Numerical Methods II, Courant Institute, Spring 2014

http://www.math.nyu.edu/faculty/goodman/teaching/NumMethII2014/index.html

Always check the `classes` message board before doing any work on the assignment.

Assignment 4, due April

Corrections: (none yet)

Runge Kutta methods (named for Karl Runge and his student Martin Kutta) are time stepping methods for solving the initial value problem for ordinary differential equations

$$\dot{x} = f(x) \; .$$

with initial data $x(t_0) = x_0$. There are time steps Δt_n, which do not have to be the same size, and discrete times $t_{n+1} = t_n + \Delta t_n$. The approximate solution at time t_n is $X_n \approx x(t_n)$. The exact increment of x is $\Delta x_n = x(t_{n+1}) - x(t_n)$, so $x(t_n + \Delta t_n) = x(t_n) + \Delta x_n$. The approximate increment is $\Delta X_n = X_{n+1} - X_n$. Runge Kutta methods have the feature that ΔX_n is a function of X_n (and Δt and f), but not on earlier values X_k for $k < n$. Therefore, they are called *one step* methods. We write $\Delta X_n = \Psi(X_n, \Delta t)$. The exact solution "operator" is written $x(t+\Delta t) = x(t) + \Phi(x(t), \Delta t)$. The error after n steps is $e_n = X_n - x(t_n)$. The *one step error* is $R(x, \Delta t) = \Psi(x, \Delta t) - \Phi(x, \Delta t)$. This plays the role of *residual* in Runge Kutta methods.

Taylor series expansion is the main tool used to design Runge Kutta methods. We use index notation and the summation convention. Component i of f is f_i. Indices after a comma represent partial derivatives, such as

$$f_{i,j}(x) = \partial_{x_j} f_i(x) \; , \quad f_{i,jk}(x) = \frac{\partial^2 f_i(x)}{\partial x_j \partial x_k} \; , \;\text{ etc.}$$

The summation convention is that there is implicitly a summation over repeated indices in a product, so

$$f_{i,j} f_j = \sum_{j=1}^{n} \frac{\partial f_i(x)}{\partial x_j} f_j(x) \; , \quad f_{i,jk} f_j f_k = \sum_{j,k} \frac{\partial^2 f_i(x)}{\partial x_j \partial x_k} f_j(x) f_k(x) \; , \text{etc.} \qquad (1) \quad \boxed{\texttt{sc}}$$

1. (*Exact solution, short time*) Show that the exact solution operator Φ has the informal expression

$$\Delta x = \Delta t f + \frac{\Delta t^2}{2} f' f + \frac{\Delta t^3}{6} f'' f^2 + \frac{\Delta t^3}{6} f'^2 f + O\big(\Delta t^4\big) \; . \qquad (2) \quad \boxed{\texttt{es}}$$

More completely, this is $x_i(t + \Delta t) - x_i(t) = \Delta t f_i(x_t) + \cdots$. Show that $f' f$ and $f'' f^2$ are given in index notation by the expressions in ($\overset{\texttt{sc}}{1}$). Find the index expression for $f'^2 f$. Hint: Differentiate $\dot{x}_i = f_i(x)$ to get $\ddot{x}_i = f_{i,j} \dot{x}_j = f_{i,j}(x) f_j(x)$, etc.

2. (*Two stage method*) An explicit two stage Runge Kutta method has the form

$$\begin{aligned} v_1 &= \Delta t f(x) \\ v_2 &= \Delta t f(x + \alpha v_1) \\ \Delta X &= a_1 v_1 + a_2 v_2 \ . \end{aligned}$$

A Taylor expansion gives the informal expression $v_2 = \Delta t f + \Delta t^2 \alpha a_2 f' f + O\big(\Delta t^3\big)$. Find the formal expression for v_2 up to this order. Show that the one step error is $O\big(\Delta t^3\big)$ if the following equations are satisfied

$$a_1 + a_2 = 1 \qquad (3) \quad \boxed{\texttt{fo}}$$

$$\alpha a_2 = \frac{1}{2} \ . \qquad (4) \quad \boxed{\texttt{so}}$$

Find the values of a_1, a_2, and α that correspond to the following examples

(a) (*predictor/corrector midpoint rule*) The *midpoint rule* for integration is

$$\int_t^{t+\Delta t} f(x(s))ds = \Delta t f(x(t + \Delta t/2)) + O\big(\Delta t^3\big) \ .$$

Use the forward Euler first order prediction of the midpoint value: $\widetilde{X} = x + \frac{\Delta t}{2} f(x)$, then use the midpoint approximation to the integral $\Delta X = \Delta t f(\widetilde{X})$.

(b) (*predictor/corrector trapezoid rule, Heun's method*) The *trapezoid rule* for integration is

$$\int_t^{t+\Delta t} f(x(s))ds = \Delta t \left[f(x(t)) + f(x(t + \Delta t/2)) \right] /2 + O\big(\Delta t^3\big) \ .$$

Use a first order prediction $x(t + \Delta t) \approx \widetilde{X} = x + \Delta t f(x)$ in this.

3. A Runge Kutta method with third order one step error is second order accurate. The solution of a linear differential equation system $\dot{x} = Ax$ involves the matrix exponential

$$e^{tA} = I + tA + \frac{t^2}{2} A^2 + \frac{t^3}{6} A^3 + \cdots \ .$$

Show that any two stage second order accurate method applied to a linear ODE has the effect of using this Taylor expansion up to second order:

$$\Delta X = \left(I + \Delta t A + \frac{\Delta t^2}{2} A^2 \right) x \ .$$

Conclude that all second order two stage Runge Kutta methods have the same linear stability diagram.

4. Show that second order two stage Runge Kutta methods are unstable for linear hyperbolic problems for any CFL number.

5. Find the stability limit $\Delta t < \lambda D \Delta x^2$ (find the maximum value of λ) when you apply a two stage second order Runge Kutta method to the diffusion equation $\partial_t u = D\partial_x^2 u$, with ∂_x^2 discretized using the standard three point approximation.

6. (*Runge Kutta methods are a mess*) A three stage Runge Kutta has the form

$$\begin{aligned} v_1 &= \Delta t f(x) \\ v_2 &= \Delta t f(x + \alpha v_1) \\ v_3 &= \Delta t f(x + \beta v_1 + \gamma v_2) \\ \Delta X &= a_1 v_1 + a_2 v_2 + a_3 v_3 \ . \end{aligned}$$

Show (lots of algebra) that this leads to the index expression informally given as

$$\begin{aligned} \Delta X = {} & \Delta t\,(a_1 + a_2 + a_3)\, f & (5) \quad \boxed{\texttt{fo3}} \\ & + \Delta t^2 \left[a_2\alpha + a_3(\beta+\gamma)\right] f' f & (6) \quad \boxed{\texttt{so3}} \\ & + \Delta t^3 \left[\frac{a_2}{2}\alpha^2 + a_3(\beta+\gamma)^2\right] f'' f^2 & (7) \quad \boxed{\texttt{to1}} \\ & + \Delta t^3 a_3 \alpha\gamma f'^2 f & (8) \\ & + O\big(\Delta t^4\big) \ . \end{aligned}$$

Use this to write four equations involving the six coefficients $a_1, a_2, a_3, \alpha, \beta, \gamma$, that make the method third order accurate overall. Find a three stage third order Runge Kutta method with all positive coefficients.

7. Show that any three stage third order Runge Kutta method, when applied to $\dot{x} = Ax$, is equivalent to using the Taylor expansion of the matrix exponential up to and including the terms of order Δt^3. Show that these schemes are stable for hyperbolic problems if the CFL condition $\lambda \leq \lambda_* = \sqrt{3} \approx 1.73$ is satisfied. (*not necessarily an action item*: It is surprising how simple this calculation turns out to be. A similar calculation shows that the four stage fourth order Runge Kutta method is stable for hyperbolic problems with CFL limit $\lambda \leq 2\sqrt{2} \approx 2.83$. Increasing the number of stages from three to four (a 33% increase) increases the maximum step size by 63%.)

$\boxed{\texttt{ut}}$ 8. (*First unit test.* This sequence of steps results in a software package for adaptive solution of ODE systems and a movie of planets moving around a star. The system is build and tested piece by piece, always using infrastructure borrowed from previous programming assignments.) Create a file `RK3.C` that contains a procedure

```
void RK3( double dx[], double x[], double dt, int n,
          double v1[], double v2[], double v3[])
```

This procedure should implement one time step of size Δt your third order three stage Runge Kutta method for a system in $\mathbb{R}^n$. It should put the computed Δx in the array `dx`. It can write in the *scratch* arrays `v1, v2, v3`, which is assumes have been allocated to size n. It should not write into the array `x`. It evaluates $f(y)$, for any $y \in \mathbb{R}^n$ by calling a procedure

```
void f( double xd[], x)
```

The code to implement this f should be in a different file *someName*`.C`. It should know the size of x, so `f` does not need `n` as an argument.

Create a file `OneStepVerify.C` that contains a `int main()` program that tests your `RK3` procedure using standard convergence analysis. It should test whether one step with your procedure `RK3` has one step error $O\big(\Delta t^4\big)$. You can learn about this kind of Taylor series convergence analysis in section 3.3.1 of

`http://www.cs.nyu.edu/courses/spring09/G22.2112-001/book/book.pdf`

Use the test problem

$$\begin{aligned} f_1(x_1,x_2) &= \quad (1+x_1^2+x_2^2)x_2 \\ f_2(x_1,x_2) &= -(1+x_1^2+x_2^2)x_1 \end{aligned}$$

with initial condition $x(0) = (0,1)$. The solution is $x(t) = (\sin(2t), \cos(2t))$. Put the code implementing this f in a file called `f2D.C`. Implement/modify a `Makefile` so that typing `make OneStepVerify` links the object files `OneStepVerify.o` and `f2D.o` and `RK3.o` to create the executable `OneStepVerify`, and then runs `OneStepVerify`. The output should be some numbers getting close to $16 = 2^4$, which indicate that the one step error of your method is $O\big(\Delta t^4\big)$. The code in `Assignment4.tar` gives hints how this and the next parts can be done. When you unpack this tarball and type:
`make nBodyMovie`, you should get output:

```
hello, n bodies.  Interact.
initialize
The time stepper says hello
The time stepper says hello
The time stepper says hello
```

9. (*Next unit test*) Create a file `FixedStepVerify.C` that contains a `int main()` that solves an ODE the ODE with initial data $x(0) = x_0$ up to time T using as many steps of size Δt as necessary. This should consist mainly of a sequence of calls to `RK3(...)`. Do a convergence study on the

ODE and initial data of part $\overset{\texttt{ut}}{(8)}$ up to time $T = 4\pi$, which corresponds to 4 full revolutions. Verify that the result is third order accurate. Add some lines to the `Makefile` so typing `make FixedStepVerify` links the object files `FixedStepVerify.o` and `f2D.o` and `RK3.o` to create the executable `FixedStepVerify`, and then runs `FixedStepVerify`. Add some lines to the `Makefile` so that typing `make UnitTest` does both unit tests. Be careful to get the dependencies right so that if you change, say, `RK3.C`, typing `make UnitTest` will cause it to re-compile and re-link.

10. (*The gravitational n body problem*) Suppose in space there are p co-planar bodies with positions $r_1, \ldots, r_p$, with $r_k \in \mathbb{R}^2$, and masses $m_1, \ldots, m_p$. According to Newton's theory of gravity, body k pulls on body j with a force

$$G m_j m_k \frac{r_k - r_j}{|r_k - r_j|^3} \; .$$

The force on body r_j is in the direction towards r_k with a strength proportional to $|r_k - r_j|^{-2}$, which is the *inverse square* law. The force is proportional to the two masses. From now on, we choose a time scale so that the universal gravitational constant is equal to 1. The motion of the bodies interacting by gravity is given by the equations of motion

$$\ddot{r}_j = A_j(r_1, \ldots, r_p) = \sum_{k \neq j} m_k \frac{r_k - r_j}{|r_k - r_j|^3} \; . \qquad (9) \qquad \boxed{\texttt{em}}$$

The initial value problem is to give $r_j(0)$ and $\dot{r}_j(0)$, then compute the trajectories $r_j(t)$ using the equations of motion $\overset{\texttt{em}}{(9)}$. These can be formulated as a system of $n = 4p$ differential equations

$$\begin{aligned}
x_{4(j-1)+1} &= r_{x,j} \quad \text{(the } x\text{−component of } r_j) \\
x_{4(j-1)+2} &= r_{y,j} \quad \text{(the } y\text{−component of } r_j) \\
x_{4(j-1)+3} &= \dot{r}_{x,j} \quad \text{(the } x\text{−component of } \dot{r}_j) \\
x_{4(j-1)+4} &= \dot{r}_{y,j} \quad \text{(the } y\text{−component of } \dot{r}_j) \; .
\end{aligned}$$

This convention makes $x_1 = r_{1,x}$, and $x_8 = \dot{r}_{2,y}$, etc. The ODE system for $x(t)$ is

$$\begin{aligned}
\frac{d}{dt}\begin{pmatrix} x_{4(j-1)+1} \\ x_{4(j-1)+2} \end{pmatrix} &= \begin{pmatrix} x_{4(j-1)+3} \\ x_{4(j-1)+4} \end{pmatrix} \\
\frac{d}{dt}\begin{pmatrix} x_{4(j-1)+3} \\ x_{4(j-1)+4} \end{pmatrix} &= \begin{pmatrix} A_{x,j}(r_1, \ldots, r_p) \\ A_{y,j}(r_1, \ldots, r_p) \end{pmatrix} \; .
\end{aligned}$$

Create a file `fnb.C` with a version of `f` that implements these ODEs. It should use a fixed small time step and print the trajectories to a file so that a Python program (which you also should write) can read them and make a movie. Start with two bodies, in which case the result should be elliptical motion about the center of mass. Then add more bodies to make more interesting movies. The command `make nBodyMovie` should build the simulator, run the simulator, then run the Python movie maker. Experiment with different numbers of planets, different masses and initial conditions. Email the movie you think is the most interesting.

The file `fnb.C` should have two procedures (at least), one to evaluate $f(x)$, and one to set the initial conditions.

11. (*Adaptive solver*) Omitted. The assignment is long enough already.