Integration of Ordinary Differential Equations

The solution of differential equations is an important problem that arises in a host of areas.
Many differential equations are too difficult to solve in closed form. Instead, it becomes necessary to employ numerical techniques. We will study some of these.

Some Applications:

□ Understanding the dynamics of some physical system.

Classic examples:
- aerodynamics
- fluid dynamics
- predator-prey
- heat diffusion
- mechanical oscillations (drums, beams)

⋮

Other examples:
- robots

  bouncing, bending

  Raibert, Canon, Koditschek, Shuttle arm

□ Developing control algorithms and strategies.

□ Solving optimization problems.

□ Understanding stochastic systems.

The subject is vast and deep. We will barely penetrate the surface. We will stick to ordinary differential equations (as opposed to partial differential equations), and we will focus on two classes of differential equations:

- First-order initial-value problems.
- Linear higher-order boundary-value problems.

Fortunately, the basic principles we will see in these two classes of problems carry over to more general settings. For instance, higher-order initial-value problems can be rewritten in vector form to yield a set of simultaneous first-order equations. First-order techniques may then be used to solve this system.

---

nth-order differential equation

We assume that a general nth-order differential equation is written in the form

$$y^{(n)}(x) = f(x, y(x), y'(x), \ldots, y^{(n-1)}(x)),$$

where f is some function $f: \mathbb{R}^n \to \mathbb{R}$.
A general solution of this equation will usually contain n arbitrary constants. A particular solution is obtained by specifying n auxiliary constraints. For instance, one might specify the values of y and its derivatives at some point $x = x_0$.

In other words, one specifies $y(x_0), y'(x_0), \ldots, y^{(n-1)}(x_0)$.

Such a problem is called an <u>initial-value problem.</u> In effect, the auxiliary conditions specify all the relevant information at some starting point x_0, and the differential equation tells us how to proceed from that point.

If the n auxiliary conditions are specified at different points, then the problem is called a <u>boundary-value problem.</u> In effect, we tie the function (or its derivatives) down at several points, and the differential equation tells us the shape of $y(x)$ between those points.

<u>First-order Initial-value problems</u>

I assume that you are familiar with the basic techniques for solving such problems analytically. We will here focus on numerical techniques. (Numerical techniques are useful both for non-linear differential equations and for equations whose analytic solutions consist of complicated integrals and exponentials and ...)

So, our basic problem is to recover $y(x)$ from the differential equation

$$\frac{dy}{dx} = f(x, y)$$

with initial condition $y(x_0) = y_0$.

Taylor's Algorithm

Suppose $y(x)$ is an exact solution to the basic problem. If y has a Taylor series expansion about x_0, then we can write

$$y(x) = y_0 + (x-x_0)\,y'(x_0) + \frac{(x-x_0)^2}{2!}\,y''(x_0) + \cdots$$

Of course, if we do not know $y(x)$, then we do not explicitly know the values of the derivatives, $y'(x_0), y''(x_0), \ldots$ However, if f is sufficiently differentiable, then we can obtain the derivatives of y from f:

$$y'(x) = f(x, y(x)) \quad \longleftarrow \text{this is just the diff. eq.}$$

i.e., $y' = f$

So $y'' = f_x + f_y y'$

(*) $= f_x + f_y f$

$$\begin{aligned} y''' &= f_{xx} + f_{xy}f + f_{yx}f + f_{yy}f^2 + f_y f_x + f_y^2 f \\ &= f_{xx} + 2f_{xy}f + f_{yy}f^2 + f_x f_y + f_y^2 f \end{aligned}$$

etc.

Let us denote by <u>$y(x_n)$ the true value of $y(x)$</u> at the point $x = x_n$, and by <u>y_n an approximate value</u>. We will generate a series of points $x_0, x_1, \ldots$ and corresponding approximate values $y_0, y_1, \ldots$ by marching out from x_0. We will obtain y_{n+1} from y_n by using a truncated Taylor series, with approximate derivative values obtained from (*).

Suppose we fix a step size $h > 0$.
So our points of interest $x_0, x_1, x_2, \dots$ are

$$x_0,\ x_0 + h,\ x_0 + 2h, \dots$$

Let us define a truncated Taylor series operator:

Def $$T_k(x,y) = f(x,y) + \frac{h}{2!} f'(x,y) + \dots + \frac{h^{k-1}}{k!} f^{(k-1)}(x,y)$$

for each $k = 1, 2, \dots$

where $$f^{(j)}(x,y) = \frac{d^j}{(dx)^j} f(x, y(x))$$

Observe that

- $$\begin{aligned} f(x,y) &= y' \\ f'(x,y) &= y'' = f_x + f_y f \\ f''(x,y) &= y''' = \dots \\ &\vdots \end{aligned}$$

- Taylor's theorem tells us that for each k

$$y(x+h) = y(x) + h T_k(x,y) + \text{remainder that depends on } y^{(k+1)}$$

This formula provides us with a numerical method for obtaining y_{n+1} from y_n.

Taylor's Algorithm of order k

Objective: To find an approximate solution to the differential equation

$$y'(x) = f(x,y)$$
$$y(a) = y_0$$

over the interval $[a,b]$.

Method:

1. Fix some discretization by picking an integer N. Let $h = \frac{b-a}{N}$ and set $x_n = a + nh$, $n = 0, 1, \ldots, N$.
2. For $n = 0, 1, \ldots, N-1$ Do:
$$y_{n+1} \leftarrow y_n + h\,T_k(x_n, y_n)$$

Terminology: The calculation of y_{n+1} uses information about y and its derivatives that comes from a single point, namely from x_n. For this reason Taylor's Algorithm is a <u>one-step</u> method.

You are probably very familiar with a special case, namely Taylor's Algorithm of order 1, also known as <u>Euler's method</u>. The basic update rule becomes:

$$\boxed{y_{n+1} \leftarrow y_n + h f(x_n, y_n)}$$

Unfortunately this method is not that great. It requires very small step sizes to obtain good accuracy. And there are stability problems (ie, errors accumulate). Nonetheless, the basic idea of adding small increments to previous estimates to obtain new estimates is central.

Error Estimates

We would like to understand the quality of our differential equation solvers by estimating the error between $y(x_n)$ and y_n.

There are three types of errors:

1. Local Discretization Error

   This is the error introduced in a single step of the equation solver as it moves from x_n to x_{n+1}. In other words, it is the error in the estimate y_{n+1} that would result if $y(x_n)$ were known perfectly.

2. Full Discretization Error

   This is the net error between $y(x_n)$ and y_n at step # n. This error is the sum of the local discretization errors, plus any numerical roundoff errors.

3. Numerical Roundoff Error

   Limited machine precision can introduce errors. We will ignore this type of error.

Local Error

Taylor's theorem tells us what the local discretization error is in Taylor's Algorithm. It is simply the remainder term.

So, the local error for Taylor's Algorithm of order k is

$$E = \frac{h^{k+1} f^{(k)}(\xi, y(\xi))}{(k+1)!}$$

where ξ is some point in the interval $(x_n, x_n + h)$. We can rewrite this as

$$E = \frac{h^{k+1} y^{(k+1)}(\xi)}{(k+1)!}$$

In general, an algorithm is said to be <u>of order k</u> if its local discretization error is $O(h^{k+1})$. So, Taylor's Algorithm of order k is indeed of order k.

Note that Euler's algorithm is of order 1 since

$$E = \frac{h^2 y''(\xi)}{2}.$$

Full Discretization Error

Estimating the full discretization error can be very difficult. Let us therefore illustrate the approach with Euler's method.

Define: $e_n = y(x_n) - y_n$

Taylor's theorem with remainder says that

$$y(x_{n+1}) = y(x_n) + h y'(x_n) + \frac{h^2}{2} y''(\xi_n)$$

for some $\xi_n \in (x_n, x_{n+1})$.

Recall that $y_{n+1} = y_n + h f(x_n, y_n)$

So $$e_{n+1} = e_n + h\left[f(x_n, y(x_n)) - f(x_n, y_n)\right] + \frac{h^2}{2} y''(\xi_n).$$

Applying the mean-value theorem to f we see that

$$e_{n+1} = e_n + h f_y(x_n, \bar{y}_n)(y(x_n) - y_n) + \frac{h^2}{2} y''(\xi_n)$$

for some $\bar{y}_n$ between $y(x_n)$ & y_n

So, $e_{n+1} = e_n \left(1 + h f_y(x_n, \bar{y}_n)\right) + \frac{h^2}{2} y''(\xi_n)$

In order to get some handle on this recurrence relation for e_n, let's assume that f_y and y'' are both bounded on $[a,b]$. Say, $\left.\begin{aligned} &|f_y(x,y(x))| \le L \\ \text{and } &|y''(x)| \le Y \end{aligned}\right\}$ for $x \in [a,b]$.

Then $|e_{n+1}| \le (1+hL)|e_n| + \frac{h^2}{2}Y$

Since $e_0 = 0$ and since $1+hL > 1$, a short induction proof shows that $|e_n| \le \eta_n$, where η_n satisfies the recurrence

$$\eta_{n+1} = (1+hL)\eta_n + \frac{h^2}{2}Y.$$

Consequently

$$\begin{aligned} |e_n| \le \eta_n &= \frac{hY}{2L}\left[(1+hL)^n - 1\right] \\ &\le \frac{hY}{2L}\left[e^{nhL} - 1\right) \\ &= \frac{hY}{2L}\left(e^{(x_n - x_0)L} - 1\right) \end{aligned}$$

<u>This says two things:</u>

(i) The full discretization error approaches zero as the step size h is made smaller. That's nice to know.

(ii) The full discretization error is $O(h)$.
[This is at fixed x ~~—~~ n is of course different.]
This means that convergence may be slow.

<u>Ex</u> Consider the differential equation

$$y' = -y^2 \qquad y(1) = 1$$

Suppose we wish a solution over the x-interval $[1,2]$.

Let's use Euler's method with step-size $h = 0.1$.
Euler's method says

$$y_{n+1} = y_n + h f(x_n, y_n), \qquad n = 0, 1, \dots$$

For this problem $y_0 = 1$ and $f(x_n, y_n) = -y_n^2$

Observe that the exact solution is $y(x) = \frac{1}{x}$.
We will compare our numerical solution against the exact solution.

<u>What kind of errors might we expect?</u>

Well, our previous calculations suggest the following bounds:

<u>local error:</u>

$$|E| \le \max_{1 \le x \le 2} \frac{h^2}{2} |y''(x)| = \max_{1 \le x \le 2} \frac{h^2}{2} \frac{2}{x^3} \le .01$$

<u>full error:</u> We use $Y = 2$ since $y''(x) = \frac{2}{x^3}$
And we use $L = 2$ since $f_y(x, y(x)) = -2y = -\frac{2}{x}$

so $$|e_n| \le \frac{h}{2}\left(e^{2(x_n - 1)} - 1\right)$$

$$\max_n |e_n| \le \frac{h}{2}\left(e^2 - 1\right) \approx .0859$$

So we should expect about 1 decimal digit of accuracy.

The following table prints the actual results obtained (Notation: f_n means $f(x_n, y_n)$.)

			exact solution
x_n	y_n	f_n	$y(x_n)$
1.0	1.0	−1.0	1.0
1.1	0.9	−0.810	0.90909
1.2	0.819	−0.67076	0.83333
1.3	0.7519	−0.56539	0.76923
1.4	0.69539	−0.48356	0.71429
1.5	0.64703	−0.41865	0.66667
1.6	0.60516	−0.36622	0.625
1.7	0.56854	−0.32324	0.58824
1.8	0.53622	−0.28753	0.55556
1.9	0.50746	−0.25752	0.52632
2.0	0.48171	−0.23205	0.5

The maximum error is just under 0.02, well within our estimates. Still, .02 isn't all that great.

Runge-Kutta Methods

Euler's method is unacceptable because it requires small step sizes. Higher-order Taylor algorithms are unacceptable because they require higher-order derivatives.

Runge-Kutta methods attempt to obtain greater accuracy than, say, Euler's method, without explicitly evaluating higher order derivatives. Instead, these methods evaluate $f(x,y)$ at selected points in the interval $[x_n, x_{n+1}]$.

The idea is to combine the estimates of y resulting from these selected evaluations in such a way that error terms of order $h, h^2, \ldots,$ etc. are cancelled out, to whatever accuracy one desires.

Forinstance, the Runge-Kutta method of order 2 tries to cancel terms of order h and h^2, leaving a local discretization error of order h^3.

Similarly, the Runge-Kutta method of order 4 tries to cancel terms of order h, h^2, h^3, h^4, leaving an $O(h^5)$ local discretization error.

Runge - Kutta Method of Order 2

For the Runge-Kutta method of order 2 we wish to evaluate $f(x, y)$ at two points in the interval $[x_n, x_{n+1}]$, then combine the results to obtain an error of order h^3. The basic step of the algorithm is:

$$y_{n+1} = y_n + ak_1 + bk_2$$

with
$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf(x_n + \alpha h, y_n + \beta k_1)$$

$a, b, \alpha,$ and β are fixed constants.
One chooses these in such a way as to obtain the $O(h^3)$ local discretization error.

Intuitively, what is the basic step doing?
Well, it first evaluates y' at (x_n, y_n) by computing $y'_n = f(x_n, y_n)$. The algorithm then tentatively uses this approximation to y' to step to the trial point $(x_n + \alpha h, y_n + \beta k_1)$. At this trial point, the algorithm re-evaluates the derivative y', then uses both derivative estimates to compute y_{n+1}.
So, by using two points to compute y', the algorithm should be obtaining second derivative information, hence be able to reduce the size of the discretization error.

Let's derive the constants $a, b, \alpha, + \beta$.

By Taylor's theorem + our expansions on page 4,

$$y(x_{n+1}) = y(x_n) + hy'(x_n) + \frac{h^2}{2}y''(x_n) + \frac{h^3}{6}y'''(x_n) + \cdots$$

(*)
$$= y(x_n) + hf(x_n, y_n) + \frac{h^2}{2}\left(f_x + ff_y\right)\Big|_{(x_n, y_n)}$$

$$+ \frac{h^3}{6}\left(f_{xx} + 2f_{xy}f + f_{yy}f^2 + f_xf_y + f_y^2 f\right)\Big|_{(x_n, y_n)}$$

$$+ O(h^4)$$

And Taylor in 2D tell us that

$$\frac{k_2}{h} = f(x_n + \alpha h, y_n + \beta k_1)$$

$$= f(x_n, y_n) + \alpha h f_x + \beta k_1 f_y + \frac{\alpha^2 h^2}{2} f_{xx} + \frac{\beta^2 k_1^2}{2} f_y f_y$$

$$+ \alpha h \beta k_1 f_{xy} + O(h^3)$$

[again, all terms are evaluated at (x_n, y_n).]

So $y_{n+1} = y_n + ak_1 + bk_2$

(**)
$$= y_n + (a+b)hf + bh^2\left(\alpha f_x + \beta f f_y\right)$$

$$+ bh^3\left(\frac{\alpha^2}{2} f_{xx} + \alpha\beta f f_{xy} + \frac{\beta^2}{2} f^2 f_{yy}\right)$$

$$+ O(h^4)$$

We want y_{n+1} to estimate $y(x_{n+1})$ with $\mathcal{O}(h^3)$ local error. So, let's equate the expressions (*) and (**), matching up terms with like powers of h, through h^2.
We get the following constraint equations for a, b, α, and β:

$a+b=1$ — this matches the h^1 terms

$\alpha b = \frac{1}{2}$ and $\beta b = \frac{1}{2}$ — this matches the h^2 terms

[For the purposes of local discretization error, $y(x_n)$ matches y_n, so there are no h^0 terms to match.]

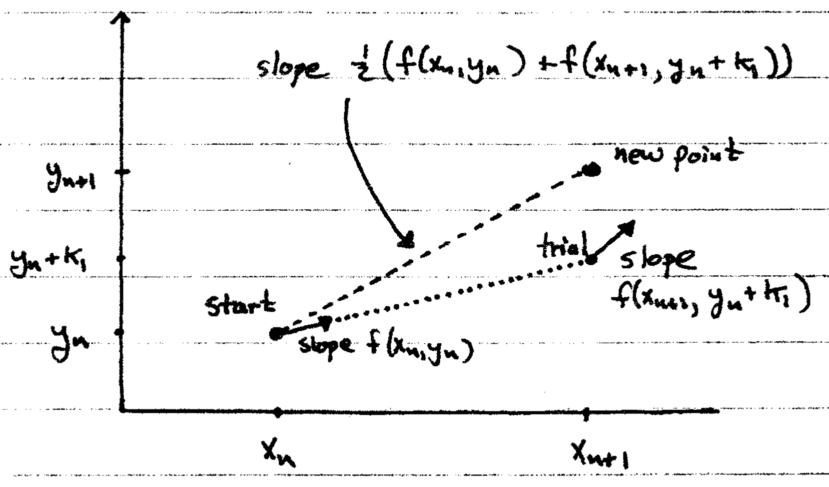
Comment: We really have a family of second order Runge-Kutta methods, since there are 4 constants but only 3 constraint equations

One popular method is to choose $a=b=\frac{1}{2}$ and $\alpha=\beta=1$. This effectively places the "trial" point at x_{n+1}, then moves from y_n to y_{n+1} by averaging the derivatives computed at x_n and x_{n+1}.

$$y_{n+1} = \frac{1}{2}(k_1+k_2)+y_n$$

with $k_1 = hf(x_n, y_n)$

$k_2 = hf(x_n+h, y_n+k_1)$


slope $\frac{1}{2}(f(x_n,y_n)+f(x_{n+1},y_n+k_1))$
new point
y_{n+1}
y_n+k_1
trial
slope $f(x_{n+1}, y_n+k_1)$
start
y_n
slope $f(x_n,y_n)$
x_n
x_{n+1}

Comment:

The Runge-Kutta method of order 2 has a local discretization error $O(h^3)$.

Compare that to Euler's method, which had local error $O(h^2)$.

In order for a Taylor-based method to obtain $O(h^3)$ error, it must compute second derivatives. Runge-Kutta-order-2 does not need to compute second derivatives. Instead it performs two evaluations of the first derivative.

Runge-Kutta of Order 4

A widely used method is the following order 4 Runge-Kutta method. It produces $O(h^5)$ local discretization error, and thus leads to solutions quickly and accurately.

$$y_{n+1} = y_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

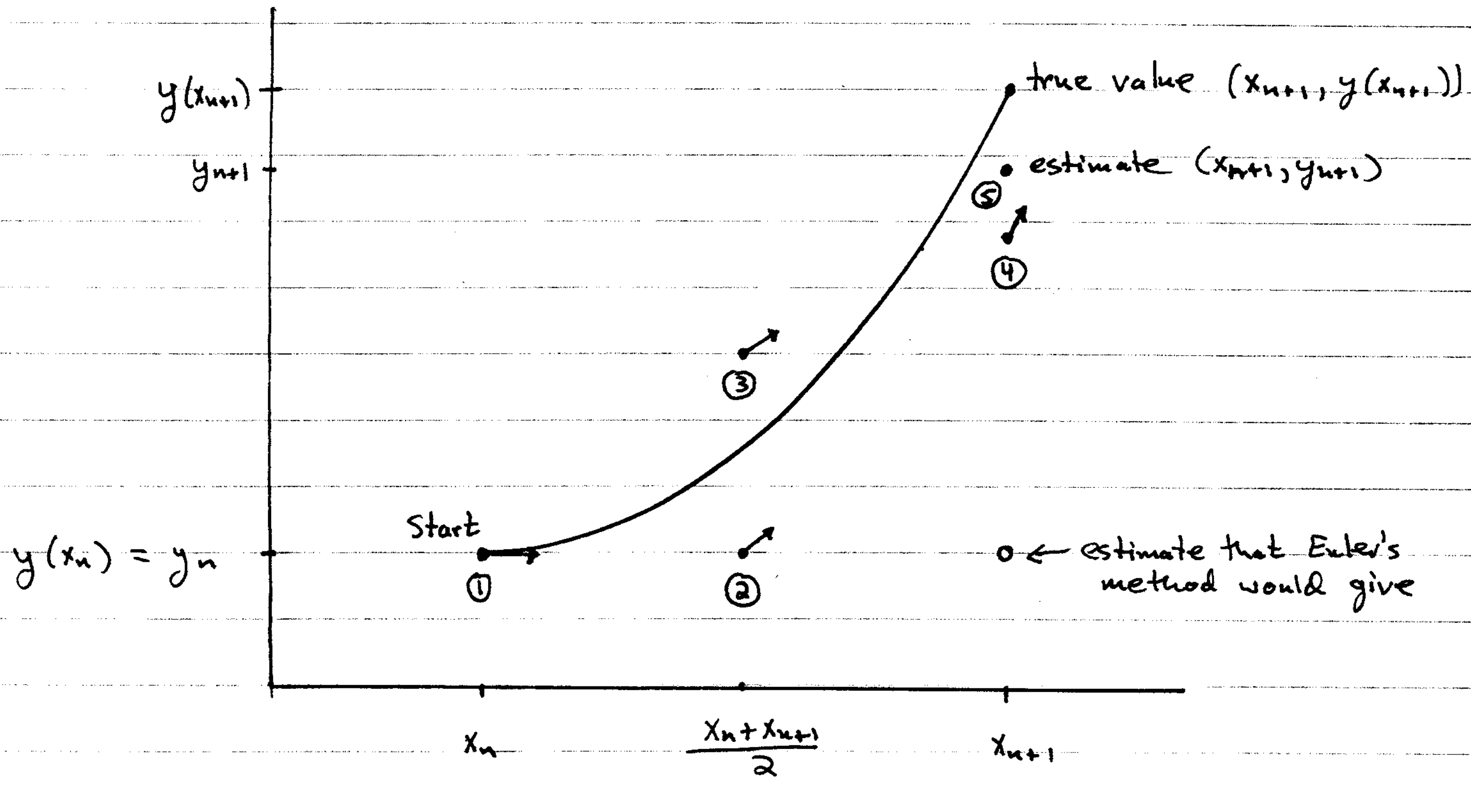
with $k_1 = hf(x_n, y_n)$ — this estimates y' at the start point

$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$ — this estimates y' at the midpoint between x_n + x_{n+1}, using a y-value obtained with k_1.

$k_3 = hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2})$ — this again estimates y' at the midpoint $\frac{x_n + x_{n+1}}{2}$, but with an updated y-value, obtained by using k_2.

$k_4 = hf(x_n + h, y_n + k_3)$ — this estimates y' at the point x_{n+1}, using a y-value obtained by advancing from y_n by k_3.

Picture   Runge-Kutta of order 4



$y(x_{n+1})$
y_{n+1}
$y(x_n) = y_n$
Start
1
2
3
4
5
true value $(x_{n+1}, y(x_{n+1}))$
estimate (x_{n+1}, y_{n+1})
estimate that Euler's method would give
x_n
$\frac{x_n + x_{n+1}}{2}$
x_{n+1}

The solid curve is an integral curve, i.e., a solution to the differential equation $y'(x) = f(x,y)$. If $y(x_n) = y_n$ as in the picture, then we desire that $y_{n+1} = y(x_{n+1})$. The picture shows the process whereby y_{n+1} is computed. Points are labelled by numbers in the order that they are computed. We assume that the vector field given by $y' = f(x,y)$ is continuous, and that other integral curves locally look "like" the one shown.

Ex Recall from page 10.1 the differential equation

$$y' = -y^2 \qquad y(1) = 1.$$

If we solve this problem over the interval $[1,2]$, using the previous Runge-Kutta method of order 4 with a step-size $h = 0.1$, we get the following data:

x_n	y_n	f_n	exact solution $y(x_n)$
1.0	1.0	-1.0	1.0
1.1	0.90909	-0.82645	0.90909
1.2	0.83333	-0.69445	0.83333
1.3	0.76923	-0.59172	0.76923
1.4	0.71429	-0.51020	0.71429
1.5	0.66667	-0.44445	0.66667
1.6	0.62500	-0.39063	0.625
1.7	0.58824	-0.34602	0.58824
1.8	0.55556	-0.30864	0.55556
1.9	0.52632	-0.27701	0.52632
2.0	0.50000	-0.25000	0.5

Compare this with the results of Euler's method shown on page 10.2. To the accuracy shown — five decimal digits — there is no error. In fact, the error was less than $5 \cdot 10^{-7}$. That's not bad. The $O(h^5)$ local error of RK-4 is quite good.

Multi-Step Formulas

Runge-Kutta wins big because it really makes several small steps for each major step of the form $x_n \to x_{n+1}$.

This suggests an alternative:

In moving from x_n to x_{n+1}, make use of prior information at $x_{n-1}, x_{n-2}, \dots$. In other words, rather than evaluating $f(x,y)$ at several intermediate points, make use of known values of $f(x,y)$ at several past points.

Methods that make use of information at several of the $\{x_i\}$ are called <u>multi-step</u> <u>methods</u>.

We will here look at one such method, known as the Adams-Bashforth method.

Suppose we have approximations to $y(x)$ and $y'(x)$ at the points $x_0, \dots, x_n$.

If we integrate the differential equation

$$y'(x) = f(x, y(x))$$

from x_n to x_{n+1}, we obtain

$$\int_{x_n}^{x_{n+1}} y'(x)\,dx = \int_{x_n}^{x_{n+1}} f(x, y(x))\,dx$$

Hence $$y_{n+1} = y_n + \int_{x_n}^{x_{n+1}} f(x, y(x))\,dx$$

<u>How do we evaluate the integral?</u>

The trick is to approximate $f(x, y(x))$ with an interpolating polynomial. — Of course, we don't know exact values of $f(x, y(x))$ anywhere except at x_0. However, we do have approximate values $f(x_k, y_k)$ for $k = 0, 1, \ldots, n$. So, we will construct an interpolating polynomial that passes through some of those values.

Specifically, if we are interested in an order $m+1$ method we will approximate the function $f(x, y(x))$ with an interpolating polynomial that passes through the $m+1$ points $(x_{n-m}, f_{n-m}), \ldots, (x_n, f_n)$. Here $f_i = f(x_i, y_i)$.

We then integrate this polynomial in order to approximate the integral $\int_{x_n}^{x_{n+1}} f(x, y(x))\,dx$

In the first part of the course, we wrote interpolating polynomials in terms of divided differences. It is also possible to write interpolating polynomials in terms of forward differences. This will be convenient now.

Def Forward differences are defined recursively as:

$$\Delta^i f_k = \begin{cases} f_k & \text{if } i=0 \\ \Delta^{i-1} f_{k+1} - \Delta^{i-1} f_k & \text{if } i>0 \end{cases}$$

A forward difference table looks like this:

x_i	Δ^0	Δ^1	Δ^2	Δ^3	Δ^4	Δ^5
x_0	f_0					
		Δf_0				
x_1	f_1		$\Delta^2 f_0$			
		Δf_1		$\Delta^3 f_0$		
x_2	f_2		$\Delta^2 f_1$		$\Delta^4 f_0$	
		Δf_2		$\Delta^3 f_1$		$\Delta^5 f_0$
x_3	f_3		$\Delta^2 f_2$		$\Delta^4 f_1$	
		Δf_3		$\Delta^3 f_2$		
x_4	f_4		$\Delta^2 f_3$			
		Δf_4				
x_5	f_5					

One nice thing about forward differences is that we don't do any divisions.

It turns out that one can write the order m interpolating polynomial that passes through the points (x_i, f_i), $i = n-m, \dots, n$ as

$$p_m(x) = \sum_{k=0}^{m} (-1)^k \binom{-s}{k} \Delta^k f_{n-k},$$

where $s = \dfrac{x - x_n}{h}$

(so this is where the division occurs)

[Aside: The symbol $\binom{w}{k}$ where w is a real number and k is an integer

means: $\binom{w}{0} = 1$ for all w

$$\binom{w}{k} = \frac{w(w-1)\cdots(w-k+1)}{k!} \quad \text{if } k \ge 1.]$$

So

$$y_{n+1} \approx y_n + h \int_0^1 \sum_{k=0}^{m} (-1)^k \binom{-s}{k} \Delta^k f_{n-k} \, ds$$

(since $dx = h\,ds$)

$$= y_n + h \left\{ \gamma_0 f_n + \gamma_1 \Delta f_{n-1} + \cdots + \gamma_m \Delta^m f_{n-m} \right\}$$

where $\gamma_k = (-1)^k \int_0^1 \binom{-s}{k} ds$

<u>Note:</u> The $\{\gamma_k\}$ are precomputable numbers.

One popular Adams-Bashforth method is the order 4 method. Let's derive it.

We need to compute $\gamma_0, \gamma_1, \gamma_2$, and γ_3.

$$\gamma_0 = 1$$

$$\gamma_1 = -\int_0^1 (-s)\,ds = \frac{1}{2}$$

$$\gamma_2 = \int_0^1 \frac{(-s)(-s-1)}{2}\,ds = \frac{5}{12}$$

$$\gamma_3 = -\int_0^1 \frac{(-s)(-s-1)(-s-2)}{6}\,ds = \frac{3}{8}$$

So $$y_{n+1} = y_n + h\left(f_n + \frac{1}{2}\Delta f_{n-1} + \frac{5}{12}\Delta^2 f_{n-2} + \frac{3}{8}\Delta^3 f_{n-3}\right)$$

Expanding out, we have
$$\Delta f_{n-1} = f_n - f_{n-1}$$
$$\Delta^2 f_{n-2} = f_n - 2f_{n-1} + f_{n-2}$$
$$\Delta^3 f_{n-3} = f_n - 3f_{n-1} + 3f_{n-2} - f_{n-3}$$

So, the update formula becomes

$$\boxed{y_{n+1} = y_n + \frac{h}{24}\left(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}\right)}$$

It turns out that the local discretization error is given by

$$E = \frac{251}{720} h^5 y^{(v)}(\xi) \text{ for some } \xi \in (x_{n-3}, x_{n+1}).$$

Ex Let's return to our example $y' = -y^2$, $y(1) = 1$, to be solved over the interval $[1,2]$.

In order to start off the order 4 Adams-Bashforth method we need to know the value of $f(x, y(x))$ at the four points x_0, x_1, x_2, x_3. In general one obtains these either by knowing them, or by running a highly accurate method that computes approximations to $y(x_1)$, $y(x_2)$, + $y(x_3)$. In our case, we'll take the values obtained by the Runge-Kutta example of page 18.

If we use a step-size $h = 0.1$, we obtain the following data:

x_n	y_n	f_n	exact solution $y(x_n)$
1.0	1.0	−1.0	1.0
1.1	0.90909	−0.82645	0.90909
1.2	0.83333	−0.69445	0.83333
1.3	0.76923	−0.59172	0.76923
1.4	0.71444	−0.51042	0.71429
1.5	0.66686	−0.44470	0.66667
1.6	0.62525	−0.39093	0.625
1.7	0.58848	−0.34631	0.58824
1.8	0.55580	−0.30892	0.55556
1.9	0.52655	−0.27726	0.52632
2.0	0.50023	−0.25023	0.5

Notice that the maximum error is approximately $2.5 \cdot 10^{-4}$.
This is ~~indeed~~ even within the bound we obtained for the local error, namely
$$|E| \le \max_{1 \le x \le 2} \frac{251}{720} h^5 |y^{(v)}(x)| = \max_{1 \le x \le 2} \frac{251}{720} (.1)^5 \frac{120}{x^6} \doteq 4.2 \cdot 10^{-4}.$$

Comparison of Runge-Kutta and Adams-Bashforth

We have seen two order 4 methods.
How do they compare?

Error

Both methods are order 4 methods, meaning that their local discretization error is $O(h^5)$.
However, the constant coefficient in the error term tends to be higher for the Adams-Bashforth method than for the Runge-Kutta method. Consequently, the Runge-Kutta method tends to exhibit greater accuracy. We saw this in the last example.

Function Evaluation

Both methods require evaluation of $f(x, y(x))$ at four points in order to move from x_n to x_{n+1}. However, while Runge-Kutta must generate three intermediate points at which to evaluate $f(x, y(x))$, Adams-Bashforth uses information already available. Consequently, Adams-Bashforth requires less computation and is therefore faster. One drawback is that the method is not "self-starting"; one must supply four initial data points rather than just one.

Comment:

There are multi-step methods that integrate over several steps.

Basic formula:

$$y_{n+1} = y_{n-p} + h \int_{-p}^{1} \sum_{k=0}^{m} (-1)^k \binom{-s}{k} \Delta^k f_{n-k}\, ds$$

($p=0$ corresponds to Adams-Bashford,

so integration is from x_{n-p} to x_{n+1} using interpolation at the points $x_n, x_{n-1}, \ldots, x_{n-m}$

Some typical cases:

$p=1, m=1$ $\quad y_{n+1} = y_{n-1} + 2hf_n, \quad E = \frac{h^3}{3} y'''(\xi)$.

$p=3, m=3$ $\quad y_{n+1} = y_{n-3} + \frac{4h}{3}(2f_n - f_{n-1} + 2f_{n-2}), \quad E = \frac{14}{45} h^5 y^{v}(\xi)$.

The $p=1, m=1$ case is similar to Euler in simplicity but has a better local discretization error.

(but stability problems)

Omitted Topics

We now leave the subject of initial value problems, but you should be aware of some other important techniques and issues (see NR:C + CdB).

- Adaptive control of the step-size,
- Multi-step formulas that integrate over several intervals.
- Predictor-Corrector methods (implicit methods),
- Extrapolation to the limit,
- Numerical instability (particularly in multi-step methods),
- Stiff Equations
  (higher order equations or systems of equations with solutions that have different time scales)

Boundary Value Problems

Some examples:

① Second-order system:

$$y''(x) = y(x) \qquad \begin{matrix} y(0) = 0 \\ y(1) = 1 \end{matrix}$$

② Fourth-order system:

$$y^{(iv)}(x) + ky(x) = q \qquad \begin{matrix} y(0) = y'(0) = 0 \\ y(L) = y''(L) = 0 \end{matrix}$$

Problem ① is a classic exponential growth/decay problem.

Problem ② arises in the context of beam bending. $y(x)$ represents the beam deflection at point x along its length. q represents a uniform load. L is the beam's length. The first two boundary conditions say that the $x=0$ end of the beam is rigidly attached. The second two boundary conditions say that the $x=L$ end of the beam is simply supported.

Finite - Difference Method

We will use a finite-difference method to obtain numerical solutions to boundary value problems. We assume that our differential equation is a linear differential equation.

Suppose we are interested in a solution over some interval $[a,b]$, with boundary conditions specified at the endpoints of this interval.

We split the interval into N equal parts, each of width $h = \frac{b-a}{N}$.

- We set $x_0 = a$
  $x_N = b$

- and then we define the <u>interior</u> <u>mesh</u> <u>points</u> as

$$x_n = x_0 + nh \quad , \quad n = 1, 2, \dots, N-1$$

- Sometimes it will also be useful to define <u>exterior</u> <u>mesh</u> <u>points.</u> Examples of such points include

$$x_{-1} = x_0 - h \ , \quad x_{-2} = x_0 - 2h \ , \dots$$

$$x_{N+1} = x_N + h \ , \quad x_{N+2} = x_N + 2h \ , \dots$$

- As before we let $y(x_n)$ denote the exact solution $y(x)$ at the point $x = x_n$, and we let y_n denote our approximate solution.

- Next we set up finite difference approximations for each of the derivatives that appears in our differential equation. In order to achieve $O(h^2)$ error, we use central differences. For example,

$$y'(x_n) \approx \frac{y_{n+1} - y_{n-1}}{2h}$$

$$y''(x_n) \approx \frac{y_{n+1} - 2y_n + y_{n-1}}{h^2}$$

$$y^{(iv)}(x_n) \approx \frac{y_{n+2} - 4y_{n+1} + 6y_n - 4y_{n-1} + y_{n-2}}{h^4}$$

- We then write down the differential equation once for each interior mesh point, using the finite-difference approximations. We also write down each of the boundary conditions, using the finite-difference approximations. Notice that this may force us to create exterior mesh points. That's OK. In the end we will have created a system of linear equations

$$Ay = b.$$

If we have m mesh points total (interior, boundary, & exterior), then y is an $m \times 1$ vector representing the unknowns $\{y_i\}$, A is an $m \times m$ matrix, and b is an $m \times 1$ constant vector. The coefficients of A are determined by the differential equation and the boundary conditions, as are the coefficients of b. — For a second order system, A is tridiagonal. If h is small enough, the diagonal entries are non-zero.

Central Difference Derivations

(References:
Hildebrand, Finite Difference Equations & Simula
Hämmerlin & Hoffman, Numerical Mathematic

$$\delta^0 f_k = f_k$$

$$\delta^m f_{k+\frac{1}{2}} = \delta^{m-1} f_{k+1} - \delta^{m-1} f_k \;, \quad m = 1, 3, 5, \ldots$$

$$\delta^m f_k = \delta^{m-1} f_{k+\frac{1}{2}} - \delta^{m-1} f_{k-\frac{1}{2}} \;, \quad m = 2, 4, 6, \ldots$$

$$D^n f_k = \left.\frac{d^n f}{dx^n}\right|_{x=x_k} \approx \frac{\delta^n f_k}{h^n} \quad \text{for small } h$$

Let's compute:

1) $$\delta^1 f_{\frac{1}{2}} = \delta^0 f_1 - \delta^0 f_0 = f_1 - f_0$$

2) $$\begin{aligned}\delta^2 f_0 &= \delta^1 f_{\frac{1}{2}} - \delta^1 f_{-\frac{1}{2}} \\ &= (f_1 - f_0) - (f_0 - f_{-1}) \\ &= f_1 - 2f_0 + f_{-1}\end{aligned}$$

3) $$\begin{aligned}\delta^3 f_{\frac{1}{2}} &= \delta^2 f_1 - \delta^2 f_0 \\ &= (f_2 - 2f_1 + f_0) - (f_1 - 2f_0 + f_{-1}) \\ &= f_2 - 3f_1 + 3f_0 - f_{-1}\end{aligned}$$

4) $\delta^4 f_0 = \delta^3 f_{\frac{1}{2}} - \delta^3 f_{-\frac{1}{2}}$

$= (f_2 - 3f_1 + 3f_0 - f_{-1}) - (f_1 - 3f_0 + 3f_{-1} - f_{-2}$

$= f_2 - 4f_1 + 6f_0 - 4f_{-1} + f_{-2}$

5) $\delta^5 f_{\frac{1}{2}} = \delta^4 f_1 - \delta^4 f_0$

$= (f_3 - 4f_2 + 6f_1 - 4f_0 + f_{-1})$

$- (f_2 - 4f_1 + 6f_0 - 4f_{-1} + f_{-2})$

$= f_3 - 5f_2 + 10f_1 - 10f_0 + 5f_{-1} - f_{-2}$

This business about shifting & subtracting seems to imply a Pascal's Triangle pattern.

etc.

What does this say about derivatives at some point?

1) $D^1 f_{\frac{1}{2}} \approx \dfrac{f_1 - f_0}{h}$ (note these are $\frac{1}{2}$ steps)

and thus $D^1 f_0 \approx \dfrac{f_1 - f_{-1}}{2h}$ (for full steps)

2) $D^2 f_0 \approx \dfrac{f_1 - 2f_0 + f_{-1}}{h^2}$

3) $D^3 f_{\frac{1}{2}} \approx \dfrac{f_2 - 3f_1 + 3f_0 - f_{-1}}{h^3}$ (for $\frac{1}{2}$ steps)

and thus $D^3 f_0 \approx \dfrac{f_3 - 3f_1 + 3f_{-1} - f_{-3}}{8h^3}$ (for full steps)

4) $D^4 f_0 \approx \dfrac{f_2 - 4f_1 + 6f_0 - 4f_{-1} + f_{-2}}{h^4}$

5) $D^5 f_{\frac{1}{2}} \approx \dfrac{f_3 - 5f_2 + 10f_1 - 10f_0 + 5f_{-1} - f_{-2}}{h^5}$ (for $\frac{1}{2}$ steps)

and thus $D^5 f_0 \approx \dfrac{f_5 - 5f_3 + 10f_1 - 10f_{-1} + 5f_{-3} - f_{-5}}{32h^5}$ (for full steps)

etc.

Ex All this is best illustrated by example.

Let's consider the problem ① on page 27,

$$y'' = y \qquad y(0) = 0 \qquad y(1) = 1.$$

A finite-difference approximation yields:

$$\frac{y_{n+1} - 2y_n + y_{n-1}}{h^2} = y_n \qquad \text{for } n = 1, \ldots, N-1$$

In other words,

$$-y_{n-1} + (2+h^2)y_n - y_{n+1} = 0 \quad , \quad n = 1, \ldots, N-1$$

$$y_0 = 0$$
$$y_N = 1$$

In matrix form:

$$\begin{bmatrix} 1 & 0 & & & & \\ -1 & (2+h^2) & -1 & & \bigcirc & \\ & -1 & (2+h^2) & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & \bigcirc & & -1 & (2+h^2) & -1 \\ & & & & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \\ y_N \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

Because the boundary conditions are so simple, we see that $y_0 = 0$, $y_N = 1$, so this system can be pre-simplified to

$$\begin{bmatrix} (2+h^2) & -1 & & & 0 \\ -1 & (2+h^2) & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & & & -1 \\ 0 & & & -1 & (2+h^2) \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ \\ y_{N-1} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ \\ 1 \end{bmatrix}$$

If we use $h = 0.1$, then we get the following 9x9 system

$$\begin{bmatrix} 2.01 & -1 & & & & & & & \\ -1 & 2.01 & -1 & & & & & 0 & \\ & -1 & 2.01 & -1 & & & & & \\ & & -1 & 2.01 & -1 & & & & \\ & & & -1 & 2.01 & -1 & & & \\ & & & & -1 & 2.01 & -1 & & \\ & 0 & & & & -1 & 2.01 & -1 & \\ & & & & & & -1 & 2.01 & -1 \\ & & & & & & & -1 & 2.01 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Let us solve this system using, say, Gaussian Elimination and compare the results to truth.

Note by the way that an exact solution to our differential equation is $y(x) = \dfrac{e^x - e^{-x}}{e - \frac{1}{e}} = \dfrac{\sinh x}{\sinh 1}$.

		exact solution
x_n	y_n	$y(x_n)$
0	0	0
0.1	0.08524	0.08523
0.2	0.17134	0.17132
0.3	0.25915	0.25912
0.4	0.34955	0.34952
0.5	0.44345	0.44341
0.6	0.54178	0.54174
0.7	0.64553	0.64549
0.8	0.75574	0.75571
0.9	0.87350	0.87348
1.0	1.0	1.0

The maximum error is about $5 \cdot 10^{-5}$.

Relaxation

We have seen how to construct a linear system $Ay=b$ from a linear boundary value problem. We can of course solve $Ay=b$ in any way we like. Sometimes, rather than use a direct method, we may wish to "relax" to a solution.

The basic idea is to start with a guess to the solution of $Ay=b$, then use each of the constraint equations to update this guess.

For instance, for 2nd order boundary value problems, A is a tridiagonal matrix. A typical interior row of $Ay=b$ represents the equation

$$a_{n-1}y_{n-1} + d_n y_n + c_{n+1}y_{n+1} = b_n$$

where

$$A = \begin{pmatrix} d_1 & c_2 & \ddots & & 0 \\ a_1 & \ddots & \ddots & \ddots & \\ & \ddots & \ddots & \ddots & c_m \\ 0 & & \ddots & a_{m-1} & d_m \end{pmatrix} \quad \& \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

we can rewrite this as

$$(*) \qquad y_n = \frac{b_n - a_{n-1}y_{n-1} - c_{n+1}y_{n+1}}{d_n}$$

[m is probably $N-1$]

(Recall, if h is small enough, the d_i are non-zero.)

Here is the relaxation procedure

Let's denote by $y_n^{(i)}$ the approximate solution to y_n that we have obtained at the ith relaxation step.

So $(y_0^{(0)}, y_1^{(0)}, \dots, y_N^{(0)})$ constitutes our initial guess.

We update guesses using the formula (*) on p. 33. There are two common updating methods:

Jacobi Iteration

In this method one updates values at the $i+1$st stage using only values from the ith stage.

So $$y_n^{(i+1)} = \frac{b_n - a_{n-1}y_{n-1}^{(i)} - c_{n+1}y_{n+1}^{(i)}}{d_n}, \quad n=1,2,\dots$$

Gauss-Seider Iteration

In this method one updates values at the $i+1$st stage using the most recently available values, independent of whether they were obtained in the ith stage or $i+1$st stage.

So $$y_n^{(i+1)} = \frac{b_n - a_{n-1}y_{n-1}^{(i+1)} - c_{n+1}y_{n+1}^{(i)}}{d_n}$$
$n=1,2,\dots$

Notice that we use $y_{n-1}^{(i+1)}$.

Starts to be practical

Successive Overrelaxation

$$y_n^{(i+1)} = \underset{\uparrow \text{ note}}{(1-\omega)} y_n^{(i)} + \underset{\uparrow \text{ overrelaxation parameter}}{\omega} \frac{b_n - a_{n-1} y_{n-1}^{(i+1)} - c_{n+1} y_{n+1}^{(i)}}{d_n} \qquad i = 1, 2, \ldots$$

This can be shown to converge for $0 < \omega < 2$. To get overrelaxation choose $\omega > 1$, i.e., overshoot the true correction term.

There are ways of choosing ω nicely, related to the spectral radius of the Jacobi (or Gauss-Seidel) method.

NRiC says to use $\omega = \dfrac{2}{1 + \sqrt{1 - \rho_J^2}}$.

$\uparrow \rho_J$ = spectral radius of Jacobi method

$(\rho_{G.S.} = \rho_J^2)$

This tends to faster convergence than G.S. (for $1 < \omega < 2$)

Where does all this come from?

Related to iterative methods for solving linear equations.

Suppose we start with a system $Ax = b$.
We decide this is too difficult to solve and instead look at

$$Mx = (M-A)x + b \qquad \text{for some } M.$$

Iterative methods solve the system

$$Mx^{(i+1)} = (M-A)x^{(i)} + b.$$

Choosing M suitably is key.

- $\underline{M=A}$ gives us convergence in one step, but then we are doing the work we wanted to avoid.
- $M = I$ gives us fixed point iteration

$$x^{(i+1)} = Bx^{(i)} + b \quad , \quad B = I - A$$

If $\|B\| < 1$ then this is a contraction mapping, and thus converges

- If we write $A = L + D + U$
then we can find Jacobi, G.S., & SOR.

- **M = D yields Jacobi:**

$$D x^{(i+1)} = -(L + U) x^{(i)} + b.$$

- **M = L + D yields Gauss-Seidel:**

$$(L+D) x^{(i+1)} = -U x^{(i)} + b.$$

- **For SOR, it is useful to rewrite $Ax = b$:**

$$\omega A x = \omega b \qquad (\text{assume } \omega \neq 0; \text{ generally } 1 < \omega < 2)$$

Now work with ωA and use $M = D + \omega L$ for that. This gives the iteration

$$(D + \omega L) x^{(i+1)} = -(\omega U + (\omega - 1) D) x^{(i)} + \omega b.$$

For each coordinate x_k we therefore have

$$x_k^{(i+1)} = (1-\omega) x_k^{(i)} - \omega \frac{\sum_{j=1}^{k-1} a_{kj} x_j^{(i+1)} + \sum_{j=k+1}^{N} a_{kj} x_j^{(i)} - b_k}{a_{kk}}$$

(assuming A is $N \times N$).

Note that we can think of that as

$$x^{(i+1)} = (1-\omega) x^{(i)} + \omega x_{GS}^{(i+1)}$$

vector we would get from $x^{(i)}$ using Gauss-Seidel.

Important questions:

When does each method converge?
How fast?

The answers are related to the eigenvalues of $M^{-1}(M-A)$.

Suppose we define the error term $e^{(i)} = x - x^{(i)}$

Then $Me^{(i+1)} = (M-A)e^{(i)}$

so $e^{(i+1)} = Be^{(i)}$ with $B = M^{-1}(M-A)$
$= I - M^{-1}A$

(Aside: This suggests the closer to A M is the better (of course).

Thus $e^{(i)} = B^i e^{(0)}$

So $e^{(i)} \to \underline{0}$ iff $B^i \to 0$
iff every eigenvalue λ of B satisfies $|\lambda| < 1$.

The rate of convergence is governed by the largest $|\lambda|$, called the <u>spectral radius</u> of B.

<u>Note:</u> If $\sum_{j=1}^{n} |b_{ij}| < 1$ then all eigenvalues satisfy $|\lambda| < 1$.

<u>Note:</u> If the original matrix A is diagonally dominant then this implies convergence of the Jacobi method for many of our 2nd order bndry value problems.

Comments

Why use relaxation?

Easy to implement.
Can handle complex boundary conditions.
Convergence, when it exists, tends to be global.

What is bad about relaxation?

Convergence may be very slow.
Difficulty handling differential equations whose solutions are highly oscillatory.

Does the method always converge?

No, but there is an eigenvalue test for convergence.
If this test is positive, then convergence is global.
For 2nd order boundary value problems, if h is small enough, then convergence is global.

[Aside: Relaxation is a form of fixed-point iteration:

$$y^{(i+1)} = By^{(i)} + b'$$

where B is a matrix determined by A and the type of relaxation.
The rate of convergence is governed by the largest eigenvalue magnitude, $|\lambda|$, of . If $|\lambda| < 1$ then convergence is global.]

Omitted Topics

We have left out some important topics whose existence you should know.

- ### Shooting Methods

The basic idea is this: Instead of tying a general solution to a boundary value problem down at both points, one only ties it down at one end. This leaves a free parameter (in the case of a 2nd order problem). For a given value of this free parameter one then integrates out a solution to the differential equation. Specifically, you start at the tied-down boundary point and integrate out just like an initial value problem. When you get to the other boundary point, the error between your solution and the true boundary condition tells you how to adjust the free parameter. You repeat this process until you obtain a solution that matches the second boundary condition.

Advantages:
- Convergence tends to be very quick.
- The method also works for non-linear diff. eq.

Disadvantage:
- Convergence isn't guaranteed.
  (Secant method in (free parameter, y_{bondry}) space)

- Collocation Methods

Write the solution formally as a linear combination of N orthogonal functions, each of which satisfies the boundary conditions. The objective is to find the coefficients in this sum. One does this by forcing the solution to satisfy the differential equation at the N-2 interior points and to satisfy the two boundary conditions (in the case of a 2nd order problem).

The result is an approximate solution to the differential equation at non-mesh points, and an exact solution at mesh points.

- Perturbation Methods

Useful for approximating solutions to non-linear equations.