_0^2 s^2 + \frac{y_0^3 s^3}{3})^2 ds = y_0 + y_0^2(t + y_0 t^2 + y_0^2 t^3 + \frac{2y_0^3 t^4}{3} + \frac{y_0^4 t^5}{3} + \frac{y_0^5 t^6}{9} + \frac{y_0^6 t^7}{63})$$
$$= y_0(1 + y_0 t + y_0^2 t^2 + y_0^3 t^3) + \dots.$$

We guess that the fixed point iterations will look like

$$y_n(t) = y_0(1 + y_0 t + \cdots + y_0^n t^n) + \dots$$

which will converge to

$$y(t) = y_0(1 + y_0 t + y_0^2 t^2 + \dots) = \frac{y_0}{1 - y_0 t} = \frac{1}{\frac{1}{y_0} - t}.$$

Question 1.7. What conditions did we need for fixed point iteration to work for solving equations?

Recall that we needed two main conditions for fixed point iteration to work. The first was invariance ($g(x) \in [a, b]$ for $x \in [a, b]$) and the second was contractive ($|g'(x)| < 1$ for $x \in [a, b]$). That was a special case of a more general fixed point theorem. The important criterion to show in the current situation is *contractive*, or that for two functions u, v we have that $F(u), F(v)$ are "closer" than u and v are. One way to achieve this is to satisfy the following inequality for functions $u(t), v(t) \in V$:

$$\sup_{a \le t \le b} |F(u(t)) - F(v(t))| \le L \sup_{a \le t \le b} |u(t) - v(t)|.$$

We can simplify the left hand side somewhat

$$\begin{aligned} |F(u(t)) - F(v(t))| &= \left| y_0 + \int_a^t f(s, u(s))ds - y_0 - \int_a^t f(s, v(s))ds \right| \\ &= \left| \int_a^t f(s, u(s)) - f(s, v(s))ds \right| \\ &\le \int_a^t |f(s, u(s)) - f(s, v(s))|ds. \end{aligned}$$

This naturally leads us to the idea of a *Lipschitz condition* on f.

Definition 1.8. The function $f(t, y)$ satisfies a *Lipschitz condition* in the variable y if there is a constant $L > 0$ such that

$$|f(t, y_1) - f(t, y_2)| \le L|y_1 - y_2|.$$

This condition is much stronger than being continuous, but weaker than being differentiable. According to our discussion above, we have roughly the following theorem.

Theorem 1.9. *The initial value problem*

$$y'(t) = f(t, y) \qquad y(a) = y_0$$

has a unique solution if $f(t, y)$ satisfies a Lipschitz condition in the variable y.

One easy way to guarantee that f satisfies a Lipschitz condition is the following condition, which should be very reminiscent of the situation for fixed point iteration for solving (usual) equations.

Theorem 1.10. *If $f(t, y)$ is defined on a convex set (typically we consider the sets $\{(t, y) \mid a \le t \le b, -\infty \le y \le \infty\}$) has a bounded derivative*

$$|f_y(t, y)| \le L$$

then f satisfies a Lipschitz condition in the variable y with Lipschitz constant L.

There's still one important point to discuss with respect to applying numerical methods to initial value problems.

1.2 Well-posedness

Initial value problems that provide models for real world phenomena typically only give approximations of the underlying phenomenon. This could be due to various factors, chief among which are the fact that the true phenomenon is intractably complicated and the fact that we often need to use floating point approximations of the true values. Therefore, for various reasons, the problem we're actually interested in solving may be only approximately related to the problem that we have. In order to guard against this, we give the following definition.

Definition 1.11. The initial value problem

$$y'(t) = f(t, y) \qquad a \leq t \leq b \qquad y(a) = y_0$$

is *well-posed* if

1. it has a unique solution

2. There are $\varepsilon_0 > 0$ and $k > 0$ such that for any $0 < \varepsilon < \varepsilon_0$, whenever $\delta(t)$ is a continuous function with $|\delta(t)| < \varepsilon$ for all $t \in [a, b]$ and when $|\delta_0 < \varepsilon|$, then the initial value problem
$$z' = f(t, z) + \delta(t) \qquad a \leq t \leq b \qquad z(a) = y_0 + \delta_0$$
has a unique solution $z(t)$ satisfying
$$\sup_{t \in [a,b]} |z(t) - y(t)| < k\varepsilon.$$

The transformed problem in the definition is often called a *perturbed problem* and allows for the possibility of error in the modeling of the physical phenomenon. In general, well-posedness is exactly the condition we want so that numerical methods give meaningful approximations of the desired solution. It turns out that we get well-posedness is for free if we satisfy the Lipschitz condition.

Theorem 1.12. *If $f(t, y)$ is continuous and satisfies a Lipschitz condition in the variable y on the set $\{(t, y) \mid a \leq t \leq b, -\infty \leq y \leq \infty\}$, then the initial value problem*

$$y' = f(t, y) \qquad a \leq t \leq b \qquad y(a) = y_0$$

is well-posed.

For the remainder of this unit, unless otherwise stated, we will work with well-posed initial value problems (the generalization to systems is straightforward). As we saw from the Picard integral equation

$$y(t) = y_0 + \int_a^t f(s, y(s))ds$$

we can use knowledge of $y(s)$ for $a \leq s < t$ to figure out $y(t)$. While this provides a nice theoretical understanding, it doesn't quite give a numerical method for approximating the solution $y(t)$ we are after.

Question 1.13. How would you discretize this idea to get an actual numerical method?

One way to discretize this idea is the following.

1. First, subdivide the interval $[a, b]$ using $N + 1$ *mesh points*
$$t_i = a + ih \qquad h = \frac{b-a}{N}.$$
2. Consider the Picard integral equation on each adjacent pair of mesh points
$$y(t_i) = y(t_{i-1}) + \int_{t_{i-1}}^{t_i} f(s, y(s))ds = y(t_{i-1}) + \int_{t_{i-1}}^{t_i} y'(s)ds = y(t_{i-1}) + y(t_i) - y(t_{i-1}).$$
If we find some way to approximate either of those integrals using some kind of numerical scheme, then we can get an approximation for $y(t_i)$.
3. Using the approximations of $y(t_i)$, we can then interpolate a function passing through all the points we found.

2 Euler's method

We left off with the question of how best to approximate the following integral
$$\int_{t_{i-1}}^{t_i} f(s, y(s))ds = \int_{t_{i-1}}^{t_i} y'(s)ds?$$

One way is to proceed inductively. Since we know the initial value and slope at t_0 (i.e. we have the initial condition $y(t_0) = w_0 = y_0$ and thus we can compute $y'(t_0) = f(t_0, w_0)$), we can integrate the tangent line approximation to $y(t)$ at t_0 in order to approximate the next point via
$$y(t_1) \approx w_1 = w_0 + hy'(t_0) = w_0 + hf(t_0, w_0).$$
More generally, assuming we've already approximated $y(t_i) \approx w_i$, then we can use the formula
$$y(t_{i+1}) \approx w_{i+1} = w_i + hf(t_i, w_i).$$

Question 2.1. How are we relying on the assumption of well-posedness here?

If we do not have well-posedness, then there is no reason that $f(t_i, w_i) \approx f(t_i, y(t_i)) = y'(t_i)$ should give a good approximation. In order to understand the error of Euler's method, we take a slightly different perspective. First, consider the Taylor expansion of $y(t)$ at t_i
$$y(t) = y(t_i) + y'(t_i)(t - t_i) + \frac{y''(\xi_i(t))}{2}(t - t_i)^2$$
evaluated at t_{i+1}
$$y(t_{i+1}) = y(t_i) + y'(t_i)(t_{i+1} - t_i) + \frac{y''(\xi_i)}{2}(t_{i+1} - t_i)^2 = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(\xi_i).$$

We then obtain Euler's method by truncating the error term $\frac{h^2}{2}y''(\xi_i)$. To simplify notation, let $y_i = y(t_i)$. Then to estimate the accumulated error, we investigate the differences

$$\begin{aligned} y_{i+1} - w_{i+1} &= (y_i + hf(t_i, y_i) + \frac{h^2}{2}y''(\xi_i)) - (w_i + hf(t_i, w_i)) \\ &= y_i - w_i + h(f(t_i, y_i) - f(t_i, w_i)) + \frac{h^2}{2}y''(\xi_i). \end{aligned}$$

Taking absolute values, we obtain

$$|y_{i+1} - w_{i+1}| \leq |y_i - w_i| + h|f(t_i, y_i) - f(t_i, w_i)| + \frac{h^2}{2}|y''(\xi_i)|.$$

This equation is not very easy to manipulate. To get something workable, we'll assume that $f(t, y)$ is Lipschitz in the variable y with Lipschitz constant L to handle the second term and the second derivative is bounded $|y''| < M$ to handle the third term. (We're basically making these assumptions anyway – furthermore, we can try to obtain y'' using the chain rule $y'' = f_t(t, y) + f_y(t, y)f(t, y)$.) This gives for each i the bound

$$|y_{i+1} - w_{i+1}| \leq |y_i - w_i| + hL|y_i - w_i| + \frac{h^2 M}{2} = (1 + hL)|y_i - w_i| + \frac{h^2 M}{2}.$$

If we continue backwards:

$$\begin{aligned} |y_{i+1} - w_{i+1}| &\leq (1 + hL)|y_i - w_i| + \frac{h^2 M}{2} \\ &\leq (1 + hL)((1 + hL)|y_{i-1} - w_{i-1}| + \frac{h^2 M}{2}) + \frac{h^2 M}{2} \\ &\cdots \\ &\leq (1 + hL)^{i+1}|y_0 - w_0| + \frac{h^2 M}{2}(1 + (1 + hL) + \cdots + (1 + hL)^i) \\ &\leq \frac{(1 + hL)^{i+1} - 1}{(1 + hL) - 1}\frac{h^2 M}{2} = ((1 + hL)^{i+1} - 1)\frac{hM}{2L} \\ &\leq (e^{(i+1)hL} - 1)\frac{hM}{2L} = (e^{(t_{i+1} - a)L} - 1)\frac{hM}{2L}. \end{aligned}$$

Theorem 2.2. *Suppose f satisfies a Lipschitz condition with constant L and that y has a bounded second derivative $|y''| \leq M$. Then*

$$|y_i - w_i| \leq \frac{hM}{2L}(e^{(t_i - a)L} - 1).$$

If we account for floating point errors ($y_0 \to y_0 + \delta_0$ and $w_{i+1} \to w_i + hf(t_i, w_i) + \delta_{i+1}$)

Theorem 2.3. *Under the same conditions as before, letting $\delta = \max_i |\delta_i|$, then*

$$|y_i - w_i| \leq \frac{1}{L}\left(\frac{hM}{2} + \frac{\delta}{h}\right)\left(e^{(t_i - a)L} - 1\right) + |\delta_0|e^{(t_i - a)L}.$$

Due to the presence of roundoff errors, we can no longer let h go to 0. Instead, the optimal choice occurs at approximately $h = \sqrt{2\delta/M}$.

2.1 Higher order Taylor's method

As we saw before, Euler's method can be interpreted as using a first order Taylor approximation to $y(t)$ at the point t_i to approximate $y(t_{i+1})$. However, the linear error term is troubling. For example, if we want six decimal places of accuracy, we'll need on the order of one million steps to achieve the desired accuracy.

Question 2.4. Improve Euler's method by using a better approximation.

We can improve this by using higher order Taylor approximations

$$y(t) = y(t_i) + y'(t_i)(t - t_i) + \cdots + \frac{y^{(n)}(t_i)}{n!}(t - t_i)^n + \frac{y^{(n+1)}(\xi(t))}{(n+1)!}(t - t_i)^{n+1}$$

for approximating the integral

$$\int_{t_i}^{t_{i+1}} y'(s)ds.$$

(Recall from our analysis of the Picard integral equation that everything essentially comes down to how well we can approximate these integrals.) Evaluating at t_{i+1} yields

$$y_{i+1} = y_i + hy_i' + \cdots + \frac{y_i^{(n)}}{n!}h^n + \frac{h^{n+1}}{(n+1)!}y^{(n+1)}(\xi_i)$$

and plugging in $y' = f(t, y)$ gives

$$y_{i+1} = y_i + hf(t_i, y_i) + \cdots + \frac{h^n}{n!}f^{(n-1)}(t_i, y_i) + \frac{h^{n+1}}{(n+1)!}f^{(n)}(\xi_i, y(\xi_i))$$

and we obtain the Taylor method of order n (Euler is Taylor of order one) by truncating the error term

$$y_{i+1} \approx y_i + hf(t, y_i) + \cdots + \frac{h^n}{n!}f^{(n-1)}(t, y_i).$$

If we want to interpolate the solution, it makes more sense to use Hermite interpolation since we already know (approximations of) the derivatives from the previous computations.

Unfortunately, the error of higher order Taylor methods is much more difficult to analyze because assuming that f is Lipschitz doesn't give us much control over the derivatives $f^{(k)}(t_k, w_k)$, so we won't be able to obtain good estimates for the differences

$$f^{(k)}(t_k, w_k) - f^{(k)}(t_k, y_k).$$

As a result, we need to seek other methods to evaluate the accuracy of our methods. To do this, we'll use the *local truncation error*.

Definition 2.5. Consider the difference method (i.e. approximating a differential equation using a difference equation)

$$w_0 = y_0$$
$$w_{i+1} = w_i + h\varphi(t_i, w_i)$$

for some φ. The *local truncation error* associated to this difference method is

$$\tau_{i+1}(h) = \frac{y_{i+1} - y_i}{h} - \varphi(t_i, y_i).$$

This measures the amount by which the unique solution to the differential equation fails to satisfy the difference equation being used for approximation. This seems at best tangentially related to our goal, since what we really want to know is how well our approximation satisfies the differential equation rather than how well the difference equation is satisfied by the exact solution. However, because we don't usually know the exact solution for equations we want to solve, we cannot hope to be able to know the exact error either.

Question 2.6. What is the local truncation error of Euler's method?

We saw above that the local truncation error (and in fact the error as well) both depend linearly on the step size h. This means that e.g. if we want 6 decimal places of accuracy, we'll need around one million steps, which is not very satisfactory.

Theorem 2.7. *Taylor's method of order n has local truncation error*

$$\tau_{i+1}(h) = \frac{h^n}{(n+1)!} f^{(n)}(\xi_i, y(\xi_i)).$$

3 Error analysis for difference methods

The best case scenario for a difference method would be that the solution the method computes converges to the actual solution (i.e. if $w_i \to y(t_i)$ as the step size goes to 0). We measure this by the following definition.

Definition 3.1. A difference equation method is *convergent* with respect to the differential equation it approximates if

$$\lim_{h\to 0} \max_{1\le i\le N} |w_i - y(t_i)| = 0.$$

Here, $y(t_i)$ denotes the exact value of the solution and w_i is the approximation obtained in the i^{th} step of the difference method.

However, as we saw above, outside of some particularly simple cases, it is very difficult to analyze the error/convergence of many methods. The solution we came up with for the higher order Taylor methods above was to study the *local truncation error* $\tau_i(h) = \frac{y_i - y_{i-1}}{h} - \varphi(t_i, y_i)$ associated to the one step method

$$w_0 = y_0$$

$$w_{i+1} = w_i + h\varphi(t_i, w_i).$$

Although it's not exactly what we want, one criterion we might ask for to determine the error is that the local truncation errors go to zero, or in other words the difference equation we use converges to the actual differential equation as h goes to zero.

Definition 3.2. A one-step difference method with local truncation errors $\tau_i(h)$ is *consistent* with the differential equation it approximates if

$$\lim_{h\to 0} \max_{1\le i\le N} |\tau_i(h)| = 0.$$

Complicating matters further, we haven't even discussed numerical issues arising from roundoff errors. For this, we say that a numerical method is *stable* if small perturbations in the initial conditions produce small changes in the output approximations. This is closely related to well-posedness that we required above. Luckily, the following theorem rescues us in this dire situation.

Theorem 3.3. *Suppose that the initial value problem*

$$y' = f(t, y) \qquad a \leq t \leq b \qquad y(a) = y_0$$

is approximated by a difference method of the form

$$w_0 = y_0$$

$$w_{i+1} = w_i + h\varphi(t_i, w_i, h).$$

If φ is continuous and satisfies a Lipschitz condition in the variable w, then

1. *the method is stable,*

2. *the method is convergent if and only if it is consistent, which is equivalent to*

$$\varphi(t, y, 0) = f(t, y)$$

3. *For each i, the approximation error is bounded in terms of the local truncation error*

$$|w_i - y(t_i)| \leq \frac{\tau_i(h)}{L} e^{L(t_i - a)}.$$

This theorem justifies our methods, because we're already assuming a Lipschitz condition in order to gain existence, uniqueness, and well-posedness. Here, we see further that the Lipschitz condition also guarantees good behavior of the method in terms of convergence as well as the error.

4 Higher order equations and systems of equations

Euler/higher order Taylor (as well as all other methods we will discuss) apply equally well to systems of equations as well as higher order equations. To solve first order systems, we carefully generalize our previous first order methods. To solve higher order equations, we will need to transform them first to first order systems. In general, we can transform higher order systems into first order systems and then apply (generalizations of) any of the methods we've discussed for solving first order ODEs.

4.1 First order systems

So far, we've only looked at first order equations

$$y' = f(t, y) \qquad a \le t \le b \qquad y'(a) = y_0.$$

More generally, we would also like to be able to solve systems of first order equations

$$\begin{aligned} u_1' &= f_1(t, u_1, \dots, u_m) \\ u_2' &= f_2(t, u_1, \dots, u_m) \\ &\vdots \\ u_m' &= f_m(t, u_1, \dots, u_m) \end{aligned}$$

on $t \in [a, b]$ with initial conditions $u_i(a) = \alpha_i$. While the setup has become slightly more complicated, the idea for solving these systems will be exactly the same as for first order equations: we will approximate the integrals

$$u_i(t_{j+1}) = \int_{t_j}^{t_{j+1}} f_i(t, u_1, \dots, u_m) dt$$

using exactly the same ideas that we have been studying so far. We can generalize any of the first order ODE methods that we will see, but so far we have only seen Euler's method, so we will demonstrate how to generalize Euler's method to first order systems.

Example 4.1. Euler's method for a first order system is as follows. Suppose that we have already approximated $u_i(t_j) \approx w_{ij}$. Then at each time step t_j, we use

$$u_i(t_{j+1}) = u_i(t_j) + \int_{t_j}^{t_{j+1}} f(t, u_1(t), \dots, u_m(t)) dt \approx w_{ij} + hf(t_j, w_{1j}, \dots, w_{mj})$$

to form the approximations for the next round $u_i(t_{j+1})$.

As with first order equations, we rely on a *Lipschitz condition* in order to guarantee existence and uniqueness of solutions so that our numerical methods have a chance of succeeding.

Definition 4.2. The function $f(t, y_1, \dots, y_m)$ satisfies a *Lipschitz condition* in the variables $u_1, \dots, u_m$ if there is $L > 0$ such that

$$|f(t, u_1, \dots, u_m) - f(t, z_1, \dots, z_m)| \le L \sum_{j=1}^{m} |u_j - z_j|$$

for all $(t, u_1, \dots, u_m)$ and $(t, z_1, \dots, z_m)$ where $a \le t \le b$.

As before, we can guarantee that a given f satisfies a Lipschitz condition if it is bounded in the following sense.

Theorem 4.3. *Suppose that f is continuously differentiable in each u_i and that*

$$\left| \frac{\partial f(t, u_1, \dots, u_m)}{\partial u_i} \right| \le L$$

for each i. Then f satisfies a Lipschitz condition with Lipschitz constant L.

Theorem 4.4. *Suppose that $f_i(t, u_1, \dots, u_m)$ satisfies a Lipschitz condition for each i. Then the first order system specified by f_i has a unique solution.*

4.2 Higher order equations

Another generalization is higher order equations (i.e. involving higher derivatives) such as

$$y^{(m)} = f(t, y, y^{(1)}, \dots, y^{(m-1)}) \qquad a \le t \le b \qquad y^{(i-1)}(a) = \alpha_i, 1 \le i \le m.$$

The trick to solving these equations is to transform them into first order systems of equations, and then apply what we described above. In order to accomplish this transformation, we define new functions $u_1(t) = y(t), u_2(t) = y^{(1)}(t), \dots, u_m(t) = y^{(m-1)}(t)$ to represent each derivative. Then the higher order equation becomes the following system of equations

$$\begin{aligned} u_1'(t) &= y^{(1)}(t) = u_2(t) & u_1(a) &= y(t) = \alpha_1 \\ u_2'(t) &= y^{(2)}(t) = u_3(t) & u_2(a) &= y^{(1)}(t) = \alpha_2 \\ &\vdots & &\vdots \\ u_{m-1}'(t) &= y^{(m-1)}(t) = u_m(t) & u_{m-1}(a) &= y^{(m-2)}(t) = \alpha_{m-1} \\ u_m'(t) &= y^{(m)}(t) = f(t, u_1, u_2, \dots, u_m) & u_m(a) &= y^{(m-1)}(a) = \alpha_m. \end{aligned}$$

(This may feel like something of a cheap trick, but it's a very common strategy in math: we take a problem we don't quite know how to solve, and manipulate it until it turns into a problem that we do know how to solve.)

Example 4.5. Transform the following second order equation into a system of first order equations

$$y'' - 2y' + 2y = e^{2t}\sin(t).$$

We set $u_1 = y$ and $u_2 = y'$. Then the above system is equivalent to

$$u_1' = u_2$$

$$u_2' = e^{2t}\sin(t) + 2u_2 - 2u_1.$$

The same method applies to any system of higher order equations. We can transform such a system into a system of first order equations by applying the above procedure to each equation involving derivatives higher than first order. Then we can solve it using any of the methods we have discussed or that we will discuss.

Question 4.6. Another approach might be to approximate $y \approx P$ using polynomial approximation and then to approximate the derivatives of y using the derivatives of P. Discuss the benefits/drawbacks of this approach?

5 Runge-Kutta methods

We saw with the higher order Taylor methods that we can better approximate the integral and obtain good local truncation error at the cost of having to evaluate derivatives of $f(t, y)$. In practice, these extra steps complicate matters sufficiently that Taylor methods of higher order are rarely used. The goal of Runge-Kutta methods is to obtain the better local truncation errors of Taylor methods without needing to compute the higher order derivatives of $f(t, y)$. To do this, we'll devise better methods for approximating the integral

$$\int_{t_i}^{t_{i+1}} y'(s)ds$$

and determine how to obtain good performance in terms of local truncation errors. Recall from numerical integration that we approximate the value of an integral by sampling function values on the integration interval and using a weighted sum to approximate the value of the integral. We'd like to do that here, but unfortunately we don't know $y'(s)$ explicitly (unless $f(t, y)$ does not depend on y, in which case we can really just do usual numerical integration).

5.1 Midpoint Runge-Kutta

For example, consider the midpoint method for numerical integration. In this case, we would use the approximation

$$\int_{t_i}^{t_{i+1}} f(s, y(s))ds \approx w_i + hf(t_i + \frac{h}{2}, y(t_i + \frac{h}{2})).$$

Unfortunately, we don't have access to $y(t_i + \frac{h}{2})$.

Question 5.1. What can we use to approximate $y(t_i + \frac{h}{2})$?

In absence of better information about $y(t)$, a natural thing to do is make a small Euler step from $y(t_i)$, which we have already approximated with w_i.

$$y(t_i + \frac{h}{2}) \approx w_i + \frac{h}{2}f(t_i, w_i).$$

Then we can plug this into our formula above,

$$hf(t_i + \frac{h}{2}, y(t_i + \frac{h}{2})) \approx hf(t_i + \frac{h}{2}, w_i + \frac{h}{2}f(t_i, w_i)),$$

yielding the difference method

$$w_0 = y_0$$
$$w_{i+1} = w_i + hf(t_i + \frac{h}{2}, w_i + \frac{h}{2}f(t_i, w_i)).$$

To see whether this extra effort got us anything good, let's look at the local truncation errors

$$\tau_{i+1}(h) = \frac{y_{i+1} - y_i}{h} - f(t_i + \frac{h}{2}, y_i + \frac{h}{2}f(t_i, y_i)).$$

To find the order of the dependence of τ_{i+1} on h, we can look at a Taylor expansion near $h = 0$

$$\tau_{i+1}(h) = \tau(0) + \tau'(0)h + \frac{\tau''(0)}{2!}h^2 + \dots$$

and determine which coefficients $\tau_{i+1}(0), \tau'_{i+1}(0), \tau''_{i+1}(0)$, etc. are equal to 0. Recall that $y_{i+1} = y(t_{i+1})$ and $y_i = y(t_i)$. Here we can view $y(t_{i+1})$ as a function of h via $y(t_i + h)$ and Taylor expand it near $h = 0$.

$$y(t_i + h) = y(t_i) + y'(t_i)h + \frac{y''(t_i)}{2}h^2 + \cdots = y_i + y'_i h + \frac{y''_i}{2}h^2 + \ldots$$

Similarly, we may view $f(t_i + \frac{h}{2}, y_i + \frac{h}{2}f(t_i, y_i))$ as a function of h and Taylor expand this around $h = 0$ as well.

$$f(t_i + \frac{h}{2}, y_i + \frac{h}{2}f(t_i, y_i)) = f(t_i, y_i) + (f_t(t_i, y_i) + f_y(t_i, y_i)f(t_i, y_i))\frac{h}{2} + \ldots$$

This gives us the expression

$$\begin{aligned}
\tau_{i+1}(h) &= \frac{y_i + y'_i h + \frac{y''_i h^2}{2} + \cdots - y_i}{h} - f(t_i, y_i) - (f_t(t_i, y_i) + f_y(t_i, y_i)f(t_i, y_i))\frac{h}{2} - \ldots \\
&= y'_i + \frac{y''_i}{2}h + \cdots - f(t_i, y_i) - (f_t(t_i, y_i) + f_y(t_i, y_i)f(t_i, y_i))\frac{h}{2} - \ldots \\
&= y'_i - f(t_i, y_i) + (y''_i - (f_t(t_i, y_i) + f_y(t_i, y_i)f(t_i, y_i)))\frac{h}{2} + O(h^2).
\end{aligned}$$

where $O(h^2)$ means that the remaining terms are of order h^2 or higher. By definition

$$y'(t) = f(t, y(t)) \implies y'_i = y'(t_i) = f(t_i, y(t_i))$$

so the first term above is equal to zero. Moreover,

$$y''_i = y''(t_i) = \frac{d}{dt}f(t, y(t))|_{t_i} = f_t(t_i, y_i) + f_y(t_i, y_i)y'(t_i) = f_t(t_i, y_i) + f_y(t_i, y_i)f(t_i, y_i)$$

so the second term above is equal to zero as well.

Question 5.2. Is the coefficient of h^2 above equal to zero?

Since the coefficient of h^2 above is nonzero, we write $\tau_{i+1}(h) = O(h^2)$ to denote that $\tau_{i+1}(h) = a_2h^2 + a_3h^3 + \ldots$, where $a_2, a_3, \ldots$ are coefficients not involving h. This is a great improvement – in fact, it had better be a good improvement since it cost us one extra evaluation of $f(t, y)$. We could have instead performed Euler's method with half the step size otherwise.

In any case, we were able to achieve second order performance without needing to evaluate further derivatives of $f(t, y)$, meaning that we obtained the performance of the higher order Taylor methods for a much cheaper cost.

5.2 General Runge-Kutta

We achieved success using the midpoint Runge-Kutta rule by nesting one evaluation of $f(t, y)$ inside another. More generally, we can seek to gain performance by exploiting this idea repeatedly. The general setup for a Runge-Kutta method is as follows.

Definition 5.3. We define an *s-stage Runge-Kutta method* for solving the initial value problem

$$y'(t) = f(t, y(t)) \qquad a \le t \le b \qquad y(a) = y_0$$

as follows. First, we begin with $w_0 = y_0$ as always. Otherwise, for each i, we compute

$$k_j = f(t_i + c_j h, y_i + h \sum_{l=1}^{s} a_{il} k_l) \qquad 1 \le j \le s$$

and then finally we compute

$$y_{i+1} = y_i + h \sum_{l=1}^{s} b_l k_l.$$

Typically, we represent this schematically using a *Butcher array*

$$\begin{array}{c|ccccc} c_1 & a_{11} & a_{12} & \dots & a_{1,s-1} & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2,a-1} & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_{s-1} & b_s \end{array}.$$

If $c_s = 1$ and $b_i = a_{si}$, then we say that the method has the *first same as last* (FSAL) property. While this doesn't have any effect on the accuracy of the method, it increases efficiency by making the last step k_s of one round equal to the first step of the next, saving one function evaluation.

Something's a bit strange here. Suppose $s = 1$. Then above, the formula says that we need to compute

$$k_1 = f(t_i + c_1 h, y_i + h(a_{11} k_1)).$$

In other words, k_1 depends on itself!

Question 5.4. How can we plug k_1 into the formula above if we don't know what it is?

The issue here is that the equation above only defines k_1 implicitly – in terms of itself. In order to handle this issue, we need to apply a (potentially expensive) rootfinding operation such as Newton's method to find a zero of

$$k_1 - f(t_i + c_1 h, y_i + h(a_{11} k_1)).$$

Note however that if $a_{ij} = 0$ whenever $i \le j$, then we can compute each stage explicitly by directly plugging in the intermediate values as we compute them.

Definition 5.5. We say that a Butcher array of the form

$$\begin{array}{c|ccccc} c_1 & 0 & 0 & \dots & 0 & 0 \\ c_2 & a_{21} & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & 0 \\ \hline & b_1 & b_2 & \dots & b_{s-1} & b_s \end{array}.$$

defines an *explicit* Runge-Kutta method. Notationally, this means that $a_{ij} = 0$ whenever $i \leq j$ (i.e. the matrix of a_{ij}'s is strictly lower triangular). In general, we say that a Butcher array defines an *implicit* Runge-Kutta method otherwise.

While the explicit Runge-Kutta methods are computationally simpler, because we do not need to use rootfinding methods to solve the implicit equations, we will see an important class of initial value problems for which implicit Runge-Kutta methods have much better performance.

Example 5.6. The Butcher array for the midpoint method is given by

$$\begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2} & \\ \hline & 0 & 1 \end{array}$$

Turning back to the local truncation error for general Runge-Kutta methods, we find

$$\tau_{i+1}(h) = y_i' + \frac{y_i''h}{2} + \frac{y_i'''h^2}{3!} + \cdots - \sum_l b_l k_l(0) - h\sum_l b_l k_l'(0) - h^2 \sum_l b_l k_l''(0) - \ldots$$

where $k_l(h) = f(t_i + c_l h, y_i + h\sum_{i=1}^s a_{il}k_l(h))$. Thus, in order for a Runge-Kutta method to yield an order p local truncation error, we require

$$\frac{y_i^{(j)}}{j!} = \sum_l b_l k_l^{(j-1)}(0) \qquad 1 \leq j \leq p.$$

While the first equation when $j = 1$ is relatively straightforward to interpret

$$y_i' = \sum_l b_l k_l(0) = \sum_l b_l f(t_i, y_i) = y_i' \sum_l b_l \implies \sum_l b_l = 1,$$

the rest are not so straightforward to solve (plugging in the formulas for the k_l yield nonlinear systems of equations in the coefficients a, b, c), and provide the main source of difficulty in applying higher order Runge-Kutta methods in general. We conclude for now with an example.

Example 5.7. One of the most popular Runge-Kutta methods is the fourth order (explicit) Runge-Kutta method specified by the following Butcher array

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array}.$$

The fully written formulation is as follows:

$$w_0 = y_0$$

$$\begin{aligned}k_1 &= f(t_i, w_i)\\ k_2 &= f(t_i + \frac{h}{2}, w_i + \frac{h}{2}k_1)\\ k_3 &= f(t_i + \frac{h}{2}, w_i + \frac{h}{2}k_2)\\ k_4 &= f(t_i + h, w_i + hk_3)\\ w_{i+1} &= w_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4).\end{aligned}$$

k_1 describes an Euler step from t_i to t_{i+1}. k_2 describes an Euler step from t_i to t_{i+1} using the iniital Euler step. k_3 describes an Euler step from t_i to t_{i+1} using the k_2 midpoint step. Finally, k_4 describes an Euler step from t_i to t_{i+1} using the k_3 midpoint step.

5.3 Iterated Deferred Correction

We saw that Runge-Kutta methods for solving the IVP

$$y'(t) = f(t, y(t)) \qquad t \in [a, b] \qquad y(a) = y_0.$$

can yield very good results. However, it turns out to be rather difficult to come up with good higher order Runge-Kutta methods. For example, ode45 in MatLab (also sometimes called DOPRI5) was discovered by Dormand and Prince in the late 1970s and was essentially the state of the art for 30 years. More recently, ode78 and ode89 in MatLab were discovered by Verner in the late 2000s. These are all adaptive methods based on Runge-Kutte methods that work very well in practice for solving a wide variety of IVPs. We will now explore another way to come up with high order Runge-Kutta methods using an iterative approach.

Let $y(t)$ be the exact solution to the IVP above, and suppose that we obtain approximations $y(t_i) \approx w_i$. Then we can interpolate the (t_i, w_i) that we found to produce a continuous approximation $y(t) \approx w_0(t)$ to the exact solution. We will define $e_1(t) = y(t) - w_0(t)$ to be the error of this approximation and now show that $e_1(t)$ satisfies the initial value problem

$$e_1'(t) = f(t, w_0(t) + e_1(t)) - w_0'(t) \qquad t \in [a, b] \qquad e_1(a) = 0.$$

Since $w_0 = y_0$ is the initial value of the original IVP, then $e_1(a) = y(a) - w_0(a) = y_0 - w_0 = 0$ is the initial value for the error IVP. To see that e_1' satisfies the differential equation, we simply plug in its definition:

$$e_1'(t) = y'(t) - w_0'(t) = f(t, y(t)) - w_0'(t)$$

It's tempting to leave the equation as is, but remember that we don't know $y(t)$, so we would not be able to evaluate this IVP. Thus we need to substitute $y(t) = e_1(t) + w_0(t)$ back in:

$$= f(t, e_1(t) + w_0(t)) - w_0'(t).$$

Now we produce approximations $e_1(t_i) \approx u_i$ using some numerical method and then interpolate the (t_i, u_i) to produce a continuous approximation $e_1(t) \approx u_1(t)$. Then we hope that

$$y(t) \approx w_0(t) + u_1(t) = w_1(t)$$

is a better approximation than just $w_0(t)$. This process is known as *deferred correction* because we first construct the solution and then try to correct it all at once. We can now iterate this procedure as many times as we like. Next consider $e_2(t) = y(t) - w_1(t)$.

Using exactly the same ideas as above, we find the next IVP

$$e_2'(t) = f(t, e_2(t) + w_1(t)) - w_1'(t) \qquad t \in [a, b] \qquad e_2(a) = 0$$

which is solved by e_2. Once again, we approximate $e_2(t) \approx u_2(t)$ using some numerical method and then interpolation and then hope that the iterated correction

$$y(t) \approx w_1(t) + u_2(t) = w_2(t)$$

is even better than $w_1(t)$.

Unfortunately, this strategy as stated is typically unusable in practice. If we want to interpolate the same number of points, then the interpolating polynomial will need to have degree equal to the number of timesteps. In most problems, we will use a moderate number of timesteps to obtain the approximations $w_i \approx y(t_i)$, so we will need to work with very high degree polynomials. For example, it is not unreasonable to take a million timesteps for a slightly complicated system. However, degree one million polynomials do not perform well in practice due to roundoff (the number of arithmetic operations simply to evaluate such a large degree polynomial will immense), computational time and memory (storing that many coefficients is very memory inefficient), and the Runge phenomenon (interpolations become less and less accurate near the endpoints using equally spaced interpolation points). Due to these issues, it is not reasonable in practice to use the procedure as described above.

Thus, it usually makes more sense to apply the above procedure to smaller subintervals at a time. To implement this idea, we will choose some integer p and then in the first iteration of deferred correction, we approximate the error first on the interval $[t_0, t_p]$ using the IVP

$$e_1'(t) = f(t, w_0(t) + e_1(t)) - w_0'(t) \qquad t \in [t_0, t_p] \qquad e_1(t_0) = 0,$$

where $w_0(t)$ interpolates the points (t_i, w_i) for $0 \le i \le p$. We will use the approximation $e_1(t) \approx u_1(t)$ that we find to correct the original approximation to

$$y(t) \approx w_0(t) + u_1(t) = w_1(t)$$

on the interval $[t_0, t_p]$. After this, we then proceed to the next subinterval $[t_p, t_{2p}]$. It may be tempting to reuse the approximation we found at the last step $e(t_p) \approx u_p$, but this idea doesn't quite work out for various reasons. First, the IVP that we want to solve is changing, so the previous approximation may not be relevant anymore (and may even lead to increased error). Secondly, we've already corrected the value of $y(t_p)$ in the previous step, so we do not want to mess with it again. Thus, we will consider the IVP

$$e_1'(t) = f(t, w_0(t) + e_1(t)) - w_0'(t) \qquad t \in [t_p, t_{2p}] \qquad e_1(t_p) = 0,$$

where $w_0(t)$ now interpolates the points (t_i, w_i) for $p \le i \le 2p$. Again, we approximate a solution $e_1(t) \approx u_1(t)$ and use it to correct the original approximation on the interval $[t_p, t_{2p}]$. We continue in this way until we reach the end of the timesteps.

Unfortunately, if we're not careful, then we might end up wasting a lot of work. Suppose that we apply some one-step numerical method to the IVP

$$y'(t) = f(t, y(t)) \qquad t \in [a, b] \qquad y(a) = y_0$$

to produce an approximation $y(t_i) \approx w_i$ (i.e. we have $w_0 = y_0$ and $w_{i+1} = w_i + h\varphi(t_i, w_i)$ for some function φ). For this problem, we'll choose $p = 1$. Since $p = 1$, we will use the degree 1 interpolating polynomial on the intervals $t_i \to t_{i+1}$ for each i. Thus the interpolation points are (t_i, w_i) and (t_{i+1}, w_{i+1}), so we will use the polynomial

$$w_0(t) = \frac{t - t_{i+1}}{t_i - t_{i+1}} w_i + \frac{t - t_i}{t_{i+1} - t_i} w_{i+1} = -(t - t_{i+1})\frac{w_i}{h} + (t - t_i)\frac{w_{i+1}}{h}.$$

Its derivative is

$$w_0'(t) = -\frac{w_i}{h} + \frac{w_{i+1}}{h} = \frac{w_{i+1} - w}{h} = \varphi(t_i, w_i).$$

Thus the IVP we want to solve is

$$e_1'(t) = f(t, e_1(t) + w_0(t)) - w_0'(t) \qquad t \in [t_i, t_{i+1}] \qquad e_1(t_i) = 0.$$

Taking an Euler step yields

$$e_1(t_{i+1}) \approx 0 + h(f(t_i, e_1(t_i) + w_0(t_i)) - w_0'(t))$$

Plugging in what we found for u_1 above, we obtain

$$\begin{aligned} &= hf(t_i, 0 - (t_i - t_{i+1})\frac{w_i}{h}) - h\varphi(t_i, w_i) \\ &= hf(t_i, w_i) - h\varphi(t_i, w_i) = u_1(t_{i+1}). \end{aligned}$$

Then correcting our original approximation, we find

$$\begin{aligned} w_1(t_{i+1}) &= w_0(t_{i+1}) + u_1(t_{i+1}) \\ &= w_0(t_i) + h\varphi(t_i, w_i) + hf(t_i, w_i) - h\varphi(t_i, w_i) \\ &= w_0(t_i) + hf(t_i, w_i) \end{aligned}$$

which is exactly Euler's method.

This gives us a dose of pessimism: if we started off with an accurate approximation, we might end up with a less accurate one if we're not careful! However, this is a reasonable outcome: if we have a very accurate approximation and then we use an inaccurate approximation of the error, we're doing more harm than good by using it for correction. Here is the "right" way to apply iterated deferred correction.

1. Approximate a solution to the initial IVP using any numerical method you like.

2. If your approximation has local truncation error of order k (i.e. the local truncation error is bounded above by h^k times some constant), then choose $p > k$.

3. Each iteration of iterated deferred correction with your choice of $p > k$ will increase the order of the local truncation errors by 1 each time, up to order p. (This means that after $0 < i < p - k$ iterations, the local truncation errors will have order $k + i$.)

It turns out that this process is equivalent to using fixed point iteration to solve an implicit Runge-Kutta method, so iterated deferred correction provides another way to view Runge-Kutta methods.

6 Multistep methods

So far, we've seen one-step methods where the approximation w_i to $y_i = y(t_i)$ depends only on w_{i-1} (e.g. Euler+Taylor) and possibly other intermediate terms between y_{i-1} and y_i computed based on w_{i-1} (e.g. Runge-Kutta). In contrast, multi-step methods also use further previous information gained in previous steps: for example, in order to compute w_i, we may also try to use $w_{i-1}, w_{i-2}, \ldots, w_{i-k}$ as well by finding an interpolating polynomial for the points (t_j, w_j) for $i-k+1 \leq j \leq i$ and then integrating the interpolating polynomial $P_{k-1}(x)$ from t_i to t_{i+1} in order to approximate the Picard integral:

$$w_{i+1} = w_i + \int_{t_i}^{t_{i+1}} f(s, y(s))ds \approx w_i + \int_{t_i}^{t_{i+1}} P_{k-1}(x)dx.$$

Alternatively, we may decide to start from w_{i-1} and then integrate over t_{i-1} to t_{i+1}:

$$w_{i+1} = w_{i-1} + \int_{t_{i-1}}^{t_{i+1}} f(s, y(s))ds \approx w_{i-1} + \int_{t_{i-1}}^{t_{i+1}} P_{k-1}(x)dx.$$

In general, we could even do something more complicated such as taking an average of various approximations in order to "average out the errors" of the different approximations. This leads to the general definition for a multistep method.

Question 6.1. Are there any issues with implementing the ideas above?

The most pressing issue is that in order to come up with an interpolating polynomial to $f(t, y(t))$ of degree $k-1$ to integrate for our approximation, we need to have k points available. In initial value problems, however, we're typically only given a single initial value.

Question 6.2. How can we get enough points to start the interpolation?

Sometimes, we may be lucky enough to have the terms directly available. Generally, we will only be given the single initial value. Some common options to start off with are to generate a few starting values using Euler steps, higher order Taylor methods, or better yet Runge-Kutta methods. After handling this issue, we're faced with a much more subtle issue having to do with the interpolation polynomials.

Question 6.3. How well does the interpolation polynomial behave outside of the interpolation interval?

Unfortunately, we have no guarantees about how the interpolation polynomial behaves outside of the interpolation interval, which makes integrating from t_i to t_{i+1} potentially a bad idea. To make matters worse, we've seen in examples that in fact the behavior of the polynomials is essentially guaranteed to be quite bad near the ends of the interpolation interval and much worse outside of the interpolation interval unless we take a tremendous number of points. However, we also saw that computing the interpolation polynomials themselves can be quite time consuming as well if we're not careful.

Question 6.4. How can we modify the interpolation polynomial in order to minimize the impact of integrating outside of the interpolation interval?

Just like we had with Runge-Kutta methods, we also have implicit multistep methods. To use these, we can simply say that we'll let P_{k-1} interpolate from t_{i-k+1} to t_{i+1} instead of just to t_i. As before, this has the downside that we'll no longer be able to compute the polynomials directly and then use them to approximate the integral. Instead, we'll need to use some kind of root-finding (i.e. bisection, fixed point, Newton) to determine the new value w_{i+1} explicitly.

Definition 6.5. An m-step multistep method for the initial value problem

$$y' = f(t, y) \qquad a \le t \le b \qquad y(t_0) = y_0$$

is given by the update rule

$$w_{i+1} = a_i w_i + \cdots + a_{i-m+1} w_{i-m+1} + h(b_{i+1} f(t_{i+1}, w_{i+1}) + \cdots + b_{i-m+1} f(t_{i-m+1}, w_{i-m+1})).$$

The initial values can either be specified or can be determined using any of the methods described above. If $b_{i+1} = 0$, then the method is said to be *explicit* and these methods are often referred to as Adams-Bashforth methods. Otherwise, if $b_{i+1} \neq 0$, then the method is said to be *implicit* and these methods are often referred to as Adams-Moulton methods.

As we have done previously, we now turn to understanding the local truncation errors of multistep methods in order to study the errors of these methods. For the remainder of this section, we will work with $a_i = 1$ and $a_{i-1} = \cdots = a_{i-m+1} = 0$. These are exactly the methods we introduced at the beginning of this section: we approximate the integral $\int_{t_i}^{t_{i+1}} f(s, y(s))ds$ by first (approximately) interpolating $f(s, y(s))$ using the values we found previously and then integrating our approximation. Fortunately, due to this construction, the local truncation errors are easier to compute in this case because we can leverage results from polynomial interpolation. For the multistep methods we consider, the local truncation error is given by

$$\begin{aligned} h\tau_{i+1}(h) &= y_{i+1} - y_i - hb_{i+1} f(t_{i+1}, y_{i+1}) - \cdots - b_{i-m+1} f(t_{i-m+1}, y_{i-m+1}). \\ &= \int_{t_i}^{t_{i+1}} y'(s)ds - \int_{t_i}^{t_{i+1}} P(s)ds = \int_{t_i}^{t_{i+1}} y'(s) - P(s)ds \end{aligned}$$

where $P(s)$ interpolates $y'(s) = f(s, y(s))$ at $t_{i-m+1}, \ldots, t_i$ (and t_{i+1} if we are using an implicit method). Thus for an explicit m-step method, we have the interpolation error

$$\begin{aligned} &= \int_{t_i}^{t_{i+1}} \frac{y^{(m+1)}(\xi(s))}{m!}(s - t_{i-m+1}) \ldots (s - t_i) \approx h^{m+1} y^{(m+1)}(\xi(s)) \\ &\implies \tau_{i+1}(h) \approx h^m y^{(m+1)}(\xi(s)) \end{aligned}$$

while for an implicit m-step method, we would instead have

$$\begin{aligned} &= \int_{t_i}^{t_{i+1}} \frac{y^{(m+2)}(\xi(s))}{(m+1)!}(s - t_{i-m+1}) \ldots (s - t_i)(s - t_{i+1})ds \approx h^{m+2} y^{(m+2)}(\xi(s)) \\ &\implies \tau_{i+1}(h) \approx h^{m+1} y^{(m+2)}(\xi(s)). \end{aligned}$$

As we can see from the local truncation errors, implicit multistep (Adams-Moulton) methods outperform the equivalent explicit multistep (Adams-Bashforth) methods. However, the tradeoff is that the implicit methods require a computationally expensive root-finding step and as a result these implicit methods are rarely used directly in practice. Instead, they are used in pairs with explicit methods as part of a *predictor-corrector scheme.* These are implemented as follows:

1. Use a one-step method (such as an explicit Runge-Kutta method or Euler's method) to m starting values $y_0, y_1, \ldots, y_{m-1}$.
2. Use an explicit m-step (Adams-Bashforth) multistep method to approximate $w_{mp} \approx y_m$. (This is the *predictor* step.)
3. Use an implicit $m-1$-step (Adams-Moulton) multistep method to approximate $w_m \approx y_m$ using w_{mp} to remove the implicit dependence of w_m on itself. (This is the *corrector* step.)
4. Repeat steps 2 and 3 for as long as desired.

The benefit of predictor-corrector methods is that we are often able to use the output of a relatively crude, but straightforward to compute, predictor method as the input to a more accurate, but difficult to compute, corrector method. This alleviates the computational issues of the correcter method while still benefiting from its increased accuracy. In particular, the local truncation error of a predictor-corrector method is approximately determined by the local truncation errors of the predictor and corrector methods.

$$\tau_i^{pc}(h) \approx \tau_i^c(h) + h\tau_i^p(h)$$

where $\tau^{pc}, \tau^c, \tau^p$ are the local truncation errors of the predictor-corrector, corrector, and predictor methods, respectively.

Question 6.6. Explain why this result makes sense intuitively.

Intuitively, since the predictor-corrector method uses the corrector method as its final step, the result cannot be more accurate than the corrector method itself is (since we are not getting the exact solution to the corrector method). Furthermore, since the output of the predictor method is multiplied by coefficient h in the predictor-corrector method, we expect that its contribution is h times its local truncation error, which is roughly what we see. Unfortunately, we can no longer bound the error directly by the local truncation error. However, we can obtain the following bound for some constants k_1, k_2:

$$|w_i - y_i| \leq \left(\max_j |w_j - y_j| + k_1 \max_j \sigma_j(h)\right) e^{k_2(t_i - a)}$$

Furthermore, the error analysis becomes a bit more complicated for the multistep methods than it was for the one-step methods. To begin with, we will already assume that our function $F(t_i, h, w_{i+1}, \ldots, w_{i+1-m})$ satisfies a Lipschitz condition. The notion of *convergence* is the same:

Definition 6.7. A multistep method is **convergent** if the solution to the difference equation converges to the solution of the differential equation as the step size approaches 0. In mathematical notation,

$$\lim_{h\to 0} \max_{0\le i\le N} |w_i - y_i| = 0.$$

For consistency, recall that we previously only required that the local truncation errors go to 0. However, there is an extra requirement for multistep methods that whatever method we use to "get started" by computing enough values for interpolation must also be consistent.

Definition 6.8. A multistep method is **consistent** if the local truncation errors go to zero and if the starting method is also consistent. In mathematical notation,

$$\lim_{h\to 0} |\tau_i(h)| = 0 \qquad i = m, m+1, \dots, N$$

$$\lim_{h\to 0} |\alpha_i - y_i| = 0 \qquad i = 1, 2, \dots, m-1$$

Unfortunately, the Lipschitz condition is no longer sufficient to guarantee good properties of our numerical methods. In order to obtain any results at all, we must further investigate the multistep method. Towards that goal, we define the *characteristic polynomial*

$$\lambda^m - a_i\lambda^{m-1} - a_{i-1}\lambda^{m-2} - \cdots - a_{i-m+1}.$$

Using this polynomial, we make the following definitions.

Definition 6.9. If all roots of the characteristic polynomial of a multistep method are bounded by 1 in absolute value, then the method is said to satisfy the *root condition.* In this case, if $\lambda = 1$ is the only root with magnitude 1, then the method is said to be *strongly stable.* If there are multiple roots with magnitude 1, then the method is said to be *weakly stable.* Finally, if a method does not sastisfy the root condition at all, then it is called *unstable.*

Strongly stable methods have the best behavior with respect to roundoff errors. The following exercise is the reason that we use present the Adams methods as our standard multistep methods.

Exercise 6.10. Show that all the Adams methods (explicit and implicit) are strongly stable.

Theorem 6.11. *A multistep method is stable if and only if it satisfies the root condition. Furthermore, if the method is consistent, then it is stable if and only if it is convergent.*

As we can see, there is a strong relationship between convergence, consistency, and stability. However, the situation is no longer good enough that the Lipschitz condition is sufficient to guarantee everything we want.

7 Adaptive methods

In numerical integration, one technique we saw for decreasing error was to use more and more subintervals for composite integration. This led to some natural questions:

1. Do we face roundoff errors due to having the step size be too small?

2. How can we choose the best step size to balance accuracy and time?

We saw that composite integration does not face too many issues due to roundoff errors (as long as you choose a reasonable step size). Furthermore, we saw that adaptive methods give a good way to balance accuracy and time by estimating the error and increasing the accuracy when the error is too large and decreasing the accuracy when the error is very small. The main ingredient there was the ability to obtain an estimate on the error by comparing the results of two different integration schemes (e.g. we investigated Simpson's rule vs composite Simpson with two subintervals in class).

Adaptive methods also make sense for solving differential equations, and the key ingredient will be some way to obtain an error estimate at each timestep. In general, we will never be able to understand the exact error of a particular method. However, we will be able to approximate it in many situations. Predictor-corrector methods give a particularly convenient method of approximating the error. Since the predictor and corrector steps give separate estimates of the desired true value (with the corrector step presumably being more accurate), we can use the difference between the predictor and corrector approximations in order to approximate the actual local truncation error, just as we did with adaptive integration. If the approximated local truncation errors exceed a desired tolerance, then we must decrease the step size (which is computationally quite intensive: we need to recompute all the equally spaced points for multistep!) and try again. Otherwise, if the local truncation error is very small, then we may increase the step size to save time in future computations.

More generally, adaptive methods make sense any time you have two approximations of a given quantity where one estimate is (sufficiently) more accurate than the other. The intuition is that the difference between the two estimates will be a good approximation for the actual error which we can then use to make decisions about whether to increase or decrease the step size. (For example, if the inaccurate estimate has an actual error of 1000 while the more accurate estimate has an error of around 10, then the difference $990 \approx 1000$ will be a very good estimate of the actual error, which we have no idea about.) Very roughly speaking, one can expect that

$$\tau_{i+1}(h) \approx \frac{|w_{i+1} - w_{i+1,p}|}{h},$$

although the particular constant will be quite important, so a more careful analysis is needed in each case. Adaptive methods for solving differential equations are incredibly efficient, and are the standard in most numerical ode packages. One of the standard adaptive methods is ode45, which is based on two Runge-Kutta methods, one of order four and one of order five. The Butcher array for this is:

$$\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & \frac{-56}{15} & \frac{32}{9} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\
1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\
\hline
y_1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\
\hline
\widehat{y_1} & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{92097}{339200} & \frac{187}{2100} & \frac{1}{40}
\end{array}$$

This Butcher array describes a seven-step pair of explicit Runge-Kutta methods. The row with y_1 is a fifth order method while the row with $\widehat{y_1}$ is a fourth order method. At each timestep, we compute both y_1 and $\widehat{y_1}$. Then we use their difference as an approximation for the error of the fifth order method. If the error is too large, we decrease the step size and try again while if the error is sufficiently small, then we may increase the step size and continue to the next step. This method is ridiculously effective in practice, as you will see on the homework.

8 Stiff equations

While we have now covered a variety of numerical methods for solving differential equations, we have not commented much on the particular equations that we aim to solve, other than requiring that the functions $f(t, y_1, \ldots, y_m)$ defining our equations satisfy a Lipschitz condition. It turns out that there is a large class of differential equations which require special care when applying numerical methods. These equations are known as *stiff equations* and commonly arise in many applications. The defining characteristic of stiff methods is that the magnitude of the derivative increases while the magnitude of the solution does not. This leads to the consequence that unless we are very careful in our choice of step size, the error of our method can blow up very quickly. (Recall that all of our methods have errors involving both the step size as well as the derivative, so if the derivative is very large, then we need a much smaller step size to compensate.) In particular, this will be troublesome for adaptive methods because the step size may need to be smaller than expected based on the local truncation errors.

Example 8.1. Consider the initial value problem

$$y' = -30y \qquad t \geq 0 \qquad y(0) = 1.$$

We can solve this exactly by separation of variables to get $y(t) = e^{-30t}$. Now let's see what happens when we apply Euler's method with a step size of 0.1. (It's not too difficult here since $f(t, y) = -30y$ does not depend on t.)

$$\begin{aligned}
w_0 &= y(0) = 1 \\
w_1 &= 1 + (.1)(-30)(1) = 1 - 3 = -2 \\
w_2 &= -2 + (.1)(-30)(-2) = -2 + 6 = 4 \\
w_3 &= 4 + (.1)(-30)(4) = -8 \\
w_4 &= -8 + (.1)(-30)(-8) = 16
\end{aligned}$$

In general, we will obtain the "approximate" solution

$$w_i = (-2)^i$$

which looks nothing like the exact solution! In fact, this solution blows up whereas the actual solution decays quickly to 0.

Let's try again with $h = .05$.

$$\begin{aligned}
w_0 &= y(0) = 1 \\
w_1 &= 1 + (.05)(-30)(1) = 1 - \frac{3}{2} = -\frac{1}{2} \\
w_2 &= -\frac{1}{2} + (.05)(-30)(-\frac{1}{2}) = -\frac{1}{2} + \frac{3}{4} = \frac{1}{4} \\
w_3 &= \frac{1}{4} + (.05)(-30)(\frac{1}{4}) = \frac{1}{4} - \frac{3}{8} = -\frac{1}{8}
\end{aligned}$$

In general, we will obtain the approximate solution

$$w_i = \frac{1}{(-2)^i}$$

which now oscillates around the exact solution. This behavior is much better than before, but still not exactly what we want.

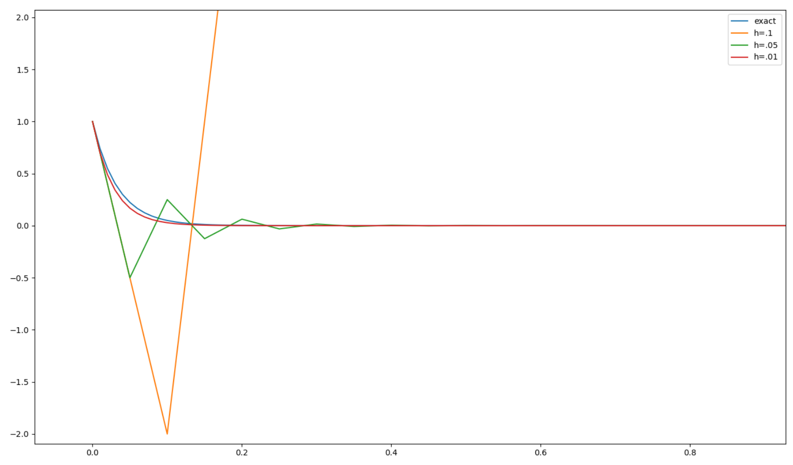
Finally, let's try $h = .01$.

$$\begin{aligned}
w_0 &= y(0) = 1 \\
w_1 &= 1 + (.01)(-30)(1) = 1 - .3 = .7 \\
w_2 &= .7 + (.01)(-30)(.7) = .7 - .21 = .49 \\
w_3 &= .49 + (.01)(-30)(.49) = .49 - 147 = .343
\end{aligned}$$

In general, we will obtain the approximate solution

$$w_i = .7^i$$

which decays to 0 like the exact solution does. This behavior is much better than before, and getting closer and closer to the exact solution. Plotting everything in one graph gives



Note that the main idea in adaptive methods is to concentrate computational power when the function has "low complexity" (in terms of the derivative, this means where the function gets flatter and flatter). However, if you were to try this above (for example, when $t > .02$ the function looks more or less flat), you would quickly find that your approximation starts to blow up again, so the step size needs to remain relatively small throughout the entire computation in order to achieve anything even remotely reasonable.

In general, we see that if we apply Euler's method with step size h to the initial value problem

$$y' = ky \qquad t \geq 0 \qquad y(0) = 1$$

with $k < 0$, then we will obtain the approximation

$$w_i = (1 + hk)^i w_0 = (1 + hk)^i.$$

(This system is a *test equation* for stiffness.) If we furthermore have a roundoff error in the initial value $\tilde{w_0} = w_0 + \delta_0$, then the same will apply for the roundoff errors as well:

$$\tilde{w_i} = (1 + hk)^i (w_0 + \delta_0) = (1 + hk)^i + (1 + hk)^i \delta_0.$$

This will only converge if $|1 + hk| < 1$ (both for the actual solution as well as for the roundoff error), otherwise we will obtain a solution that oscillates forever if $|1+hk| = 1$ and a solution that blows up quickly if $|1 + hk| > 1$. This implies that we may only choose the step size $h < \frac{2}{|k|}$ for this initial value problem if we want to obtain a solution. The function $\varphi(hk) = 1+hk$ is known as the *stability function* of Euler's method because it controls when we can expect Euler's method to give a reasonable approximation as well as be numerically stable with respect to roundoff issues.

In general, any one step method applied to the initial value problem

$$y' = ky \qquad t \geq 0 \qquad y(0) = 1$$

with $k < 0$ will yield an approximation of the form

$$w_i = \varphi(hk)^i$$

and the accuracy of the method depends critically on whether the stability function $phi(hk)$ satisfies $|\varphi(hk)| < 1$ or not. The situation is slightly more complicated for multistep methods. In general, any m-step multistep method applied to the initial value problem

$$y' = ky \qquad t \geq 0 \qquad y(0) = 1$$

with $k < 0$ will yield an approximation of the form

$$w_i = \sum_{j=1}^{m} c_j * \beta_j^i,$$

where the β_j are the (distinct) roots of the stability function $\varphi(x, hk)$ associated to the multistep method. (If the roots of $\varphi(x, hk)$ are not distinct, then a slight modification is necessary.) We can see here that if $|\beta_j| \geq 1$ for any j, then the solution will not converge for a similar reason as before. This leads us to the definition of the *region of absolute stability.*

Definition 8.2. The region of absolute stability (RAS) for a one-step method with stability function $\varphi(z)$ is $\{z \in \mathbb{C} \mid |\varphi(z)| < 1\}$. The RAS for a multistep method with stability function $\varphi(x, z)$ is $\{z \in \mathbb{C} \mid |\beta_j| < 1 \text{ for all roots } \beta_j \text{ of } Q(x, z)\}$.

The reason for this name is that a method can be reasonably applied to a stiff equation if and only if $h\lambda$ lies with the region of absolute stability of the method throughout the interval of t values for the approximation. Note that we only need to check the particular initial value problem above to determine the stability properties of a numerical ode solver. One particular notion of stability is *A-stability*, where the RAS contains the entire left half-plane (all points $z \in \mathbb{C}$ with $\text{Re}(z) < 0$). These methods are particularly attractive for solving stiff equations due to their stability properties. However, it turns out that A-stability is essentially incompatible with explicit methods, and is even fairly rare among implicit methods as well. (You can read about the "second Dahlquist barrier".) Despite the pessimism about stability, we conclude with some small hope in the form of the implicit Euler method which uses the update step

$$w_{i+1} = w_i + hf(t_{i+1}, w_{i+1})$$

and the implicit trapezoid method which uses the update step

$$w_{i+1} = w_i + \frac{h}{2}(f(t_i, w_i) + f(t_{i+1}, w_{i+1})).$$

(There is more hope to be found in implicit Runge-Kutta methods. Check e.g. Hammer-Hollingsworth which combines ideas from Gaussian integration with Runge-Kutta methods.)

Example 8.3. Consider the same initial value problem as before

$$y' = -30y \qquad t \geq 0 \qquad y(0) = 1.$$

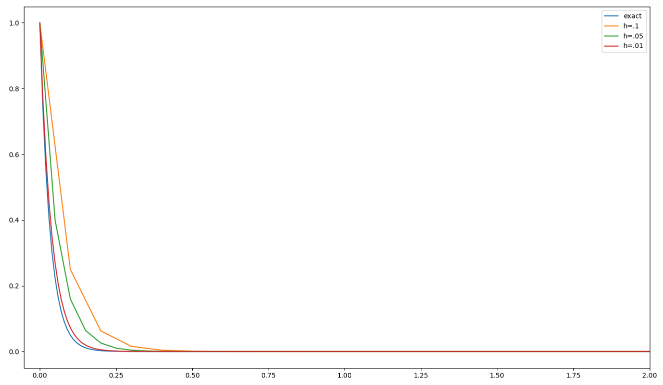
We will apply the implicit Euler method with a step size of h.

$$\begin{aligned}
w_0 &= y(0) = 1 \\
w_1 &= 1 + h(-30)(w_1) \implies w_1 = \frac{1}{1+30h} \\
w_2 &= \frac{1}{1+30h} + h(-30)(w_2) \implies w_2 = \frac{1}{(1+30h)^2}
\end{aligned}$$

and in general, we obtain

$$w_i = \frac{1}{(1+30h)^i}$$

which converges for any positive step size $h > 0$. This is much better than we saw with the explicit Euler's method, because we now obtain convergence of the solution to 0 (matching the behavior of the exact solution) for *any* step size! We also removed the nasty oscillation behavior that we saw before, which could happen even if the solution converges.



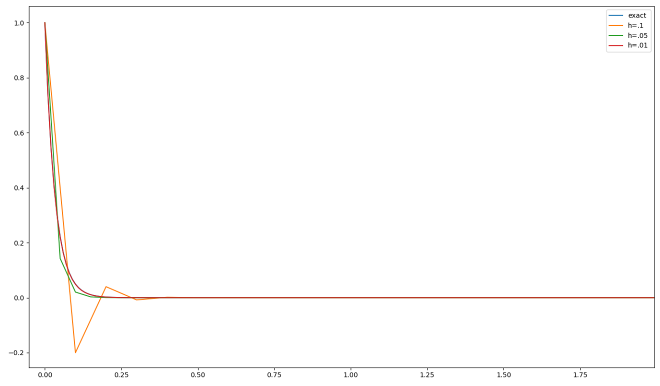
Next, we will apply the implicit trapezoid rule with a step size of h.

$$\begin{aligned}
w_0 &= y(0) = 1 \\
w_1 &= 1 + \frac{h}{2}((-30)(1) + (-30)(w_1)) \implies w_1 = \frac{1-15h}{1+15h} \\
w_2 &= \frac{1-15h}{1+15h} + \frac{h}{2}(-30\frac{1-15h}{1+15h} - 30w_2) \implies w_2 = \left(\frac{1-15h}{1+15h}\right)^2
\end{aligned}$$

and in general, we obtain

$$w_i = \left(\frac{1-15h}{1+15h}\right)^i$$

which also has good convergence behavior, though we do see some oscillation. While solving for w_i was easy for the test equation (we solved the equations algebraically!), in general we will need to apply a fancier root-finding technique such as Newton's method. However, the improvement in performance and ability to use larger step sizes more than compensates for this additional complexity. Thus, for stiff equations, we should always try to use implicit methods, even if it costs a bit more due to needing to use a root-finding method.


exact
h=.1
h=.05
h=.01