PHYS358: Session 04

Ordinary Differential Equations (4): Implicit Integration

So far, all the ODE integration methods we met are *explicit* integrators, i.e. they step forward in time. In other words, the RHS of the ODE system is always determined by the quantities of the previous integration step.

As we have just seen in class, this can fail under specific circumstances. The solution is to use *implicit* integrators, i.e. ODE systems in which (at least formally) the RHS is determined by the value of the current integration step.

The simplest implicit ODE integrator is the backward Euler step, which you will implement in your homework. Here, we test a few fancier algorithms, which are in fact *semi-implicit*, i.e. they base their updates partially on the current integration step.

Download the class assignment package from Sakai. The code pieces you'll find in `class/04/post`.

1. Run `ode_test.py doubleexp rk2`. This is the double exponential coupled ODE we discussed in class. What do you notice about the "result"?
2. Run `ode_test.py doubleexp rk4`. Does the result improve or get worse? Why? You also can try `rk5`, but don't get your hopes up.
3. Try `ode_test.py doubleexp kr4si`. This is a fixed-step implicit integrator of fourth order (Kaps-Rentrop). Take a note of the residual (second plot), and of the number of iterations.
4. Repeat the test with `rb34si`. This is an adaptive-stepsize semi-implicit Rosenbrock method. What does the residual do, and the number of iterations?
5. Finally, try `ode_test.py doubleexp rk45`. Again, take note of the residual, and of the number of iterations.
6. **Summarize your findings.** Which integrator performs best, in your opinion?

Now lets switch to a different ODE system. This is one of the meanest stiff ODE tests. If you're curious, read up on the Enright-Pryce problem. You can find its implementation in `ode_dydx.py` and `ode_jac.py`.

1. Run `ode_test.py enrightpryce rk4`. Don't worry if the result does not look promising.
2. Repeat with `rk45`. Any improvement? Take note of the iterations.
3. Repeat with `rb34si`. You should see a marked decrease in number of iterations used. How does that compare to `doubleexp`?
4. Feel free to test how many iterations you'd need to get this working with `rk4`.