ODEs (as of 07dec00)

Here are some comments concerning slight differences in the treatment in class compared to what the book does.

1. Loosely speaking, an *ordinary differential equation* (*ODE*) is an equation that involves one or more derivatives of an unknown function y. However, that is a bit misleading. The following equations all fit this definition:

$$y'(y(t)) = 1$$

$$y'(t) = \int_0^1 y(t-s)g(s)ds$$

$$y'(t) = f(t, y(t-10))$$

but are not considered to be ODEs. So, more precisely, an **ordinary differential equation** (**ODE**) is a relationship

$$g(t, y(t), y'(t), \ldots, y^{(n)}(t)) = 0$$

between the values of a function and that of one or more of its derivatives. The **order** of the differential equation is that of the highest derivative explicitly appearing in it. Usually, it is possible to solve the equation explicitly for that highest derivative, i.e., write the equation in the form

$$y^{(n)}(t) = f(t, y(t), y'(t), \ldots, y^{(n-1)}(t)), \tag{1}$$

with $f(t, z_0, z_1, \ldots, z_{n-1})$ some scalar-valued function of its $n+1$ arguments t, $z_0, \ldots, z_{n-1}$.

2. The book rightfully stresses the study of *first-order* ODEs, i.e., differential equations of the form

$$y'(t) = f(t, y(t)), \tag{2}$$

with $f(t, z)$ some known function of its two arguments, t and z. Such an equation usually has many solutions. A particular one is selected by prescribing its value at a certain point. By tradition, this is the left endpoint of the interval $[a \,..\, b]$ on which we seek the function y. Therefore, the prescribed value of $y(a)$ is called an **initial condition**.

Standard software packages usually provide for the solution of first-order ODEs only. E.g., `matlab`'s `ode23` and `ode45` both solve first-order ODEs. On the other hand, such software can solve the first-order ODE (2) even when the solution (and, correspondingly, the function f) is *vector-valued*. On ODE for a vector-valued function is also called a *system of ODEs*. For this reason, the nth order ODE (1) is usually converted to an equivalent first-order system

$$Z'(t) = F(t, Z(t))$$

for the vector-valued function $Z(t) := (Z_1(t), \ldots, Z_n(t)) := ((y(t), y'(t), y''(t), \ldots, y^{(n-1)}(t))$, with the vector-valued function $F := (F_1, F_2, \ldots, F_n)$ given by

$$F_i(t, Z(t)) := \begin{cases} Z_{i+1}(t) & i < n; \\ f(t, Z(t)) & i = n. \end{cases}$$

See the Kepler example in the book (page 343), where this same idea is applied to convert a second-order ODE for a function with two components to a first-order ODE for a function with four components.

3. It always pays to consider very simple examples. A particularly simple example occurs when f doesn't really depend on its second argument, i.e., when

$$f(t,z) = g(t)$$

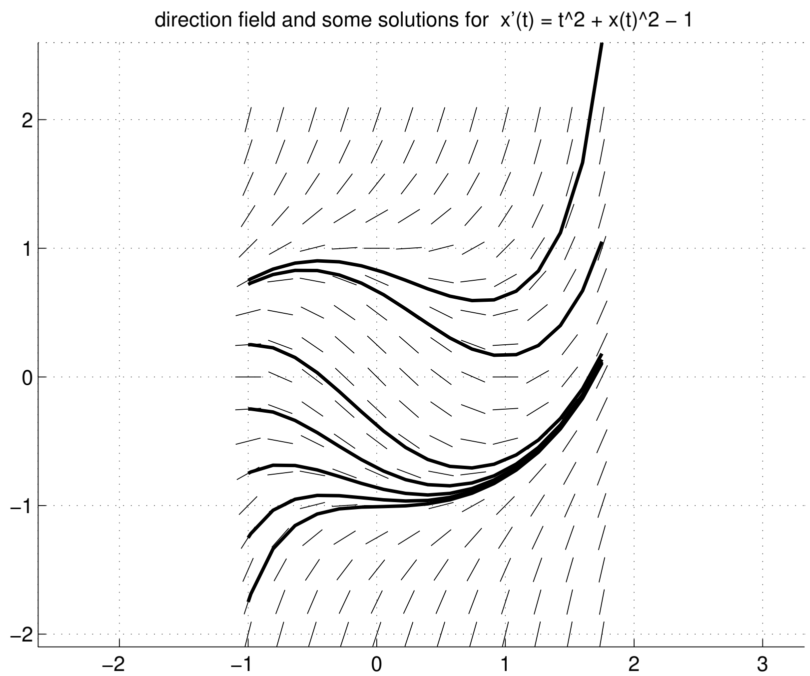
for some function g. In this case, we can write down the solution immediately:

$$y(t) = y(a) + \int_a^t g(s)\,\mathrm{d}s.$$

Of course, if we really look at this, we haven't accomplished much unless we have some way of carrying out the integration. Still, it permits you in this case to compare the numerical methods to be discussed with the quadrature methods discussed earlier in this course.

4. The book fails entirely to stress the geometric picture of the **direction field** associated with the first-order ODE (2). Here is such a picture, for the interval $[-1\,..\,1.75]$ and for the specific ODE

$$y'(t) = t^2 + (y(t))^2 - 1.$$



The figure shows, at various points in the (t,z)-plane, a little line segment whose slope equals $f(t,z)$. If (t,z) is one of these points, and the solution of interest to us has the value z at t, i.e., $y(t) = z$, then its graph would go through the point (t,z) with slope exactly that of the little line segment plotted there.

In particular, the solution will start out at $t = a$ with slope $y'(a) = f(a, y(a))$. The figure shows several such solutions, all starting at $a = -1$, but with different initial values. With the direction field plotted, we can get a rough idea of what the solution will do farther to the right. For example, it looks as if any solution starting with $y(-1) > 1$ is sure to grow very rapidly as t grows. Solutions y with $y(-1)$ near zero quickly reach an area in which all directions point downward and, correspondingly, they all decrease as t increases, until they pass through this area of negative slope, after which they all grow.

There are more subtle things to see here for which we actually need to compute the solutions, but which should be pointed out: The initial values used here are, in fact,

$$-1.75,\ -1.25,\ -.75,\ -.25,\ .25,\ .71878,\ .75.$$

For all but the two uppermost of these, the solution seems to come together eventually. The two uppermost solutions, on the other hand, start out very close, but quickly move away from each other and from the rest. Think of the implication for numerical calculations: in an extreme case, by just rounding the initial value, you may end up with a totally different solution. This is an example of **instability** (see later).

Taylor-Series method Since we know that

$$y'(t) = f(t, y(t)),$$

we know $y'(a)$ since we are given $y(a)$, hence can compute $y'(a)$ by evaluating $f(t, z)$ at $(a, y(a))$.

Then, with both $y(a)$ and $y'(a)$ now known, we can differentiate the known expression $f(t, y(t))$ for $y(t)$ with respect to t and so obtain a formula for $y''(t)$ in terms of t, $y(t)$ and $y'(t)$, namely the formula

$$y''(t) = f_t + f_z \cdot y',$$

in which f_t and f_z are the abbreviations

$$f_t := (D_1 f)(t, y(t)), \quad f_z := (D_2 f)(t, y(t)),$$

and $D_j f$ is the partial derivative of f with respect to its jth argument, $j = 1, 2$.

For the example of the figure, we had

$$y'(t) = t^2 + (y(t))^2 - 1,$$

hence

$$y''(t) = 2t + 2y(t)y'(t),$$

therefore also

$$y'''(t) = 2 + 2(y'(t))^2 + 2y(t)y''(t),$$

etc.

In this way, we can, in principle, compute as many terms of the Taylor series

$$y(a + h) = y(a) + y'(a)h + y''(a)h^2/2 + \cdots$$

for y at a as we care to, thus obtaining more and more accurate estimates for $y(a+h)$. This is the Taylor series method.

This method looks a bit forbidding, if the repeated differentiations of $f(t, y(t))$ are written out formally, in terms of the various partial derivatives of various orders of the function f.

However, things look less forbidding if you think of the method in the following terms: Assume that you have written a subroutine for the evaluation of $f(t, y(t))$ for given t and $y(t)$. Then, this would be a good occasion to apply what you have learned about *differentiating a program* earlier. Differentiate this program with respect to t as many times as required, and so obtain a program that computes, for given $(t, y(t))$, not only $y'(t)$, but also at the same time $y''(t)$, $y'''(t)$, etc.

Or, as done in class, if the 'slope' function $f = f(t, z)$ is simple, then it is easy to use `Matlab`'s symbolic toolbox command `diff` to provide an m-file `dtftz(f,zprime)` like the following:

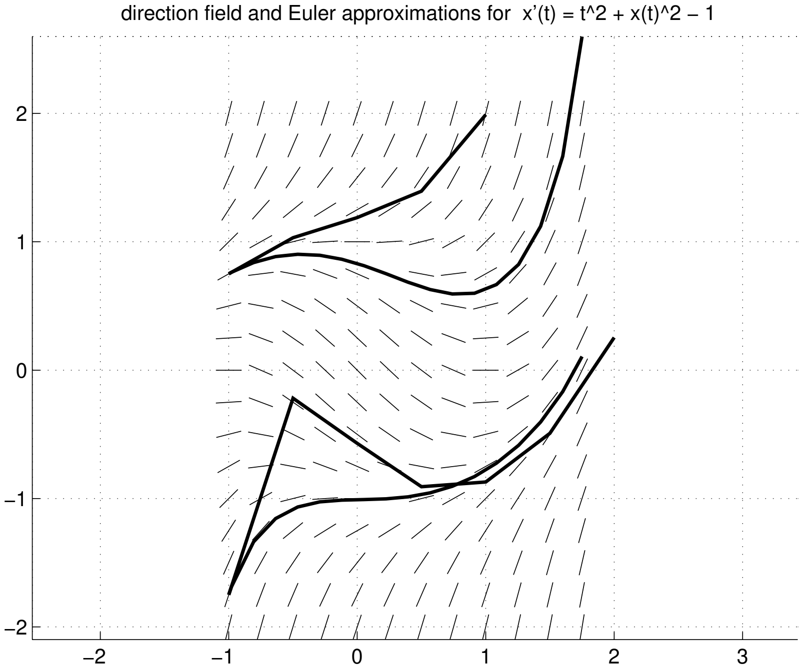
```
function deriv = dtftz(f,zprime)
%DTFTZ total t derivative of  f(t,z)  for given dz/dt
%
%        deriv = dtftz(f,zprime)
%
% returns the string containing the total derivative
%
%        deriv = D_t f(t,z(t)) + z' D_z f(t,z)
%
% wrto  t  of the function f(t,z) described by the string  f ,
% using z'(t) as specified by the string  zprime .
deriv = char(simplify(sym( [ ...
       char(diff(f,'t')), '+(', zprime, ')*(', char(diff(f,'z')), ')'...
                            ])));
```

that inputs the string `f` that describes f in terms of t and z, and a string `zprime` describing z', and outputs a string describing the total derivative $f_t + f_z z'$ of $t \mapsto f(t, z(t))$ with respect to t. E.g., the command `inline(dtftz(f,f),'t','z')` would provide the function $t \mapsto y''(t)$ as a function of t and $z = y(t)$.

Euler's method is, in a way, the simplest Taylor-Series method: one stops with the first term:

$$y(a+h) \sim y(a) + hf(a, y(a)) \;\; =: y_{a+h}.$$

For most examples, this will only work if h is very small, hence one simply repeats this procedure at $t = a + h$, $t = a + 2h$, etc. Geometrically, we can think of it as following that slope segment for a little bit, computing the required slope at the point we reach, going in that direction for a little bit, etc., as indicated for our earlier example for two of the earlier solutions shown, and for a stepsize of $h = 1/2$, in the following figure.



For the upper solution, the last two steps of Euler's method aren't even shown since it has a value greater than 12 by the time it reaches $t = 2$. For the lower solution, Euler's method also initially grows much faster, but is then pushed back by those negative slopes and ultimately seems to parallel the correct solution, at a slightly lower level.

Runge-Kutta methods The basic idea is simple. Instead of using the right side just once, as in Euler's method, where

$$y(t+h) \sim y(t) + hf(t, y(t)),$$

why not sample the function $f(t,z)$ at several points near the point $(t, y(t))$, i.e., get numbers $k_j := hf(t + \alpha_j h, y + hz_j)$ and use a weighted sum

$$y(t+h) \sim y(t) + w_1 k_1 + w_2 k_2 + \cdots.$$

We are free to choose the weights w_j, the α_j, and the z_j. The goal would be to choose them in such a way that the expansion of the right side in powers of h agrees to as many terms as possible with the Taylor series expansion

$$y(t+h) = y(t) + hy'(t) + (h^2/2)y''(t) + \cdots.$$

With r such suitably chosen k_j, we would expect to be able to match all terms up to and including $(h^r/r!)y^{(r)}(t)$, hence making the **Local Truncation Error** or **LTE** be of order $r+1$, i.e., involve h^{r+1}.

The book goes through the manageable example of the second-order Runge-Kutta method, and wisely refrains from deriving the most popular Runge-Kutta method, the fourth order one.

An equivalent but, perhaps, simpler view (that also applies to the discussion of multistep methods later) is to think of such methods in terms of *numerical quadrature*. After all, we know that

$$y(t+h) = y(t) + \int_t^{t+h} f(t, y(t))\,\mathrm{d}t,$$

hence it makes sense to think of our approximation

$$y(t+h) \sim y(t) + w_1 k_1 + w_2 k_2 + \cdots$$

as providing (or being derived from) the approximation

$$w_1 k_1 + w_2 k_2 + \cdots \sim \int_t^{t+h} f(t, y(t))\,\mathrm{d}t,$$

particularly since we are choosing k_j in the form $hf(t+\alpha_j h, y+hz_j)$, with the numbers $f(t+\alpha_j h, y+hz_j)$ supposedly close to values of the exact integrand, $f(t, y(t))$. The only complication is due to the fact that we don't know the integrand exactly, hence have to guess at reasonable values for $y(t+\alpha_j h)$, and do provide these guesses in the form $y_j + hz_j$. With this view, it is not surprising that, for the simplest ODE $y'(t) = g(t)$, the various Runge-Kutta methods reduce to standard quadrature rules; e.g., RK2 becomes the Trapezoidal Rule, and RK4 becomes Simpson's rule.

This view also makes it easy to remember the meaning of the **order** of a numerical method. It works just as with the order of a composite quadrature rule. If the simple rule (here the single step) has (local truncation) error consth^{r+1}, then, at best, the error at the right endpoint b of the interval of $[a \,..\, b]$ of interest will be $(b-a)/h$ times that error, i.e., roughly of the form consth^r, and therefore will be called a **method of order** r. E.g., the local truncation error for RK2 is like that for the trapezoidal rule, i.e., $O(h^3)$, hence the method is called Runge-Kutta *of order 2*.

stability

In the present context, this term (see Section 9.1.2 in the book) refers to the effect of an error in the initial condition.

So, denote by $y(t, s)$ the value at t of the solution of our first-order ODE

$$y'(t) = f(t, y(t))$$

that satisfies the initial condition $y(a) = s$. In an earlier figure, we saw that even very small changes in s can ultimately lead to large differences in the solution, but we also saw that rather large differences in the initial data can, at times, have little effect for the solution at later times.

Specifically, we want to compare $y(t, s)$ with $y(t, s+\sigma)$ for some error or perturbation σ in the initial value. We look at their difference,

$$\eta(t) := y(t, s+\sigma) - y(t, s)$$

as a function of t. On differentiating both sides, we find that

$$\eta'(t) = y'(t, s+\sigma) - y'(t, s) = f(t, y(t, s+\sigma)) - f(t, y(t, s)),$$

using the fact that both $y(\cdot, s)$ and $y(\cdot, s+\sigma)$ satisfy our differential equation. Now, assuming that f is smooth, we can apply Rolle's Theorem that $g(a) - g(b) = g'(\xi)(a-b)$ for some ξ between a and b to the function $g(z) = f(t, z)$ with $a = y(t, s+\sigma)$, $b = y(t, s)$, to conclude that

$$\eta'(t) = f_z(t, \xi_t)(y(t, s+\sigma) - y(t, s)),$$

i.e.,

$$\eta'(t) = f_z(t, \xi_t)\,\eta(t),$$

for some ξ_t between $y(t, s)$ and $y(t, s+\sigma)$.

This simple formula tells us a lot, as follows: If $f_z(t, \xi_t)$ is positive, then a positive $y(t)$ will grow more positive, while a negative $y(t)$ will grow more negative. In other words, *if $f_z(t, \xi_t)$ is positive, then the gap between $y(t, s)$ and $y(t, s+\sigma)$ will be* **amplified** *or widened, i.e., will be made worse.*

On the other hand, if $f_z(t, \xi_t)$ is negative, then a positive $y(t)$ will grow less positive, while a negative $y(t)$ will grow less negative. In other words, *if $f_z(t, \xi_t)$ is negative, then the gap between $y(t, s)$ and $y(t, s+\sigma)$ will be* **attenuated** *or lessened, i.e., will be made better.*

This is very nicely illustrated in our earlier example. There, we had

$$f(t, z) = t^2 + z^2 - 1,$$

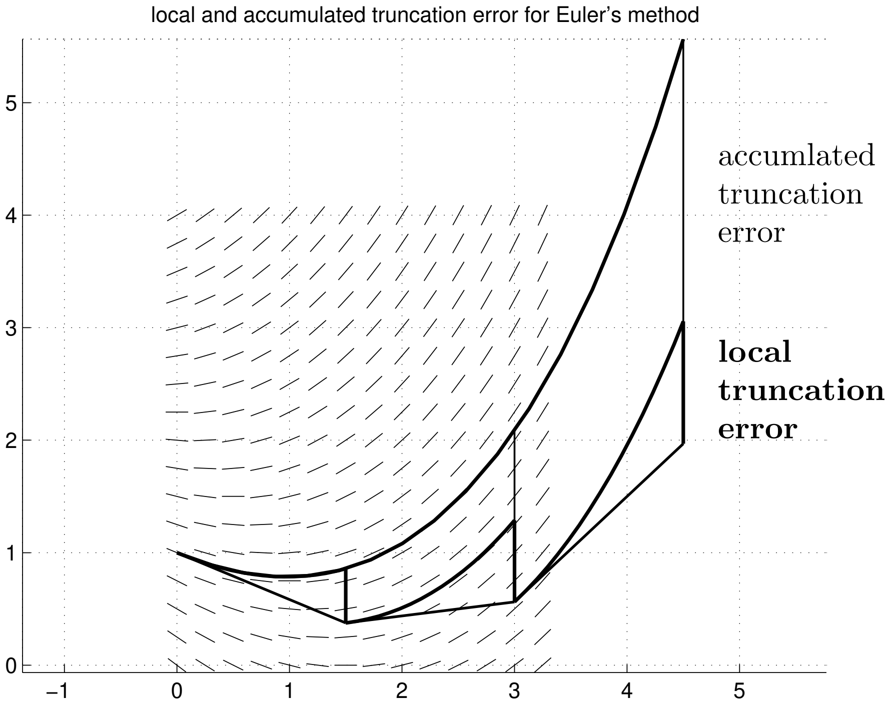
hence

$$f_z(t, z) = 2z.$$

Thus, $f_z(t, z)$ is positive or negative depending on whether z is positive or negative. Indeed, we notice in the figure that any two solutions both above the t-axis pull apart as t increases while, any two solutions below the t-axis draw together as t increases, and both the amplification above and the attenuation below can be extreme, even for this simple and mild example.

error

As the next figure shows, after several steps of a numerical scheme, such as Euler or Runge-Kutta, the error consists of two parts, the **local truncation error** and the **accumulated truncation error**. In finite-precision arithmetic, there is, in addition, the local and the accumulated roundoff error. Once we decide with what precision to calculate, then the only part of this error we can control is the local truncation error (at every step), and we control this by controlling the step size.



The details of such a **step-size control** can be quite intricate, but the basic idea is simple: after taking a step of size h, compare an estimate for the local truncation error with a specified tolerance; if the tolerance is smaller, shorten (usually, halve) the step size and try again; if the tolerance is slightly larger, accept and do the next step with the same h; if the tolerance is significantly larger, accept and do the next step with a larger (usually, doubled) h.

The simplest way to get an **estimate for the local truncation error** is to do also two steps with half the stepsize and compare. This is a bit expensive compared to the available alternative. Also, while the local truncation error behaves like consth^r for some r as $h \to 0$, we are usually working with values of h for which this is not yet true.

The standard alternative is something like Runge-Kutta-Fehlberg, – not explicitly mentioned in the book but used in `ode23` and `ode45`, hence, e.g., the `45`, to indicate that a 4th order RK along with a related 5th order formula developed by Fehlberg, is used. In this approach, the error in the given method is estimated by comparing it with a higher-order result. This assumes that the current step is sufficiently small, since otherwise there is no reason to believe that the higher-order method is more accurate.

Here is a simple numerical example. Taking the earlier sample equation with

$$f(t, z) = t^2 + z^2 - 1,$$

and starting at $a = -1$ with $y(a) = 3/4$, and using $h = 1$, one step of Euler's method gives the approximation:

$$y^E(a + h) = y^E(0) = 3/4 + F_1,$$

with

$$F_1 = 1 \cdot ((-1)^2 + (3/4)^2 - 1) = (3/4)^2,$$

hence

$$y^E(a+h) = y^E(0) = 3/4 + F_1 = 21/16 = 1.3125.$$

From this, 2nd order Runge-Kutta computes

$$F_2 = hf(a+h, y^E(a+h)) = 1\cdot((0)^2 + (21/16)^2 - 1) = (21/16)^2 - 1 = .72265625,$$

hence

$$y^{RK}(a+h) = 3/4 + (F_1+F_2)/2 = 1.3925781,$$

giving an estimate of the error in the Euler approximation of ~ 0.08. In fact, from the earlier figure, $y(0) \sim .8$, so the error in the Euler approximation is much larger than our estimate. Of course, the reason for this is that the specific h used here was not 'small enough'. A look at the earlier figure confirms this: The direction field at the end of the Euler step is very different from that near the beginning of the step.

This illustrates one of the difficulties of local step size control.

One additional issue has to be faced: is one looking for a small absolute error, or for several correct digits, i.e., a small relative error, or for a combination of the two?

Another difficulty is the possible instability of the differential equation itself, i.e., the fact that, even though the local truncation error is carefully controlled, the accumulated error may grow or, worse yet, we may have no idea about it. Of course, one can monitor the size of $f_z(t, y(t))$ and more sophisticated ODE solvers will do that.

However, the only defense against this is to run one's program with several tolerances and compare the results.

higher order ODEs and/or systems

Since, by now, the available software for solving first-order initial value problems (**IVP**s) has become quite sophisticated, the standard approach to solving higher-order ODEs and/or systems of ODEs is to convert any such to an equivalent first-order *system*.

In this conversion, it pays to make a dictionary that shows how the original dependent variables and their derivatives are related to the entries of the new vector-valued dependent variable.

It is also possible to introduce the independent variable as the zeroth entry of the solution vector, thereby making the system formally autonomous (i.e., the right side doesn't depend explicitly on the independent variable any more). This makes the discretization process neater.

use of Hermite interpolation

The methods for solving ODEs discussed so far (i.e., one-step methods) as well as the methods only touched on in class (the multistep-methods, see Section 9.3 and its subsections) provide the value of the solution at a certain set of points $t_0 = a < t_1 = a+h_1 < t_2 = t_1 + h_2 < \cdots < t_n = b$, the meshpoints used. For values at other points, local polynomial interpolation is very convenient. Since we also know the first derivative of the solution at these meshpoints, Hermite interpolation is particularly useful here. No doubt you remember very well how to construct, e.g., the piecewise cubic Hermite interpolant. (We match the computed values $y(t_i)$ and $y(t_{i+1})$ as well as the numbers $y'(t_j) = f(t_j, y(t_j))$, $j = i, i+1$ by doing cubic interpolation at the points $t_i, t_i, t_{i+1}, t_{i+1}$. For this, we generate the divided difference table as discussed earlier in the course, using the fact that $y[t_j, t_j] = y'(t_j)$, $j = i, i+1$.)

multistep methods

We briefly discussed **multistep methods** in which one uses the approximation

$$y(t+h) \sim y(t) + \int_t^{t+h} p_{k-1}(t)\,\mathrm{d}t,$$

with p_{k-1} the polynomial approximation to $y'(t) = f(t, y(t))$ obtained by matching at certain k nearby t_j the (approximate) value $f(t_j, y_j) \sim y'(t_j)$.

The standard methods come in two flavors:

(i) k-th order **Adams-Bashforth** or **explicit** methods, in which the calculation of y_{n+1} is based on (t_j, y_j) for $j = n-k+1{:}n$ for some k; and

(ii) kth order **Adams-Moulton** or **implicit** methods, in which the calculation of y_{n+1} is based on (t_j, y_j) for $j = n-k+2{:}n+1$ for some k, i.e., includes in the construction of p_{k-1} the very information (t_{n+1}, y_{n+1}) to be determined; hence the term ‘implicit’.

The two kinds are usually used in tandem, using first the explicit method to *predict* y_{n+1}, and then follow it up with the implicit method to *correct* that value, and using the difference between the predicted and corrected value in the stepsize control.

BVPs

We only discussed briefly a particular example, that of a two-point Boundary Value Problem for a second-order ODE, solving it approximately by finite differences (see page 255-256), and gave one example in which we used *Shooting* to solve such a problem with the aid of an IVP solver.