predcor2: Description and Usage

`predcor2` is a MATLAB implementation of a simple second-order predictor-corrector algorithm for approximating the solution of an ODE initial-value problem

$$y' = f(t, y), \quad y(t_0) = y_0.$$

Specifically, `predcor2` implements the second-order implicit Adams–Moulton method (the trapezoidal method) using the second-order explicit Adams–Bashforth method as a predictor. Startup is done using the implicit backward-Euler method, with the explicit forward-Euler method as a predictor. Stepsizes are adaptively determined to maintain a bound on an estimate of the local error.

About the methods: Both the trapezoidal method and the backward-Euler method have the desirable property that their absolute stability regions include the entire left half of the complex plane. The Adams–Moulton methods of order greater than two do not have this property. Indeed, while their regions of absolute stability lie mostly in the left half-plane, they are bounded and rapidly grow smaller as the order increases (see Figure 6.8, page 408 of the text). These properties undesirably restrict allowable stepsizes on some problems.

Features. The following `predcor2` features are typical of those found in more sophisticated codes.

- The lower-order methods in the same method families are used for startup. In this case, these are the forward and backward Euler methods, which are the first-order Adams–Bashforth and Adams–Moulton methods, respectively.
- The local error is estimated using the difference between the predictor step and the final step determined by the corrector iterations.
- Stepsizes are maintained to keep the local error estimate at about the "right" size (small enough for accuracy and large enough for efficiency) and also to maintain adequately fast convergence of the corrector iterations.
- Corrector iteration options include both fixed-point iteration (the default) and a modified Newton's method (also known as a *chord method*). In the latter, the Jacobian $\partial f/\partial y$ is evaluated and factored at some step, after which the factors are used to compute subsequent approximate Newton steps for as long as they are effective.
- A "fresh" Jacobian and its factors are obtained if, at some step, the modified Newton corrector iterates with "stale" factors have not satisfied the convergence criterion after a prescribed number of iterations (currently three), or if the stepsize changes by more than a specified amount (currently 30%) from one integration step to the next.

Limitations. The simplicity of `predcor2` is accompanied by the following limitations.

- The most serious limitation is that the method order is restricted to two. This is likely to result in inefficiency on many problems and failure on many difficult ones.

- There is no capability for taking advantage of Jacobian sparsity. This is a very serious disadvantage on large-scale problems for which the Jacobian is sparse.
- There is no capability for approximating the Jacobian using finite-difference techniques when an analytic routine for evaluating it is not available.
- There is no capability for addressing bad scaling of problem variables.
- The code is without many "bells and whistles" that enhance usage, such as allowing optional specification of of particular output points, printing out statistics and diagnostic data, etc. In particular, it lacks a counterpart of MATLAB's slick `odeset` capability for changing various tolerances and defaults within the code. However, because the code is so simple, most of these things can be easily found and changed by hand if desired.

Usage. The full calling command is

```
[T,Y,nsteps,nfe,nje] = predcor2(odefun,t_int,y_0,itmeth,odejac);
```

Of the five inputs, only the first three are required; the last two are optional. These are explained as follows:

Required arguments:

- `odefun` is a function handle (`@odefun`) for evaluating the right-hand side of the ODE. This should accept `(t,y)` as arguments and return the value of the function.
- `t_int` is a row two-vector `[t_0, t_f]` that specifies the initial and final t-values.
- `y_0` is a column vector specifying the initial solution value.

Optional arguments:

- `itmeth` is a string that specifies the type of corrector iteration used by the code. It must be either `'fixed-point'` or `'Newton'`. Note the single quotes (either right-slanted or straight-up-and-down on your keyboard); these are required to designate the argument as a string. If the `itmeth` argument is not provided, then the code uses fixed-point iteration by default. If the `itmeth` argument is `'Newton'`, then the `odejac` argument must also be provided.
- `odejac` is a function handle (`@odejac`) for evaluating $\partial f/\partial y$, the Jacobian with respect to y of f. This should accept `(t,y)` as arguments and return the value of the Jacobian. This argument is required when the `itmeth` argument is `'Newton'`.

Output:

- `T` is a column vector of length `nsteps + 1`, where `nsteps` is the number of steps taken by the code during the integration. The components of `T` are the t-values determined by the code during the integration. The first component of `T` is `t_0`, and the last is `t_f`.
- `nsteps` is the number of steps taken by the code during the integration.
- `nfe` is the number of function evalutions required by the code during the integration.
- `nje` is the number of Jacobian evalutions required by the code during the integration.

Examples. Suppose you want to solve the famous Allegash oscillator problem (commonly known as the "allegator") over the interval $0 \leq t \leq 10$ with initial value $y(0) = y_0$. Suppose further that you only care about the t-values and associated y-values determined by the code, and that you are content to let it do fixed-point iteration. Then you would first write an M-file called, say, `all_fun.m`, the first line of which would be

```
function fval = all_fun(t,y)
```

Then you would create a column vector `y_0` from the components of y_0 and call `predcor2` as follows:

```
[T,Y] = predcor2(@all_fun,[0 10],y_0);
```

It would also work to type

```
[T,Y] = predcor2(@all_fun,[0 10],y_0, 'fixed-point');
```

However, the `'fixed-point'` argument isn't necessary, since that is the default.

Now suppose you want to use modified Newton iteration and see all of the available output. For this, you would first write an M-file for evaluating the Jacobian, called, say, `all_jac.m` and having first line

```
function Jval = all_jac(t,y)
```

Then you would call `predcor2` as follows:

```
[T,Y,nsteps,nfe,nje] = predcor2(@all_fun,[0 10],y_0,'Newton', @all_jac);
```