

Queens College, CUNY, Department of Computer Science

Numerical Methods

CSCI 361 / 761

Fall 2017

Instructor: Dr. Sateesh Mane

November 5, 2017

16 Lecture 16

Numerical solution of systems of ordinary differential equations

- In this lecture we continue the study of **initial value problems**.
- In this lecture we present some higher order numerical integration algorithms.
- The **fourth order Runge Kutta** algorithm is one of the most popular of all numerical integration algorithms.

16.1 Basic notation

- We repeat the basic definitions from the previous lecture.
- Let the system of coupled differential equations be

$$\frac{d\mathbf{y}}{dx} = \mathbf{f}(x, \mathbf{y}). \quad (16.1.1)$$

- There are n unknown variables y_j , $j = 1, \dots, n$.
- The starting point is $x = x_0$, and the initial value \mathbf{y}_0 is given.
- The above is called the **Cauchy problem** or **initial value problem**.
- Our interest is to integrate eq. (16.1.1) numerically, using steps h_i , so $x_{i+1} = x_i + h_i$.
- The steps h_i need not be of equal size.
- We define $\mathbf{y}_i = \mathbf{y}(x_i)$.
- We employ the notation $\mathbf{y}_i^{\text{ex}} = \mathbf{y}^{\text{ex}}(x_i)$ to denote the exact solution.
- We employ the notation $\mathbf{y}_i^{\text{num}} = \mathbf{y}^{\text{num}}(x_i)$ to denote the numerical solution.

16.2 General remarks on explicit and implicit methods

- The system of equations is given by eq. (16.1.1).
- The explicit Euler integration formula for eq. (16.1.1) is given by

$$\mathbf{y}_{i+1}^{\text{exp}} = \mathbf{y}_i^{\text{exp}} + h_i \mathbf{f}(x_i, \mathbf{y}_i^{\text{exp}}). \quad (16.2.1)$$

- The implicit Euler integration formula for eq. (16.1.1) is given by

$$\mathbf{y}_{i+1}^{\text{imp}} - h_i \mathbf{f}(x_{i+1}, \mathbf{y}_{i+1}^{\text{imp}}) = \mathbf{y}_i^{\text{imp}}. \quad (16.2.2)$$

- Here we remark on the general structure of higher order integrators.
- A general higher order explicit integration method has a formula of the form

$$\mathbf{y}_{i+1}^{\text{exp}} = \mathbf{y}_i^{\text{exp}} + h \phi(x_i, \mathbf{y}_i^{\text{exp}}). \quad (16.2.3)$$

- For the explicit Euler method, $\phi = \mathbf{f}$, but for a higher order method ϕ is more complicated than simply \mathbf{f} .
- Nevertheless, in an explicit method ϕ is a function only of known quantities, i.e. x_i and $\mathbf{y}_i^{\text{exp}}$.
- A general higher order fully implicit integration method has a formula of the form

$$\mathbf{y}_{i+1}^{\text{imp}} - h \phi(x_{i+1}, \mathbf{y}_{i+1}^{\text{imp}}) = \mathbf{y}_i^{\text{imp}}. \quad (16.2.4)$$

- For the implicit Euler method, $\phi = \mathbf{f}$, but for a higher order method ϕ is more complicated than simply \mathbf{f} .
- Nevertheless, in a fully implicit method ϕ is a function only of quantities at the step $i + 1$, i.e. x_{i+1} and $\mathbf{y}_{i+1}^{\text{imp}}$.

16.3 Local truncation error

- As surprising as it may sound, let us substitute the *exact solution into the numerical integration formula* and examine what happens.
- Logically, we would expect to do things the other way around. We would like to know by how much the numerical solution fails to satisfy the exact system of differential equations.
- However, we do not know how to calculate the derivative $d\mathbf{y}^{\text{num}}(x)/dx$ for the numerical solution.
- However, the numerical integration formula contains only finite differences, i.e. function evaluations and no derivatives.
- We define the **local truncation error** \mathbf{u}_i as follows:

$$\mathbf{u}_i = \mathbf{y}_{i+1} - \mathbf{y}_i - h_i \phi(x_i, \mathbf{y}_i). \quad (16.3.1)$$

- The definition in eq. (16.3.1) applies for all numerical integration methods. Since we are substituting the *exact solution* into the numerical integration formula, there is no need to worry about “explicit” or “implicit” methods.
- For an integration method of order k , then $\mathbf{u}_i = O(h_i^{k+1})$.
- For the explicit Euler method, we saw above that

$$\mathbf{u}_i = \frac{h_i^2}{2} \mathbf{y}''(x_i). \quad (16.3.2)$$

- Note that authors vary in their notations and definitions.
 1. Some authors shift the index and define the above as \mathbf{u}_{i+1} .
 2. Other authors divide by h and define the local truncation error as \mathbf{u}_i/h , so they say the magnitude of the local truncation error is $O(h_i^k)$.
- Of course we do not know the exact solution $\mathbf{y}^{\text{ex}}(x)$. Nevertheless, eq. (16.3.1) can be employed to derive useful theorems.

16.4 Midpoint method

- The **midpoint method** is our first example of a higher order integration algorithm.
- In the explicit Euler method, the slope (derivative) $d\mathbf{y}/dx$ was estimated numerically by using the value of \mathbf{f} at the start of the step $x = x_i$, i.e. $\mathbf{f}(x_i, \mathbf{y}_i)$.
- In the implicit Euler method, the slope (derivative) $d\mathbf{y}/dx$ was estimated numerically by using the value of \mathbf{f} at the end of the step $x = x_{i+1}$, i.e. $\mathbf{f}(x_{i+1}, \mathbf{y}_{i+1})$.
- In the **midpoint method**, the slope (derivative) $d\mathbf{y}/dx$ is estimated numerically by using the value of \mathbf{f} at the midpoint of the step $x = x_i + \frac{1}{2}h_i$, i.e. $\mathbf{f}(x_i + \frac{1}{2}h_i, \mathbf{y}(x_i + \frac{1}{2}h_i))$.
- The midpoint integration formula is

$$\mathbf{y}_{i+1}^{\text{mid}} = \mathbf{y}_i^{\text{mid}} + h \mathbf{f}(x_i + \frac{1}{2}h_i, \mathbf{y}(x_i + \frac{1}{2}h_i)). \quad (16.4.1)$$

1. However, eq. (16.4.1) cannot be used directly, because **what is the value of $\mathbf{y}(x_i + \frac{1}{2}h_i)$?**

2. We only know the value of \mathbf{y} up to $x = x_i$ but not at $x = x_i + \frac{1}{2}h_i$.

- Hence we perform a preliminary explicit Euler integration step up to the midpoint to obtain

$$\mathbf{y}(x_i + \frac{1}{2}h_i) \simeq \mathbf{y}_i + \frac{1}{2}h_i \mathbf{f}(x_i, \mathbf{y}_i). \quad (16.4.2)$$

- We employ eq. (16.4.2) to substitute for $\mathbf{y}(x_i + \frac{1}{2}h_i)$ in eq. (16.4.1).
- This yields the formula for the midpoint method for one integration step. It requires two function evaluations.

$$\mathbf{g}_1 = \mathbf{f}(x_i, \mathbf{y}_i), \quad (16.4.3a)$$

$$\mathbf{g}_2 = \mathbf{f}\left(x_i + \frac{h_i}{2}, \mathbf{y}_i + \frac{h_i}{2} \mathbf{g}_1\right), \quad (16.4.3b)$$

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i \mathbf{g}_2. \quad (16.4.3c)$$

- The midpoint method is an **explicit integration algorithm**.
- The midpoint method is a **second order integration method**.
- The midpoint method is actually a **second order Runge–Kutta method**.
- We shall discuss Runge–Kutta methods below.

16.5 Trapezoid method

- The **trapezoid method** is another explicit second order numerical integration algorithm.
- In the trapezoid method, the slope (derivative) $d\mathbf{y}/dx$ is estimated numerically as the average of the values of \mathbf{f} at the two endpoints of the integration step.
- This formula for the trapezoid method is as follows. It requires two function evaluations.

$$\mathbf{g}_1 = \mathbf{f}(x_i, \mathbf{y}_i), \quad (16.5.1a)$$

$$\mathbf{g}_2 = \mathbf{f}(x_i + h_i, \mathbf{y}_i + h_i \mathbf{g}_1), \quad (16.5.1b)$$

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{h_i}{2} (\mathbf{g}_1 + \mathbf{g}_2). \quad (16.5.1c)$$

- The trapezoid method is an **explicit integration algorithm**.
- The trapezoid method is a **second order integration method**.
- The trapezoid method is also a **second order Runge–Kutta method**.
- In fact there is a family of second order Runge–Kutta methods, parameterized by λ as follows.

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i \left[\left(1 - \frac{1}{2\lambda}\right) \mathbf{f}(x_i, \mathbf{y}_i) + \frac{1}{2\lambda} \mathbf{f}(x_i + \lambda h_i, \mathbf{y}_i + \lambda h_i \mathbf{f}(x_i, \mathbf{y}_i)) \right]. \quad (16.5.2)$$

- The value $\lambda = \frac{1}{2}$ yields the midpoint method.
- The value $\lambda = 1$ yields the trapezoid method.

16.6 Fourth order Runge–Kutta RK4

- The **fourth order Runge–Kutta** method is a fourth order explicit integration method.
- It is one of the most popular, if not *the* most popular, of all numerical integration methods.
- It is so important it has its own acronym: RK4.
- The fourth order Runge–Kutta method for one integration step is as follows. It requires four function evaluations.

$$\mathbf{g}_1 = \mathbf{f}(x_i, \mathbf{y}_i), \quad (16.6.1a)$$

$$\mathbf{g}_2 = \mathbf{f}\left(x_i + \frac{h_i}{2}, \mathbf{y}_i + \frac{h_i}{2} \mathbf{g}_1\right), \quad (16.6.1b)$$

$$\mathbf{g}_3 = \mathbf{f}\left(x_i + \frac{h_i}{2}, \mathbf{y}_i + \frac{h_i}{2} \mathbf{g}_2\right), \quad (16.6.1c)$$

$$\mathbf{g}_4 = \mathbf{f}(x_i + h_i, \mathbf{y}_i + h_i \mathbf{g}_3), \quad (16.6.1d)$$

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{h_i}{6} (\mathbf{g}_1 + 2\mathbf{g}_2 + 2\mathbf{g}_3 + \mathbf{g}_4). \quad (16.6.1e)$$

- RK4 is an **explicit integration algorithm**.
- RK4 is a **fourth order integration method**.
- Compared to the Euler method, it requires only four function evaluations (as opposed to one), but the accuracy is fourth order (as opposed to first order).

16.7 More on Runge–Kutta

- “Runge–Kutta” is in fact an infinite family of numerical integration algorithms.
- Terminology: A general Runge–Kutta method has ***s* stages**. Here s is positive integer.
- Let a be an $s \times s$ matrix and b and c be arrays of length s . The entries in a , b and c are all constant coefficients.
- Then we compute a set of s intermediate function valuations as follows

$$\mathbf{g}_j = \mathbf{f}\left(x_i + c_j h_i, \mathbf{y}_i + h_i \sum_{k=1}^s a_{jk} \mathbf{g}_k\right). \quad (16.7.1)$$

- The integration step is given by the following formula

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i \sum_{j=1}^s b_j \mathbf{g}_j. \quad (16.7.2)$$

- The values of the b_j sum to unity:

$$\sum_{j=1}^s b_j = 1. \quad (16.7.3)$$

- The value of c_j is given by the **consistency condition**

$$c_j = \sum_{k=1}^s a_{jk}. \quad (16.7.4)$$

- In general, all practical Runge–Kutta methods are required to be consistent.
- Terminology: explicit, semi-explicit, implicit.
 1. A Runge-Kutta method is **explicit** if $a_{jk} = 0$ for all $k \geq j$.
 2. A Runge-Kutta method is **semi-explicit** if $a_{jk} = 0$ for $k > j$ but there are nonzero coefficients for some values where $k = j$.
 3. A Runge-Kutta method is **implicit** if there are coefficients $a_{jk} \neq 0$ for $k > j$.
 4. There are implicit Runge–Kutta methods of all orders.
 5. The value of a_{jk} need not be positive. Negative values are allowed.
- The fourth order RK4 method is the best known and most widely used member of the family. It is explicit and has four stages.
- Historical footnote:
 1. Karl Runge was a 19th century German mathematician.
 2. Runge was the father-in-law of Richard Courant, of the Courant Institute of Mathematical Sciences in New York City.

16.8 C++ code: explicit Euler method

16.8.1 One unknown

```
// f is a function declared elsewhere
// y_in is the input value at the start of the integration step
// y_out is the output value at the end of the integration step

int f(double x, double y, double & g)
{
    // compute value of g
    return 0;
}

int Euler_explicit(double x, double h, double y_in, double & y_out)
{
    y_out = y_in;
    double g = 0;
    int rc = f(x, y_in, g);
    if (rc) return rc;
    y_out = y_in + h*g;
    return 0;
}
```

16.8.2 Multiple unknowns

```
// f is a function declared elsewhere
// y_in is the input value at the start of the integration step
// y_out is the output value at the end of the integration step

int f(int n, double x, const std::vector<double> & y, std::vector<double> & g)
{
    // compute value of g
    return 0;
}

int Euler_explicit(int n, double x, double h,
                  std::vector<double> & y_in,
                  std::vector<double> & y_out)
{
    int i = 0;
    y_out = y_in;
    if ((n < 1) || (y_in.size() < n) || (y_out.size() < n)) return 1; // fail

    std::vector<double> g(n, 0.0);
    int rc = f(n, x, y_in, g);
    if (rc) return rc;

    for (i = 0; i < n; ++i) {
        y_out[i] = y_in[i] + h*g[i];
    }
    return 0;
}
```

16.9 C++ code: midpoint method

16.9.1 One unknown

```
// f is a function declared elsewhere
// y_in is the input value at the start of the integration step
// y_out is the output value at the end of the integration step

int f(double x, double y, double & g)
{
    // compute value of g
    return 0;
}

int midpoint(double x, double h, double y_in, double & y_out)
{
    int i = 0;
    int rc = 0;
    y_out = y_in;
    double g1 = 0;
    double g2 = 0;
    rc = f(x, y_in, g1);
    if (rc) return rc;
    rc = f(x+0.5*h, y_in + 0.5*h*g1, g2);
    if (rc) return rc;
    y_out = y_in + h*g2;
    return 0;
}
```

16.9.2 Multiple unknowns

```
int f(int n, double x, const std::vector<double> & y, std::vector<double> & g)
{
    // compute value of g
    return 0;
}

int midpoint(int n, double x, double h,
            std::vector<double> & y_in,
            std::vector<double> & y_out)
{
    int i = 0;
    int rc = 0;
    y_out = y_in;
    if ((n < 1) || (y_in.size() < n) || (y_out.size() < n)) return 1; // fail

    std::vector<double> g1(n, 0.0);
    std::vector<double> g2(n, 0.0);
    std::vector<double> y_tmp(n, 0.0); // temporary storage

    rc = f(n, x, y_in, g1);
    if (rc) return rc;

    for (i = 0; i < n; ++i) {
        y_tmp[i] = y_in[i] + 0.5*h*g1[i];
    }
    rc = f(n, x+0.5*h, y_tmp, g2);
    if (rc) return rc;

    for (i = 0; i < n; ++i) {
        y_out[i] = y_in[i] + h*g2[i];
    }
    return 0;
}
```

16.10 C++ code: trapezoid method

16.10.1 One unknown

```
// f is a function declared elsewhere
// y_in is the input value at the start of the integration step
// y_out is the output value at the end of the integration step

int f(double x, double y, double & g)
{
    // compute value of g
    return 0;
}

int trapezoid(double x, double h, double y_in, double & y_out)
{
    int i = 0;
    int rc = 0;
    y_out = y_in;
    double g1 = 0;
    double g2 = 0;
    rc = f(x, y_in, g1);
    if (rc) return rc;
    rc = f(x+h, y_in + h*g1, g2);
    if (rc) return rc;
    y_out = y_in + 0.5*h*(g1 + g2);
    return 0;
}
```

16.10.2 Multiple unknowns

```
int f(int n, double x, const std::vector<double> & y, std::vector<double> & g)
{
    // compute value of g
    return 0;
}

int trapezoid(int n, double x, double h,
              std::vector<double> & y_in,
              std::vector<double> & y_out)
{
    int i = 0;
    int rc = 0;
    y_out = y_in;
    if ((n < 1) || (y_in.size() < n) || (y_out.size() < n)) return 1; // fail

    std::vector<double> g1(n, 0.0);
    std::vector<double> g2(n, 0.0);
    std::vector<double> y_tmp(n, 0.0); // temporary storage

    rc = f(n, x, y_in, g1);
    if (rc) return rc;

    for (i = 0; i < n; ++i) {
        y_tmp[i] = y_in[i] + h*g1[i];
    }
    rc = f(n, x+h, y_tmp, g2);
    if (rc) return rc;

    for (i = 0; i < n; ++i) {
        y_out[i] = y_in[i] + 0.5*h*(g1[i] + g2[i]);
    }
    return 0;
}
```

16.11 C++ code: RK4

16.11.1 One unknown

```
// f is a function declared elsewhere
// y_in is the input value at the start of the integration step
// y_out is the output value at the end of the integration step

int f(double x, double y, double & g)
{
    // compute value of g
    return 0;
}

int RK4(double x, double h, double y_in, double & y_out)
{
    int i = 0;
    int rc = 0;
    y_out = y_in;
    double g1 = 0;
    double g2 = 0;
    double g3 = 0;
    double g4 = 0;
    rc = f(x, y_in, g1);
    if (rc) return rc;
    rc = f(x+0.5*h, y_in + 0.5*h*g1, g2);
    if (rc) return rc;
    rc = f(x+0.5*h, y_in + 0.5*h*g2, g3);
    if (rc) return rc;
    rc = f(x+h, y_in + h*g3, g4);
    if (rc) return rc;
    y_out = y_in + (h/6.0)*(g1 + 2.0*g2 + 2.0*g3 + g4);
    return 0;
}
```

16.11.2 Multiple unknowns

```
int f(int n, double x, const std::vector<double> & y, std::vector<double> & g)
{
    // compute value of g
    return 0;
}

int RK4(int n, double x, double h,
        std::vector<double> & y_in,
        std::vector<double> & y_out)
{
    int i = 0;
    int rc = 0;
    y_out = y_in;
    if ((n < 1) || (y_in.size() < n) || (y_out.size() < n)) return 1; // fail

    std::vector<double> g1(n, 0.0);
    std::vector<double> g2(n, 0.0);
    std::vector<double> g3(n, 0.0);
    std::vector<double> g4(n, 0.0);
    std::vector<double> y_tmp(n, 0.0); // temporary storage

    rc = f(n, x, y_in, g1);
    if (rc) return rc;
    for (i = 0; i < n; ++i) {
        y_tmp[i] = y_in[i] + 0.5*h*g1[i];
    }
    rc = f(n, x+0.5*h, y_tmp, g2);
    if (rc) return rc;
    for (i = 0; i < n; ++i) {
        y_tmp[i] = y_in[i] + 0.5*h*g2[i];
    }
    rc = f(n, x+0.5*h, y_tmp, g3);
    if (rc) return rc;
    for (i = 0; i < n; ++i) {
        y_tmp[i] = y_in[i] + h*g3[i];
    }
    rc = f(n, x+h, y_tmp, g4);
    if (rc) return rc;
    for (i = 0; i < n; ++i) {
        y_out[i] = y_in[i] + (h/6.0)*(g1[i] + 2.0*g2[i] + 2.0*g3[i] + g4[i]);
    }
    return 0;
}
```