Queens College, CUNY, Department of Computer Science

**Numerical Methods**

**CSCI 361 / 761**

**Spring 2018**

Instructor: Dr. Sateesh Mane



April 29, 2018

# 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

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

- There are $m$ unknown variables $y_j$, $j = 1, \ldots, m$.
- The starting point is $x = x_0$, and the initial value $\boldsymbol{y}_0$ 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 $\boldsymbol{y}_i = \boldsymbol{y}(x_i)$.
- We employ the notation $\boldsymbol{y}_i^{\text{ex}} = \boldsymbol{y}^{\text{ex}}(x_i)$ to denote the exact solution.
- We employ the notation $\boldsymbol{y}_i^{\text{num}} = \boldsymbol{y}^{\text{num}}(x_i)$ 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

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

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

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

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

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

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

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

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

## 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 $d\boldsymbol{y}^{\text{num}}(x)/dx$ 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** $\boldsymbol{u}_i$ as follows:

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

- 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 $\boldsymbol{u}_i = O(h_i^{k+1})$.
- For the explicit Euler method, we saw above that

$$\boldsymbol{u}_i = \frac{h_i^2}{2}\, \boldsymbol{y}''(x_i)\,. \tag{16.3.2}$$

- Note that authors vary in their notations and definitions.
  1. Some authors shift the index and define the above as $\boldsymbol{u}_{i+1}$.
  2. Other authors divide by $h$ and define the local truncation error as $\boldsymbol{u}_i/h$, so they say the magnitude of the local truncation error is $O(h_i^k)$.
- Of course we do not know the exact solution $y^{\text{ex}}(x)$. 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) $d\boldsymbol{y}/dx$ was estimated numerically by using the value of $\boldsymbol{f}$ at the start of the step $x = x_i$, i.e. $\boldsymbol{f}(x_i, \boldsymbol{y}_i)$.
- In the implicit Euler method, the slope (derivative) $d\boldsymbol{y}/dx$ was estimated numerically by using the value of $\boldsymbol{f}$ at the end of the step $x = x_{i+1}$, i.e. $\boldsymbol{f}(x_{i+1}, \boldsymbol{y}_{i+1})$.
- In the **midpoint method**, the slope (derivative) $d\boldsymbol{y}/dx$ is estimated numerically by using the value of $\boldsymbol{f}$ at the midpoint of the step $x = x_i + \frac{1}{2}h_i$, i.e. $\boldsymbol{f}(x_i + \frac{1}{2}h_i, \boldsymbol{y}(x_i + \frac{1}{2}h_i))$.
- The midpoint integration formula is

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

  1. However, eq. (16.4.1) cannot be used directly, because ***what is the value of*** $\boldsymbol{y}(x_i + \frac{1}{2}h_i)$***?***
  2. We only know the value of $\boldsymbol{y}$ up to $x = x_i$ but not at $x = x_i + \frac{1}{2}h_i$.

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

$$\boldsymbol{y}(x_i + \tfrac{1}{2}h_i) \simeq \boldsymbol{y}_i + \tfrac{1}{2}h_i\, \boldsymbol{f}(x_i, \boldsymbol{y}_i)\,. \tag{16.4.2}$$

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

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

$$\boldsymbol{g}_2 = \boldsymbol{f}\Big(x_i + \frac{h_i}{2}, \boldsymbol{y}_i + \frac{h_i}{2}\,\boldsymbol{g}_1\Big)\,, \tag{16.4.3b}$$

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h_i\, \boldsymbol{g}_2\,. \tag{16.4.3c}$$

- 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) $d\boldsymbol{y}/dx$ is estimated numerically as the average of the values of $\boldsymbol{f}$ at the two endpoints of the integration step.
- This formula for the trapezoid method is as follows. It requires two function evaluations.

$$\boldsymbol{g}_1 = \boldsymbol{f}(x_i, \boldsymbol{y}_i)\,, \tag{16.5.1a}$$
$$\boldsymbol{g}_2 = \boldsymbol{f}(x_i + h_i, \boldsymbol{y}_i + h_i\,\boldsymbol{g}_1)\,, \tag{16.5.1b}$$
$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + \frac{h_i}{2}\left(\boldsymbol{g}_1 + \boldsymbol{g}_2\right). \tag{16.5.1c}$$

- 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 $\lambda$ as follows.

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

- 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.

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

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

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

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

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

- `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 integraton 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

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

- The integration step is given by the following formula

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h_i \sum_{j=1}^{s} b_j \boldsymbol{g}_j \, . \tag{16.7.2}$$

- The values of the $b_j$ sum to unity:

$$\sum_{j=1}^{s} b_j = 1 \, . \tag{16.7.3}$$

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

$$c_j = \sum_{k=1}^{s} a_{jk} \, . \tag{16.7.4}$$

- 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 $19^{th}$ 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 m, double x, const std::vector<double> & y, std::vector<double> & g)
{
  // compute value of g
  return 0;
}

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

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

  for (i = 0; i < m; ++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 m, double x, const std::vector<double> & y, std::vector<double> & g)
{
  // compute value of g
  return 0;
}

int midpoint(int m, 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 ((m < 1) || (y_in.size() < m) || (y_out.size() < m)) return 1; // fail

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

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

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

  for (i = 0; i < m; ++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 m, double x, const std::vector<double> & y, std::vector<double> & g)
{
  // compute value of g
  return 0;
}

int trapezoid(int m, 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 ((m < 1) || (y_in.size() < m) || (y_out.size() < m)) return 1; // fail

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

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

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

  for (i = 0; i < m; ++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 m, double x, const std::vector<double> & y, std::vector<double> & g)
{
  // compute value of g
  return 0;
}

int RK4(int m, 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 ((m < 1) || (y_in.size() < m) || (y_out.size() < m)) return 1; // fail

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

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