12.4. **Estimation of local error.** In practice, we not only want to produce an approximation to the solution at each step of the algorithm, we also want to produce an estimate of the local error. If this error is too big, we will reduce the step size to reduce the local error.

To proceed further, we need a more precise definition of local error. Consider the initial value problem:

$$u' = f(x,u), \qquad u(x_n) = y_n.$$

This is the same differential equation, but the initial condition is now given at x_n and the value is y_n, the approximate solution at that point.

The global error at x_{n+1} is $y(x_{n+1}) - y_{n+1}$. We define the local error at x_{n+1} to be $u(x_{n+1}) - y_{n+1}$. We note that the local error would equal the global error if all previous computations were exact, i.e., if $y_n = y(x_n)$, then $u(x_{n+1}) = y(x_{n+1})$.

To obtain a formula for the local error for the Taylor algorithm of order k, we expand $u(x_{n+1})$ in a Taylor series at the point x_n. Then

$$\begin{aligned} u(x_{n+1}) &= u(x_n) + hu'(x_n) + \cdots + \frac{h^k}{k!}u^{(k)}(x_n) + \frac{h^{k+1}}{(k+1)!}u^{(k+1)}(\xi) \\ &= y_n + hf(x_n,y_n) + \cdots + \frac{h^k}{k!}f^{(k-1)}(x_n,y_n)) + \frac{h^{k+1}}{(k+1)!}u^{(k+1)}(\xi) \\ &= y_{n+1} + \frac{h^{k+1}}{(k+1)!}u^{(k+1)}(\xi). \end{aligned}$$

Hence the local error

$$u(x_{n+1}) - y_{n+1} = \frac{h^{k+1}}{(k+1)!}u^{(k+1)}(\xi) = \frac{h^{k+1}}{(k+1)!}f^{(k)}(\xi, u(\xi)),$$

where $x_n \le \xi \le x_{n+1}$. Thus we see that the local error can be thought of as the local truncation error for u instead of for y. This also holds true for Runge-Kutta methods.

One can prove a theorem showing that one can control the global error by contolling the local error, but the bound is quite pessimistic and not so useful in practice.

We can estimate the local error using a technique similar to the one we used to estimate the local error in numerical quadrature, i.e., by using two methods of different orders of accuracy to compute approximate solutions and then using their difference as an estimate of the local error.

More specifically, consider two one-step methods defined by

$$y_{n+1} = y_n + h\Phi(x_n,y_n), \qquad \tilde{y}_{n+1} = y_n + h\tilde{\Phi}(x_n,y_n),$$

and suppose the local error for the first method is $\mu_n = O(h^p)$ and the local error for the second method is $\tilde{\mu}_n = O(h^q)$, where $q > p$. For example, if the first method is Euler's method and the second method is the simplified Runge-Kutta method of order 2, then $\mu_n = O(h^2)$ and $\tilde{\mu}_n = O(h^3)$.

Hence,

$$\mu_{n+1} = u(x_{n+1}) - y_{n+1} = [u(x_{n+1}) - \tilde{y}_{n+1}] + [\tilde{y}_{n+1} - y_{n+1}].$$

Now $u(x_{n+1}) - \tilde{y}_{n+1} = O(h^q)$, while $\mu_{n+1} = O(h^p)$, where $q > p$. Hence, we expect $\tilde{y}_{n+1} - y_{n+1}$ to be of size $O(h^p)$ and thus be the dominant part of the local error. Thus, we estimate μ_{n+1} by $\tilde{y}_{n+1} - y_{n+1}$.

As in numerical quadrature, we look for pairs of formulas that use many of the same function evaluations, so that the extra work done to estimate the local error is minimized. The Runge-Kutta-Fehlberg method uses a Runge-Kutta method of order 5 (but involving 6 function evaluations per step) to estimate local errors for a 4th order scheme using a subset of these function evaluations. A computer code for this algorithm along with two other pairs of formulas can be found at `http://www.netlib.org/ode/rkuite`.

Next, we consider how one can control the local error once we have an estimate for it. The situation is somewhat different than for quadrature, where the approximation on each subinterval is independent of the approximation on other subintervals. In that case, one could use a stategy based on computing approximations on all subintervals, and if a global error tolerance was not met, attempt to reduce the error by subdividing the subinterval for which the error estimate was the largest.

For the initial value problem for ordinary differential equations, we are using a marching procedure in which the value of the approximate solution at one mesh point depends (at least) on the approximation at the previous mesh point.

Thus, trying to use a strategy that involves having an approximation at all mesh points means one would have to recompute the entire approximation if the error tolerance was not met. Thus, we return to an idea that we considered first in the discussion of estimating quadrature errors, and that is trying to control the local error per unit step, i.e., the quantity μ_{n+1}/h_n where $h_n = x_{n+1} - x_n$. Note that if $\mu_{n+1} = O(h^p)$ for $p \geq 2$, then $\mu_{n+1}/h_n = O(h^{p-1})$, and so will decrease as h decreases. Thus, we can control this error by taking a smaller step size.

To decide how much to change the step size, we could employ the following strategy. Suppose that we have an estimate *est* for the local error per unit step (in going from x_n to $x_n + h_n$), i.e., for μ_{n+1}/h_n. From our discussion above, we could take $est = (\tilde{y}_{n+1} - y_{n+1})/h_n$. Next suppose the local error has the form

$$\mu_{n+1}(h) = \tau_n h^p + O(h^{p+1}),$$

where τ_n is a constant. For example, if we consider Euler's method, then $\tau_n = u''(x_n)/2$ and $p = 2$. Suppose that the estimate of the local error per unit step, *est*, satsifies $|est| > \epsilon$, where ϵ is the given error tolerance per unit step. If we instead take a step size of length γh, with $0 < \gamma < 1$, then the local error in going from x_n to $x_n + \gamma h$ is given by

$$\mu_{n+1}(\gamma h) = \tau_n \gamma^p h^p + O([\gamma h]^{p+1}) = \gamma^p \tau_n h^p + O(h^{p+1}).$$

Hence, the local error per unit step over this subinterval is

$$\mu_{n+1}(\gamma h)/(\gamma h) = \tau_n \gamma^{p-1} h^{p-1} + O([\gamma h]^p) = \gamma^{p-1} \tau_n h^{p-1} + O(h^p).$$

We then want to choose γ so that this new error is less than ϵ. Now

$$\gamma^{p-1}\tau_n h^{p-1} + O(h^p) = \gamma^{p-1}\mu_{n+1}(h)/h + O(h^p).$$

Hence an estimate for $\mu_{n+1}(\gamma h)/(\gamma h)$ is $\gamma^{p-1}est$. We could now choose γ so that $|\gamma^{p-1}est| \leq \epsilon$, For example, we could take $\gamma = (\epsilon/|est|)^{1/(p-1)}$.

If $|est| < \epsilon$, we can accept this step, but may want to increase the step size for the next step, since we expect the error is smaller than it needs to be to satisfy the error tolerance. Hence, a typical adaptive algorithm might use a stategy such as the following.

If $|est| > \epsilon$, reject y_{n+1} and recompute it with a new step size. If $|est| \leq \epsilon$ accept y_{n+1} and use a larger step size for the next step. A reasonable choice in either case is to use $\delta(\epsilon/|est|)^{1/(p-1)}h$ as the new step size, where $0 < \delta < 1$ (e.g., $\delta = .8$). The reason for using the factor δ is that rejections are expensive, so we don't want to use a new step size that we think will just satisfy the error tolerance, since our choice is only an estimate of what we think will work.

As a practical matter, most codes add other heuristic rules such as not allowing a step size greater than h_{max} or less than h_{min} and not allowing a step size to increase or decrease too rapidly.