Introduction to ODE solvers and their application in OpenFOAM

Gu Zongyuan *Department of Mathematical Science*

March 8, 2009

Contents

1 Introduction

1.1 Methods to solve ODE

The goal of this report is to show some basic information about how to solve ODEs in OpenFOAM and add it into our own case.

For the initial value ODE problem, there are three different kinds of methods in OpenFOAM:

RK: Runge-Kutta

KRR4: Kaps-Rentrop

SIBS: Semi-Implicit Bulirsh-Stoer

Now I will give a simple introduction to the Runge-Kutta method which is most commonly used for solving ODEs. Suppose we have an initial value problem specified as follows:

$$y' = f(t, y), y(t_0) = y_0$$

Basic formula for these method is:

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{1}$$

Then the Runge-Kutta method is given by:

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i$$

Where the h is the time step and the $\sum_{i=1}^{s} b_i k_i$ is an estimated slope:

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + c_2 h, y_n + a_{21} h k_1)$$

$$\vdots$$

$$k_s = f(t_n + c_s h, y_n + a_{s1} h k_1 + a_{s2} h k_2 + \cdots + a_{s,s-1} h k_{s-1})$$

The coefficients can be represented by a tableau, often called the "Butcher tableau", are usually arranged in a mnemonic device.

$$\begin{array}{cccccc}
0 & 0 & 0 & 0 & \cdots & 0 \\
c_2 & a_{21} & 0 & 0 & \cdots & 0 \\
c_3 & a_{31} & a_{32} & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & & \vdots \\
c_s & a_{s1} & a_{s2} & a_{s3} & \cdots & 0 \\
 & b_1 & b_2 & b_3 & \cdots & b_s
\end{array}$$

where 's' means the stage of the method, the Runge-Kutta method is consistent if

$$\sum_{j=1}^{i=1} a_{ij} = c_i \;\; for \;\; i = 2, \cdots, s.$$

'b' is defined by the 'conditions.' Unfortunately I couldn't find how to derive a Butcher tableau exactly but I can show an example: for 4th-order Runge-Kutta with 4 stage, the conditions are:

$$b_1 + b_2 + b_3 + b_4 = 1,$$
$$b_2c_2 + b_3c_3 + b_4c_4 = 1/2,$$
$$b_2c_2^2 + b_3c_3^2 + b_4c_4^2 = 1/3,$$
$$b_3a_{32}c_2 + b_4a_{42}c_2 + b4a_{43}c_3 = 1/6,$$
$$b_2c_2^3 + b_3c_3^3 + b_4c_4^3 = 1/4,$$
$$b_3c_3a_{32}c_2 + b_4c_4a_{42}c_2 + b_4c_4a_{43}c_3 = 1/8,$$
$$b_3a_{32}c_2^2 + b_4a_{42}c_2^2 + b_4a_{43}c_3^2 = 1/12,$$
$$b_4a_{43}a_{32}c_2 = 1/24.$$
$$k_1 = f(t_n, y_n), k_2 = f(t_n + h/2, y_n + hk_1/2) \tag{2}$$
$$k_3 = f(t_n + h/2, y_n + hk_2/2), k_4 = f(t_n + h, y_n + hk3) \tag{3}$$

Like the figure (1), the k_1 in Eqn.(2) determines the slope at the point (t_n, y_n) and k_2, k_2, k_2 determine the slope at the point $(t_n + h/2, y_n + hk_1/2)$, $(t_n + h/2, y_n + hk_2/2)$, $(t_n + h, y_n + hk3)$

Following the RK method we discuss above, we have:

$$y_{n+1} = y_n + h \times (k_1/6 + k_2/3 + k_3/3 + k_4/6)$$

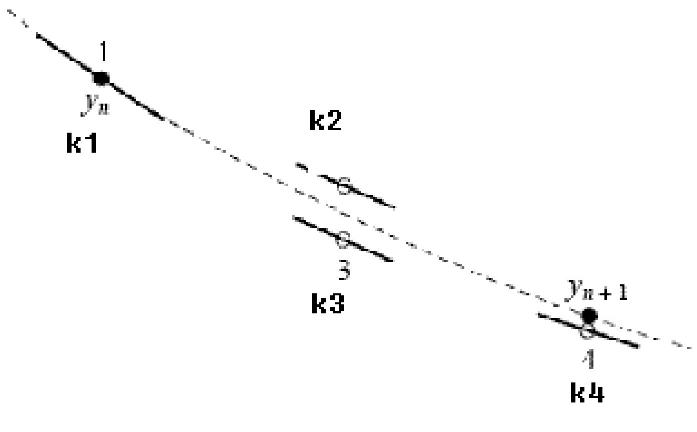

Figure 1

For 4th order Runge-Kutta method, in each step the derivative is evaluated four times like the figure (1) and the final value(y_{n+1}) is calculated by these derivatives. Usually it will have high accuracy if two derivatives can cancel the errors(like figure (1)) but in practice this is not always happen. Then I will give some basic ideas for the other two methods:

The Kaps-Rentrop method is also called Rosenbrock method. It was propoesd by Rosenbrock (1963) that Newton iterations can bt replaced by a single iteraion involving the inverse of a matrix such as $I - h\gamma f'(y(t_{n-1}))$. Here is an simple example to show how this method perform in each step:

$$(I - h(1 - \frac{\sqrt{2}}{2})f'(y_{n-1)})F_1 = f(y_{n-1})$$
$$(I - h(1 - \frac{\sqrt{2}}{2})f'(y_{n-1)})F_2 = f(y_{n-1} + h(-\frac{\sqrt{2}}{2} - \frac{1}{2})F_1)$$
$$y_n = y_{n-1} + hF_2$$

Each iteration we solve the above formula and get the final answer.

For the Bulirsh-Stoer method, the key idea is based on the Richardson extrapolation, The idea is to consider the 'y_{n+1}' of a numerical calculation as itself being an analytic function (if a complicated one) of an adjustable parameter like the step size h. That analytic function can be probed by performing the calculation with various values of h, none of them being necessarily small enough to yield the accuracy that we desire. When we know enough about the function, we fit it to some analytic form, and then evaluate it at that mythical and golden point h = 0 (see Figure (2)).

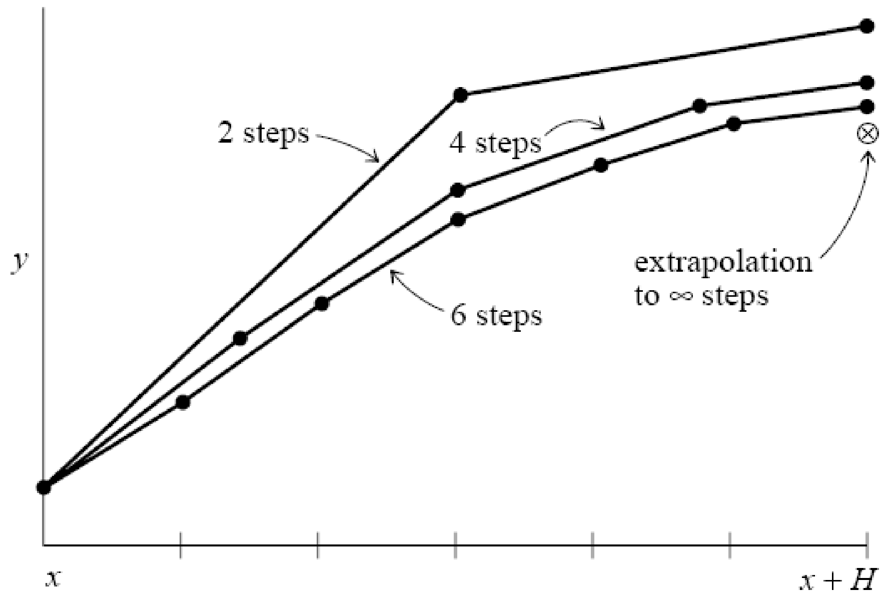

Figure 2

For more information about the methods to solve ODEs, one can read the book "Numerical Recipes: The Art of Scientific Computing" by William H. Press, published by Cambridge University Press.

1.2 Structure in OpenFOAM

The main part of the ODE solver in OpenFOAM can be found in:

$FOAM_SRC/ODE

Take a look at the subdirectory to *$FOAM_SRC/ODE/ODESolvers*:

KRR4/

RK/

SIBS/

ODESolver/

These directories define the methods of solving ODEs and also define the structure of the ODE.

Take a look at the *ODESolver/ODESolver.C*, like other OpenFOAM solvers, it starts from ”namespace Foam”:

```
// * * * * * * * * * * * * Static Data Members * * * * * *  * * * * //

defineTypeNameAndDebug(Foam::ODESolver, 0); namespace Foam {
    defineRunTimeSelectionTable(ODESolver, ODE);
};
```

Then the ODE constructors will be defined:

```
Foam::ODESolver::ODESolver(const ODE& ode) :
    n_(ode.nEqns()),
    yScale_(n_),
    dydx_(n_)
{}
```

Firstly it defines the dimension of the ordinary differential equation system, which means the order of the original ODE and the same as the number of the equations of the first-order ODE system we transfered into and it is’n_’ here, then the ’yScale_’ define y_{n+1} in Eqn.(1). The ’dydx’ will define the ordinary differential equation with transformed to the first order.

The final part of the code is solving the ODE the following things will be used:

```
const scalar xStart

scalar& hEst

const label MAXSTP = 10000;

yScale_[i] = mag(y[i]) + mag(dydx_[i]*h) + SMALL;
```

The 'xStart' is the initial time value for the ODE system, and it can close to zero but if we choose it into zero then an error will occur. The reason why this will happen I think is this program need a non-zero initial value for division. The 'hEst' is the time step in Eqn.(1). We can see the default max iteration is 10000. The last equation is similar what we discussed above in Eqn.(1). It's called Euler equation for ODEs and most numerical methods are based on it. Numerically we have:

$$\frac{y_{n+1} - y_n}{h} \approx \frac{dy}{dx}$$

Where h is the time step, if we take h h as small as possible then above equation holds. So the above codes define the initial values and the iteration equation.

Then we take a look at the file:

```
$FOAM_SRC/ODE/ODESolvers/RK/RK.C
```

First we can see that the whole Butcher tableau are defined at the beginning:

```
const scalar
    RK::a2 = 0.2, RK::a3 = 0.3, RK::a4 = 0.6, RK::a5 = 1.0, RK::a6 = 0.875,
    RK::b21 = 0.2, RK::b31 = 3.0/40.0, RK::b32 = 9.0/40.0,
    RK::b41 = 0.3, RK::b42 = -0.9, RK::b43 = 1.2,
    RK::b51 = -11.0/54.0, RK::b52 = 2.5, RK::b53 = -70.0/27.0,
    RK::b54 = 35.0/27.0,
    RK::b61 = 1631.0/55296.0, RK::b62 = 175.0/512.0, RK::b63 = 575.0/13824.0,
    RK::b64 = 44275.0/110592.0, RK::b65 = 253.0/4096.0,
    RK::c1 = 37.0/378.0, RK::c3 = 250.0/621.0,
    RK::c4 = 125.0/594.0, RK::c6 = 512.0/1771.0,
    RK::dc1 = RK::c1 - 2825.0/27648.0, RK::dc3 = RK::c3 - 18575.0/48384.0,
    RK::dc4 = RK::c4 - 13525.0/55296.0, RK::dc5 = -277.00/14336.0,
    RK::dc6 = RK::c6 - 0.25;
};
```

We can see that it's a 6th order RK method like the RK4 we have 4 point to weight but now we have more:

```
    forAll(yTemp_, i)
    {
        yTemp_[i] = y[i] + b21*h*dydx[i];
    }
```

This is the first point like 'K_1' in the RK4 then we should go to next point (time varible '$t_n \to t_n + h/2$' in RK4), this is done by:

```
ode.derivatives(x + a2*h, yTemp_, ak2_);
```

Then calculate the weighted slope of this point and add it into 'y_{n+1}' by:

```
    forAll(yTemp_, i)
    {
        yTemp_[i] = y[i] + h*(b31*dydx[i] + b32*ak2_[i]);
    }
```

Finally we get:

```
    forAll(yout, i)
    {
        yout[i] = y[i]
          + h*(c1*dydx[i] + c3*ak3_[i] + c4*ak4_[i] + c6*ak6_[i]);
    }
```

The 'yout[i]' gives us the final 'y_{n+1}' in the Eqn.(1)

2 Solving Our Own ODE

2.1 Original ODE in ODETest

The original ODE solved in ODETest located in

```
$FOAM_APP/test/ODETest
```

is a 4th-order ODE so the 'nEqns()' in the code is difine as 4:

```
    label nEqns() const
    {
        return 4;
    }
```

That means we have four ordinary equations to slove and thay are defined as following:

```
    void derivatives
    (
        const scalar x,
        const scalarField& y,
        scalarField& dydx
    ) const
    {
        dydx[0] = -y[1];
        dydx[1] = y[0] - (1.0/x)*y[1];
        dydx[2] = y[1] - (2.0/x)*y[2];
        dydx[3] = y[2] - (3.0/x)*y[3];
    }
```

If we transfer above codes into mathenmatical equations, they must be:

$$\frac{dy_1}{dx} = -y_2, \qquad \frac{dy_2}{dx} = y_1 - \frac{y_2}{x}, \qquad \frac{dy_3}{dx} = y_2 - \frac{2y_3}{x}, \qquad \frac{dy_4}{dx} = y_3 - \frac{3y_4}{x} \tag{4}$$

Then we calculate the jacobian for x and y:

$$\frac{-dy_2}{dx} = 0, \qquad \frac{d(y_1 - \frac{y_2}{x})}{dx} = \frac{y_2}{x^2}, \qquad \frac{d(y_2 - \frac{2y_3}{x})}{dx} = \frac{2y_3}{x^2}, \qquad \frac{d(y_3 - \frac{3y_4}{x})}{dx} = \frac{3y_4}{x^2}$$

$$\frac{-dy_2}{dy_1} = 0, \qquad \frac{-dy_2}{dy_2} = -1, \qquad \frac{-dy_2}{dy_3} = 0, \qquad \frac{-dy_2}{dy_4} = 0$$

$$\frac{d(y_1 - \frac{y_2}{x})}{dy_1} = 1, \qquad \frac{d(y_1 - \frac{y_2}{x})}{dy_2} = -\frac{1}{x}, \qquad \frac{d(y_1 - \frac{y_2}{x})}{dy_3} = 0, \qquad \frac{d(y_1 - \frac{y_2}{x})}{dy_4} = 0$$

$$\frac{d(y_2-\frac{2y_3}{x})}{dy_1}=0,\qquad \frac{d(y_2-\frac{2y_3}{x})}{dy_2}=1,\qquad \frac{d(y_2-\frac{2y_3}{x})}{dy_3}=-\frac{2}{x},\qquad \frac{d(y_2-\frac{2y_3}{x})}{dy_4}=0$$

$$\frac{d(y_3-\frac{3y_4}{x})}{dy_1}=0,\qquad \frac{d(y_3-\frac{3y_4}{x})}{dy_2}=0,\qquad \frac{d(y_3-\frac{3y_4}{x})}{dy_3}1,\qquad \frac{d(y_3-\frac{3y_4}{x})}{dy_4}=-\frac{3}{x}$$

And we transfer above equations into OpenFOAM code:

```
void jacobian
  (
      const scalar x,
      const scalarField& y,
      scalarField& dfdx,
      Matrix<scalar>& dfdy
  ) const
  {
      dfdx[0] = 0.0;
      dfdx[1] = (1.0/sqr(x))*y[1];
      dfdx[2] = (2.0/sqr(x))*y[2];
      dfdx[3] = (3.0/sqr(x))*y[3];

      dfdy[0][0] = 0.0;
      dfdy[0][1] = -1.0;
      dfdy[0][2] = 0.0;
      dfdy[0][3] = 0.0;

      dfdy[1][0] = 1.0;
      dfdy[1][1] = -1.0/x;
      dfdy[1][2] = 0.0;
      dfdy[1][3] = 0.0;

      dfdy[2][0] = 0.0;
      dfdy[2][1] = 1.0;
      dfdy[2][2] = -2.0/x;
      dfdy[2][3] = 0.0;

      dfdy[3][0] = 0.0;
      dfdy[3][1] = 0.0;
      dfdy[3][2] = 1.0;
      dfdy[3][3] = -3.0/x;
  }
```

Then we discuss how to define initial value in the code. Suppose we want to solve Eqn.(4) with the time $x=1$ to $x=2$ and the initial value of y in the original code are defined as some Bessel function of $x=1$ with different order. In OpenFOAM code they are written as:

```
scalar xStart = 1.0;

yStart[0] = ::Foam::j0(xStart); yStart[1] = ::Foam::j1(xStart);
yStart[2] = ::Foam::jn(2, xStart); yStart[3] = ::Foam::jn(3,xStart);
```

```
scalar x = xStart; scalar xEnd = x + 1.0; scalarField y = yStart;
```

2.2 Simple ODE

Given initial value ODES:

$$\frac{d^2y}{dx^2} = y, y(0) = 0, y'(0) = 1$$

If we want to solve our own ODEs with OpenFOAM, we need to first take a look at the following directory:

```
$FOAM_APP/test/ODETest
```

At the beginning of the code we firstly define the dimension of our ODEs:

```
label nEqns() const
    {

        return 2;
    }
```

This means the dimension of the ordinary differential equation system is '2' in our case. Because in our case the order of the ODE is '2'. Then change our ODE to first order ODEs.

$$\frac{dy}{dx} = z, \frac{dz}{dx} = y, y(0) = 0, z(0) = 1$$

In our code we just write as following:

```
    void derivatives
    (
        const scalar x,
        const scalarField& y,
        scalarField& dydx
    ) const
    {
        dydx[0] = y[1];
        dydx[1] = y[0];

    }
```

Just like MATLAB, the ODE must be written as a first order ordinary differential equation system, so we need to add one variable in this case and the equations are like above. Then we calculate the jacobian of the system not only for 'x' but also for 'y':

$$\frac{dz}{dx} = 0, \frac{dy}{dx} = 0$$

The above is the jacobian for x in our case.

$$\frac{dz}{dy} = 0, \frac{dz}{dz} = 1, \frac{dy}{dy} = 1, \frac{dy}{dz} = 0,$$

The above is the jacobian for y in our case.

```
    void jacobian
    (
        const scalar x,
        const scalarField& y,
        scalarField& dfdx,
        Matrix<scalar>& dfdy
    ) const
    {
        dfdx[0] = 0.0;
        dfdx[0] = 0.0;

        dfdy[0][0] = 0.0;
        dfdy[0][1] = 1.0;
        dfdy[1][0] = 1.0;
        dfdy[1][1] = 0.0;


    }
```

After having the derivatives, the solver also need the jacobian of the ODE system. Finally we add thee initial values in our case like:

```
scalar xStart = 0.00001; \\starting time

yStart[0] = 0.0;   \\y(0)=0

yStart[1] = 1.0;   \\y'(0)=1

scalar xEnd =  1.0;\\ end time
```

Then we need to complie the file so that we can run it in OpenFOAM using the linux command 'wmake' if we have the 'Make' direction correctly. Usually the 'Make/files' should contains following codes:

```
ODETest.C //source code file

EXE = $(FOAM_USER_APPBIN)/ODETest //command name and location you
                                  //want to have in linux
```

Which means the source code file name and command name and location you want to have in linux. The 'Make/options' should contains following codes:

```
EXE_INC = -I$(LIB_SRC)/ODE/lnInclude EXE_LIBS = -lODE
```

Which means the 'include' file location. After having all these files correctly, just type 'wmake' and it will complie the source file and run it we get:

```
Selecting ODE solver RK

Numerical:  y(2.0) = 2(1.17519 1.54307), hEst = 0.0736629
```

At the same time we solve the ODE in MATLAB, we get:

```
Y(2.0) = (1.1752 1.5431)
```

Which is exactly the same.

2.3 Boundary value ODEs

By reading Walter Gyllenram's paper, we need to solve the velocity ODE problem. For steady and non-viscous state the ODE is:

$$r\frac{d^2V_r}{dr^2} + \frac{dV_r}{dr} + \frac{4 \times V_r}{r} = 0$$

where $U_0 = U_\theta = 1$, $R = 1$ and $V_z = V_\theta = r$ in Walter Gyllenram's paper.

Re-write the ODE into first-order ODE system we have:

$$\frac{dy_1}{dx} = y_2$$
$$\frac{dy_2}{dx} = -\frac{4y_1}{x^2} - \frac{y_2}{x}$$

with the boundary value $y_1(0) = y_1(1) = 0$

Unfortunately this is a boundary value ODE problem and the solver in OpenFOAM is design for the initial value problem. So if we want to solve the problem above we need to modify the boundary condition to the initial value condition like $y_1(0) = y_1^{'}(0) = 0$ otherwise the problem can not be solved. Technically the BVP uses different methods in mathematics. The crucial distinction between initial value problems and two point boundary value problems is that in the IVP case we are able to start an known solution at its beginning (initial values) and just match it along by numerical integration to its end (final values); while in the BVP case, the boundary conditions at the starting point do not determine a unique solution to start with.

For simple boundary conditions, there is a method called shooting method. Using this method a BVP can be solved by two different IVP problems. It need some C++ programming skill to solve the BVP with OpenFOAM ODE solver, here I can give its basic idea. Suppose the two-point boundary value problem is linear:

$$y'' = p(x)y' + q(x)y + r(x), y(a) = y_a, y(b) = y_b$$

The shooting method said that we can solve two IVP instead of solving above BVP:

$$u'' = p(x)u' + q(x)u + r(x), u(a) = y_a.u'(a) = 0,$$
$$v'' = p(x)v' + q(x)v, v(a) = 0, v'(a) = 1.$$

If $v(b) \neq 0$, the solution of the original two-point BVP is given by:

$$y(x) = u(x) + \frac{y_b - u(b)}{v(b)}v(x)$$

There is a book called 'Numerical Solution of Two Point Boundary Value Problems' by Herbert B. Keller explain the mathematical theory of the boundary value problems and for the numerically solution one can also find it in the book "Numerical Recipes: The Art of Scientific Computing" by William H. Press, published by Cambridge University Press.