Numerical Methods for Ordinary Differential Equations

David Eberly
Magic Software, Inc.
http://www.magic-software.com

Created: March 2, 1999

Finding an initial point on a 1–dimensional ridge and traversing that ridge are both operations that require solving a system of ordinary differential equations of the form

$$\frac{dx(t)}{dt} = A(x(t)), \quad x(0) = x_0$$

where x_0 is initial data. The right–hand side is a function $A : \mathbb{R}^n \to \mathbb{R}^n$ which has no explicit dependence on t. Such a function is said to be *autonomous*. The general system of ordinary differential equations is

$$\frac{dx(t)}{dt} = F(t, x(t)), \quad x(0) = x_0$$

where $F : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$. This system can be solved with standard differential equation solvers. I give brief discussions of Euler's method, the Midpoint method, the Runge–Kutta fourth–order method with fixed step size, and the Runge–Kutta fifth–order method with adaptive step size. Each of the classes implementing these methods has two routines for performing a single step of the solver, one for autonomous systems and one for the general system. The classes are derived from an abstract base class, `mgcODE`, which has declaration

```
class mgcODE
{
public:
    // for dx/dt = F(t,x)
    typedef float (*Function)(float,float*);
    mgcODE (int _dim, float _step, Function* _F);
    virtual void Update (float tin, float* xin, float& tout, float* xout)=0;

    // for dx/dt = A(x)
    typedef float (*AutoFunction)(float*);
    mgcODE (int _dim, float _step, AutoFunction* _A);
    virtual void Update (float* xin, float* xout)=0;

    virtual void StepSize (float _step)=0;
    virtual float StepSize ()=0;

protected:
    int dim;
    float step;
```

```
    Function* F;
    AutoFunction* A;
};
```

The number of equations in the solver is `dim`. The two `StepSize` methods give access to the step size of the solvers. The right–hand side functions are stored in either `F` (nonautonomous) or `A` (autonomous). Any derived class must provide update methods for autonomous and nonautonomous systems.

1 Euler's Method

The easiest differential equation solver is Euler's method, a first–order method. Despite its bad reputation, it turns out not to be a bad choice if you want ridges calculated quickly in exchange for accuracy of the solution. The current ridge finding code uses this solver. Qualitatively the solutions look good. Given current point (t_n, x_n) in the algorithm, the next approximate solution point is generated by

$$\begin{aligned} x_{n+1} &= x_n + hF(t_n, x_n) \\ t_{n+1} &= t_n + h \end{aligned}$$

where $h \neq 0$ is a sufficiently small step size. I have been using $|h| = 0.1$ measured in pixel units since the B–spline grid is defined at integral points. The ridges appear to be qualitatively correct, even after 1000 Euler steps.

The class definition is

```
#include "ode.h"

class mgcEuler : public mgcODE
{
public:
    // for dx/dt = F(t,x)
    mgcEuler (int _dim, float _step, Function* _F);
    void Update (float tin, float* xin, float& tout, float* xout);

    // for dx/dt = A(x)
    mgcEuler (int _dim, float _step, AutoFunction* _A);
    void Update (float* xin, float* xout);

    void StepSize (float _step) { step = _step; }
    float StepSize () { return step; }
};
```

The method `Update` takes the current iterate and computes one step of the solver, returning the updated values. No check is made to see if the update method used matches the constructor–the user must take care to do so. If `F` is set, the other pointer `A` is set to null, so a segmentation fault will probably occur if the update does not match the constructor. An example of is

```
#include <iostream.h>

float F0 (float t, float *x) { return x[0]*x[0]; }
float F1 (float t, float *x) { return -2*x[0]*x[1]; }

int main ()
{
    const int dim = 2;
    const float step = 0.001;
    mgcODE::Function F[2] = { F0, F1 };

    mgcEuler ode(dim,step,F);
    float tin = 0.0, tout;
    float xin[dim] = { 1.0, 1.0 }, xout[dim];

    for (int i = 0; i < 10; i++) {
        cout << "t = " << tin << ' ';
        cout << "x = " << xin[0] << ' ';
        cout << "y = " << xin[1] << endl;
        ode.Update(tin,xin,tout,xout);
        for (int j = 0; j < dim; j++)
            xin[j] = xout[j];
        tin = tout;
    }

    // t = 0     x = 1       y = 1
    // t = 0.001 x = 1.001   y = 0.998
    // t = 0.002 x = 1.002   y = 0.996002
    // t = 0.003 x = 1.00301 y = 0.994006
    // t = 0.004 x = 1.00401 y = 0.992012
    // t = 0.005 x = 1.00502 y = 0.99002
    // t = 0.006 x = 1.00603 y = 0.98803
    // t = 0.007 x = 1.00704 y = 0.986042
    // t = 0.008 x = 1.00806 y = 0.984056
    // t = 0.009 x = 1.00907 y = 0.982072

    return 0;
}
```

2 Midpoint Method

This is a second–order Runge–Kutta algorithm. Given current point (t_n, x_n) in the algorithm, the next approximate solution point is generated by

$$\begin{aligned} A_1 &= F(t_n, x_n) \\ A_2 &= F(t_n + h/2, x_n + hA_1/2) \\ x_{n+1} &= x_n + hA_2 \\ t_{n+1} &= t_n + h \end{aligned}$$

The class definition is

```
#include "ode.h"

class mgcMidpoint : public mgcODE
{
public:
    // for dx/dt = F(t,x)
    mgcMidpoint (int _dim, float _step, Function* _F);
    void Update (float tin, float* xin, float& tout, float* xout);

    // for dx/dt = A(x)
    mgcMidpoint (int _dim, float _step, AutoFunction* _A);
    void Update (float* xin, float* xout);

    void StepSize (float _step) { step = _step; step2 = _step/2; }
    float StepSize () { return step; }

    ~mgcMidpoint ();

private:
    float step2;   // = step/2
    float* xtemp;
};
```

The method `Update` takes the current iterate and computes one step of the solver, returning the updated values. No check is made to see if the update method used matches the constructor–the user must take care to do so. If `F` is set, the other pointer `A` is set to null, so a segmentation fault will probably occur if the update does not match the constructor. The array `xtemp` is used for temporary storage of intermediate function evaluations. An example is

```
#include <iostream.h>

float F0 (float t, float *x) { return x[0]*x[0]; }
float F1 (float t, float *x) { return -2*x[0]*x[1]; }
```

```
int main ()
{
    const int dim = 2;
    const float step = 0.001;
    mgcODE::Function F[2] = { F0, F1 };

    mgcMidpoint ode(dim,step,F);
    float tin = 0.0, tout;
    float xin[dim] = { 1.0, 1.0 }, xout[dim];

    for (int i = 0; i < 10; i++) {
        cout << "t = " << tin << ' ';
        cout << "x = " << xin[0] << ' ';
        cout << "y = " << xin[1] << endl;
        ode.Update(tin,xin,tout,xout);
        for (int j = 0; j < dim; j++)
            xin[j] = xout[j];
        tin = tout;
    }

    // t = 0     x = 1       y = 1
    // t = 0.001 x = 1.001   y = 0.998001
    // t = 0.002 x = 1.002   y = 0.996004
    // t = 0.003 x = 1.00301 y = 0.994009
    // t = 0.004 x = 1.00402 y = 0.992016
    // t = 0.005 x = 1.00503 y = 0.990025
    // t = 0.006 x = 1.00604 y = 0.988036
    // t = 0.007 x = 1.00705 y = 0.986049
    // t = 0.008 x = 1.00806 y = 0.984064
    // t = 0.009 x = 1.00908 y = 0.982081

    return 0;
}
```

3 Runge–Kutta Fourth–Order Method

This is a fourth–order method with good accuracy. Given current point (t_n, x_n) in the algorithm, the next approximate solution point is generated by the algorithm

$$\begin{aligned} A_1 &= F(t_n, x_n) \\ A_2 &= F(t_n + h/2, x_n + hA_1/2) \\ A_3 &= F(t_n + h/2, x_n + hA_2/2) \\ A_4 &= F(t_n + h, x_n + hA_3) \\ x_{n+1} &= x_n + \tfrac{h}{6}(A_1 + 2A_2 + 2A_3 + A_4) \\ t_{n+1} &= t_n + h \end{aligned}$$

The class definition is

```
#include "ode.h"

class mgcRK4 : public mgcODE
{
public:
    // for dx/dt = F(t,x)
    mgcRK4 (int _dim, float _step, Function* _F);
    void Update (float tin, float* xin, float& tout, float* xout);

    // for dx/dt = A(x)
    mgcRK4 (int _dim, float _step, AutoFunction* _A);
    void Update (float* xin, float* xout);

    void StepSize (float _step)
        { step = _step; step2 = _step/2; step6 = _step/6; }
    float StepSize () { return step; }

    ~mgcRK4 ();

private:
    float step2;   // = step/2
    float step6;   // = step/6
    float* temp1;
    float* temp2;
    float* temp3;
    float* temp4;
    float* xtemp;
};
```

The method `Update` takes the current iterate and computes one step of the solver, returning the updated values. No check is made to see if the update method used matches the constructor–the user must take care

to do so. If `F` is set, the other pointer `A` is set to null, so a segmentation fault will probably occur if the update does not match the constructor. The arrays `temp1`, `temp2`, `temp3`, `temp4`, and `xtemp` are used for temporary storage of intermediate function evaluations. An example is

```
#include <iostream.h>

float F0 (float t, float *x) { return x[0]*x[0]; }
float F1 (float t, float *x) { return -2*x[0]*x[1]; }

int main ()
{
    const int dim = 2;
    const float step = 0.001;
    mgcODE::Function F[2] = { F0, F1 };

    mgcRK4 ode(dim,step,F);
    float tin = 0.0, tout;
    float xin[dim] = { 1.0, 1.0 }, xout[dim];

    for (int i = 0; i < 10; i++) {
        cout << "t = " << tin << ' ';
        cout << "x = " << xin[0] << ' ';
        cout << "y = " << xin[1] << endl;
        ode.Update(tin,xin,tout,xout);
        for (int j = 0; j < dim; j++)
            xin[j] = xout[j];
        tin = tout;
    }

    // t = 0     x = 1       y = 1
    // t = 0.001 x = 1.001   y = 0.998001
    // t = 0.002 x = 1.002   y = 0.996004
    // t = 0.003 x = 1.00301 y = 0.994009
    // t = 0.004 x = 1.00402 y = 0.992016
    // t = 0.005 x = 1.00503 y = 0.990025
    // t = 0.006 x = 1.00604 y = 0.988036
    // t = 0.007 x = 1.00705 y = 0.986049
    // t = 0.008 x = 1.00806 y = 0.984064
    // t = 0.009 x = 1.00908 y = 0.982081

    return 0;
}
```

4 Runge–Kutta with Adaptive Step

For some data sets, we may be able to dynamically adjust the step size h to reduce the total number of steps needed to traverse the ridge. The following algorithm is fifth–order and adjusts the step size accordingly.

1. Take two half–steps:

$$\begin{aligned} A_1 &= F(t_n, x_n) \\ A_2 &= F(t_n + h/4, x_n + hA_1/4) \\ A_3 &= F(t_n + h/4, x_n + hA_2/4) \\ A_4 &= F(t_n + h/2, x_n + hA_3/2) \\ x_{\text{inter}} &= x_n + \tfrac{h}{12}(A_1 + 2A_2 + 2A_3 + A_4) \end{aligned}$$

   and

$$\begin{aligned} B_1 &= F(t_n + h/2, x_{\text{inter}}) \\ B_2 &= F(t_n + 3h/4, x_{\text{inter}} + hB_1/4) \\ B_3 &= F(t_n + 3h/4, x_{\text{inter}} + hB_2/4) \\ B_4 &= F(t_n + h, x_{\text{inter}} + hB_3/2) \\ x_{\text{half}} &= x_{\text{inter}} + \tfrac{h}{12}(B_1 + 2B_2 + 2B_3 + B_4) \end{aligned}$$

2. Take a full step:

$$\begin{aligned} C_1 &= F(t_n, x_n) \\ C_2 &= F(t_n + h/2, x_n + hC_1/2) \\ C_3 &= F(t_n + h/2, x_n + hC_2/2) \\ C_4 &= F(t_n + h, x_n + hC_3) \\ x_{\text{full}} &= x_n + \tfrac{h}{6}(C_1 + 2C_2 + 2C_3 + C_4) \end{aligned}$$

3. Compute fractional error term Δ where $\varepsilon > 0$ is a constant specified by the user:

$$\Delta = \frac{1}{\varepsilon} \max_i \left| \frac{(x_{\text{ half}})_i - (x_{\text{full}})_i}{hF_i(t_n, x_n)} + \epsilon_0 \right|$$

   where ϵ_0 is a very small positive number which protects against the case $F_i(t_n, x_n) = 0$, in which case Δ becomes a very large positive number. The choice of ε is crucial. Its value probably could be selected by experimentation with problems in the class of those you are interested in.

4. If $\Delta \leq 1$, then the iteration is successful. The step size h is used and the next iterates are

$$\begin{aligned} x_{n+1} &= x_{\text{half}} + \tfrac{1}{15}\left(x_{\text{half}} - x_{\text{full}}\right), \\ t_{n+1} &= t_n + h. \end{aligned}$$

   The successful iteration suggests trying a larger step size for the next iteration. The step size is adjusted as follows. Let $S < 1$ be a number close to 1 (typical is $S = 0.9$). If $\Delta > (S/4)^5$, then a conservative increase is made: $h \leftarrow Sh\Delta^{-1/5}$. If $\Delta \leq (S/4)^5$, a more aggressive increase is made: $h \leftarrow 4h$.

5. If $\Delta > 1$, then the iteration fails. The step size must be reduced and the iteration is repeated starting with the initial iterate x_n. The adjustment is $h \leftarrow Sh\Delta^{-1/4}$. Repeat step 1 with this new step size. (A check must be made for the low probability case where $h \to 0$.)

The class definition is

```
#include "ode.h"

class mgcRK4Adapt : public mgcODE
{
public:
    // for dx/dt = F(t,x)
    mgcRK4Adapt (int _dim, float _step, Function* _F);
    void Update (float tin, float* xin, float& tout, float* xout);

    // for dx/dt = A(x)
    mgcRK4Adapt (int _dim, float _step, AutoFunction* _A);
    void Update (float* xin, float* xout);

    void StepSize (float _step) { step = _step; step_used = 0; }
    float StepSize () { return step; }

    mgcRK4Adapt& Safety (float _safety);
    mgcRK4Adapt& Epsilon (float _epsilon);
    float StepUsed () { return step_used; }
    int PassesMade () { return passes; }

    ~mgcRK4Adapt ();

private:
    int passes;
    float step_used, safety, epsilon, coeff;
    float* temp1;
    float* temp2;
    float* temp3;
    float* temp4;
    float* save1;
    float* xtemp;
    float* xinter;
    float* xhalf;
    float* xfull;
};
```

The value `step_used` saves the actual value of h used when the iteration is successful. The safety value S is stored as `safety`. The fractional error parameter ε is stored as `epsilon`. The value `coeff` is $(S/4)^5$ which is used for the decision on how much to increase h. The method `Update` takes the current iterate and computes one step of the solver, returning the updated values. No check is made to see if the update method used matches the constructor–the user must take care to do so. If `F` is set, the other pointer `A` is set to

null, so a segmentation fault will probably occur if the update does not match the constructor. The arrays `temp1`, `temp2`, `temp3`, `temp4`, `save1`, `xtemp`, `xinter`, `xhalf`, and `xfull` are used for temporary storage of intermediate function evaluations. The variable `passes` is used for statistics on how many times during the iteration the step size had to be reduced. An example is

```
#include <iostream.h>

// x'(t) = 1, y'(t) = 2y, x(0) = -1, y(0) = 1
// solution is x(t) = t-1, y(t) = exp(2t), so y = exp(2(x+1))

float F0 (float t, float *x) { return 1; }
float F1 (float t, float *x) { return 2*x[1]; }

int main ()
{
    const int dim = 2;
    const float step = 0.01;
    mgcODE::Function F[2] = { F0, F1 };

    mgcRK4Adapt ode(dim,step,F);
    float tin = 0.0, tout;
    float xin[dim] = { -1.0, 1.0 }, xout[dim];

    cout << "t x y step_tried step_used passes" << endl;
    for (int i = 0; i < 10; i++) {
        cout << tin << ' ' << xin[0] << ' ' << xin[1] << ' ';
        cout << ode.StepSize() << ' ' << ode.StepUsed() << ' ';
        cout << ode.PassesMade() << endl;
        ode.Update(tin,xin,tout,xout);
        for (int j = 0; j < dim; j++)
            xin[j] = xout[j];
        tin = tout;
    }

    // t          x           y       step_tried  step_used  passes
    // 0         -1           1       0.01        0          0
    // 0.01      -0.99        1.0202  0.04        0.01       1
    // 0.05      -0.95        1.10517 0.16        0.04       1
    // 0.21      -0.79        1.52196 0.373121    0.16       1
    // 0.583121  -0.416879    3.20983 0.436545    0.373121   1
    // 1.01967    0.0196654   7.68482 0.448811    0.436545   1
    // 1.46848    0.468476   18.8554  0.450984    0.448811   1
    // 1.91946    0.91946    46.4649  0.45136     0.450984   1
    // 2.37082    1.37082   114.588   0.451424    0.45136    1
    // 2.82224    1.82224   282.626   0.451437    0.451424   1

return 0;
}
```

One major concern with using Runge–Kutta methods is that they may be slow if evaluation of $F(t, x)$ or $A(x)$ is expensive. A suggestion in Numerical Recipes in C was to use the Bulirsch–Stoer method which has adaptive step size. I have not tried the method yet to compare it to the above methods.