Initial Value Problem

by Loïc Quertenmont, PhD

LINFO1113 - 2019-2020

Programme

Cours	1	Librairies mathématiques, représentation des nombres en Python et erreurs liées
Cours	2,3	Résolution des systèmes linéaires
Cours	4,5	Interpolation et Régression Linéaires
Cours	6,7	Racines d'équations
Cours	8	Différentiation numérique
Cours	9	Intégration numérique
Cours	**10**	**Equations Différentielles Ordinaires (IVP)**
Cours	11	Introduction à l'optimisation
Cours	12,13	Rappel / Répétition

Outline

- **Introduction**
- **Initial Value Problem**
- **Euler's Method**
- **Runge-Kutta Methods**
  - **2^{nd} order**
  - **4^{th} order**
- **Stability and Stiffness**

Introduction

Introduction

- First-order differential equation (ODE):
  - General Form:

$$y' = f(x, y)$$

  - Where $y' = \frac{dy}{dx}$ and $f(x, y)$ is a given function
  - The solution of this equation contains an arbitrary constant (that comes from the integration)
  - To know this constant, we need to know a point on the solution curve; y must be specified at some value of x

$$\mathrm{y}(a) = \alpha$$

  - **Initial Value Problem (IVP)**
  - Also named “Cauchy Problem”

Remark

- An ordinary differential equation (ODE) of order n

$$y^{(n)} = f\left(x, y, y', y'', \dots, y^{(n-1)}\right)$$

- Can always be transformed into n first-order ODE
- Using the notation:

$$y_0 = y, \quad y_1 = y', \quad y_2 = y'', \quad \dots, \quad y_n = y^{(n)}$$

- The equivalent set of first-order ODE is:

$$y'_0 = y_1, \quad y'_1 = y_2, \quad \dots, y'_{n-1} = y_n = f(x, y_0, y_1, y_2, \dots, y_{n-1})$$

- The solution requires n-initial values

$$y_0(a) = \alpha_0, \quad y_1(a) = \alpha_1, \quad \dots, \quad y_{n-1}(a) = \alpha_{n-1}$$

Remark

- The equations:
  - $y'_0 = y_1\,, \quad y'_1 = y_2, \quad \dots\,, \; y'_n = f(x, y_0, y_1, y_2, \dots, y_{n-1})$
  - $y_0(a) = \alpha_0\,, \quad y_1(a) = \alpha_1, \; \dots, \; y_{n-1}(a) = \alpha_{n-1}$

- Can be rewritten in a concise way, using matrix notation:

  - $\boldsymbol{y'} = F(x, \boldsymbol{y})$ where $F(x, \boldsymbol{y}) = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ f(x, \boldsymbol{y}) \end{bmatrix}$
  - $\boldsymbol{y}(a) = \boldsymbol{\alpha}$

Remark

- If y_i are specified at different values of x
  - This is call a **boundary value problem**
  - This is more complex than an IVP
  - Not discussed in this lecture

- Example

$$y'' = -y \qquad y(0) = 1 \qquad y'(0) = 0$$

  - Is an initial value problem

$$y'' = -y \qquad y(0) = 1 \qquad y(\pi) = 0$$

  - Is a boundary Problem

Example of IVP (1)

- **Population Growth**
- Let $N(t)$ be the population of a country at time t
- Let's assume we know that the population increase depends linearly on the population size. Mathematically:

$$\frac{dN(t)}{dt} = kN(t)$$

- It would be nice to know what function $N(t)$ fulfills this equation. Then we can predict the population size for the future.

Example of IVP (2)

- $\frac{dN(t)}{dt} = kN(t)$
- $\rightarrow \quad \frac{1}{kN(t)} dN(t) = dt$
- $\rightarrow \int \frac{1}{kN(t)} dN(t) = \int dt$
- $\rightarrow \quad \frac{1}{k} \log |kN(t)| = t + C_1$
- $\rightarrow \quad kN(t) = e^{kt+kC_1}$
- $\rightarrow \quad N(t) = \frac{1}{k} e^{kt+kC_1} = \frac{1}{k} e^{kC_1} e^{kt}$
- $\rightarrow \quad N(t) = Ce^{kt} \quad \text{with } C = \frac{1}{k} e^{kC_1}$

Example of IVP (3)

- Let's assume that we know that $k = 0.2$ and that the initial population at time $t = 0$ is 10000:

$$N(0) = 10000$$

- What will be the population $N(3)$?
- From $N(t) = Ce^{0.2t}$ we get

$$N(0) = Ce^{0} = 10000$$
$$\rightarrow \quad C = 10000$$

- Therefore

$$N(3) = 10000e^{0.2 \cdot 3} \approx 18221$$

Euler's Method

Euler's Method

- Euler's method is conceptually simple
- Starts from Taylor Expansion (truncated)

$$\boldsymbol{y}(x + h) \approx \boldsymbol{y}(x) + \boldsymbol{y}'(x)h$$

Known at x=a Given by f(x,y)

- It predicts the solution $\boldsymbol{y}$ at $x + h$ from the information available at x

- It can be used to move the solution forward in steps of h, **starting with the given initial values of x and y**

Illustration

The solution is a curve in the plane x,y ←→ y(x)

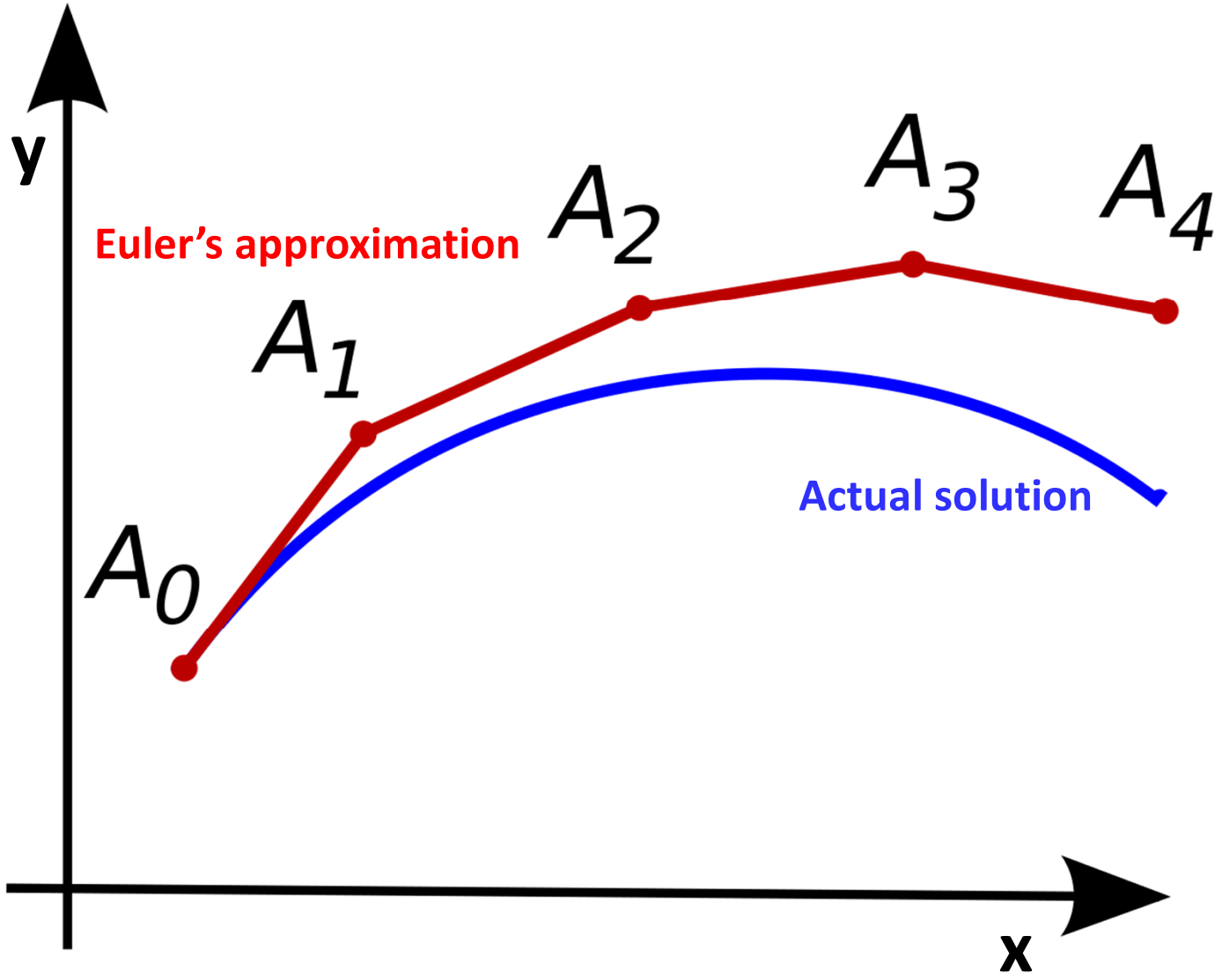


Illustration from Wikipedia

Error

- The truncation error in Taylor expansion is
  - $E = \frac{1}{2} y''(\xi) h^2 \quad \text{with} \quad x < \xi < x + h$
  - $E = O(h^2)$

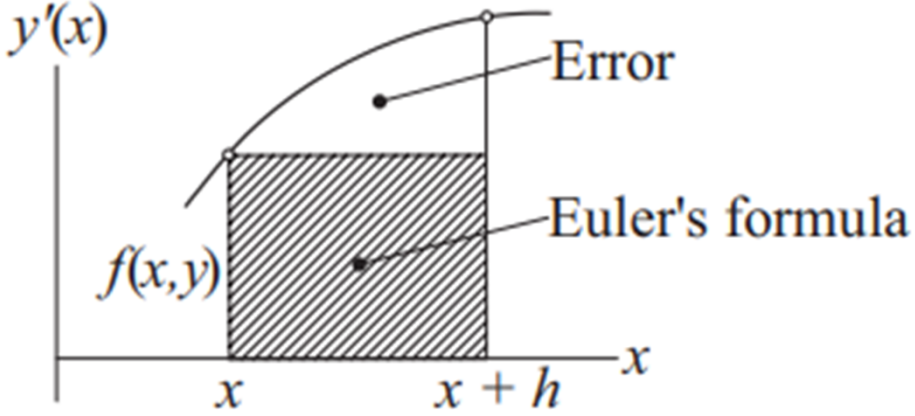
- The accumulated error E_{acc} is obtained by assuming the error is constant over a step
  - $E_{acc} = nE = \frac{x_n - x_0}{h} E = O(h)$

Error

- The change in the solution y between x and $x + h$ is:

$$y(x + h) - y(h) = \int_x^{x+h} y'dx = \int_x^{x+h} f(x,y)dx$$

- Figure of $y'vs\ x$
- Truncation Error is clearly proportional to the slope $(y')' \rightarrow y''$



- Reducing the error to an acceptable level requires a very small step (h). Which induces heavy computation and possibly large roundoff error.
- → Euler's method is therefore rarely used in practice

Example (1)

- Solve the initial value problem given by:

- $y' + 4y = x^2$ and $y(0) = 1$

- Use Euler in steps of h=0.01 from x=0 to 0.03
  - Compare to the true solution and compute the accumulated error at each step

- The analytical solution is: $y = \frac{31}{32}e^{-4x} + \frac{1}{4}x^2 - \frac{1}{8}x + \frac{1}{32}$

Example (2)

- **Solution:**
  - Using the notation $x_i = ih$ and $y_i = y(x_i)$
  - Euler Formula: $y_{i+1} = y_i + y_i'h$ with $y_i' = x_i^2 - 4y_i$
  - Step1 from x=0 to x=0.01

$$y_0 = 0$$

$$y_0' = x_0^2 - 4y_0 = 0^2 - 4(1) = -4$$

$$y_1 = y_0 + y_0'h = 1 + (-4)(0.01) = 0.96$$

$$(y_1)_{\text{exact}} = \frac{31}{32}e^{-4(0.01)} + \frac{1}{4}0.01^2 - \frac{1}{8}0.01 + \frac{1}{32} = 0.9608$$

$$E_{\text{acc}} = 0.96 - 0.9608 = -0.0008$$

Example (3)

- **Solution:**
  - Step2 from x=0.01 to x=0.02

$$y_1' = x_1^2 - 4y_1 = 0.01^2 - 4(0.96) = -3.840$$

$$y_2 = y_1 + y_1' h = 0.96 + (-3.840)(0.01) = 0.9216$$

$$(y_2)_{\text{exact}} = \frac{31}{32}e^{-4(0.02)} + \frac{1}{4}0.02^2 - \frac{1}{8}0.02 + \frac{1}{32} = 0.9231$$

$$E_{\text{acc}} = 0.9216 - 0.9231 = -0.0015$$

  - Step3 from x=0.01 to x=0.02

$$y_2' = x_2^2 - 4y_2 = 0.02^2 - 4(0.9216) = -3.686$$

$$y_3 = y_2 + y_2' h = 0.9216 + (-3.686)(0.01) = 0.8847$$

$$(y_3)_{\text{exact}} = \frac{31}{32}e^{-4(0.03)} + \frac{1}{4}0.03^2 - \frac{1}{8}0.03 + \frac{1}{32} = 0.8869$$

$$E_{\text{acc}} = 0.8847 - 0.8869 = -0.0022$$

  - The per-step error is roughly constant at 0.008.
    After 10 integration steps the accumulated error would be approximately 0.08 → Reducing the solution to one significant figure accuracy.
    After 100 steps all significant figures would be lost

Algorithm

- **Can you implement the Euler's Algorithm ?**

```
import numpy as np
def integrate(F,x,y,xStop,h):
    X = []
    Y = []
    X.append(x)
    Y.append(y)
    while x < xStop:
        h = min(h,xStop - x)
        y = y + h*F(x,y)
        x = x + h
        X.append(x)
        Y.append(y)
    return np.array(X),np.array(Y)
```

Example 2 (1)

- Solve the initial value problem given by:

- $y'' = 0.1y' - x$ and $y(0) = 0$ and $y'(0) = 1$

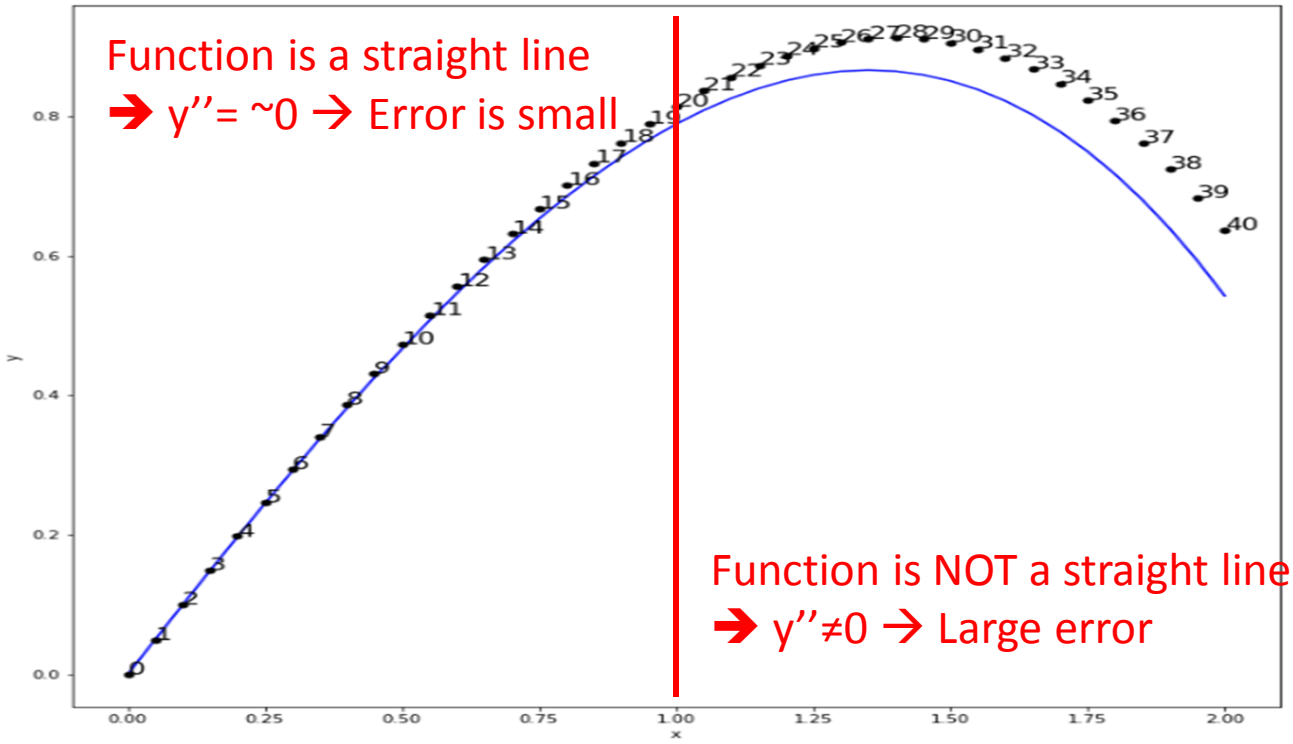
- Use Euler in steps of h=0.05 from x=0 to 2
  - Compare to the true solution and compute the accumulated error at each step

- The analytical solution is: $y = 100x - 5x^2 + 990(e^{-0.1x} - 1)$

Example 2 (2)

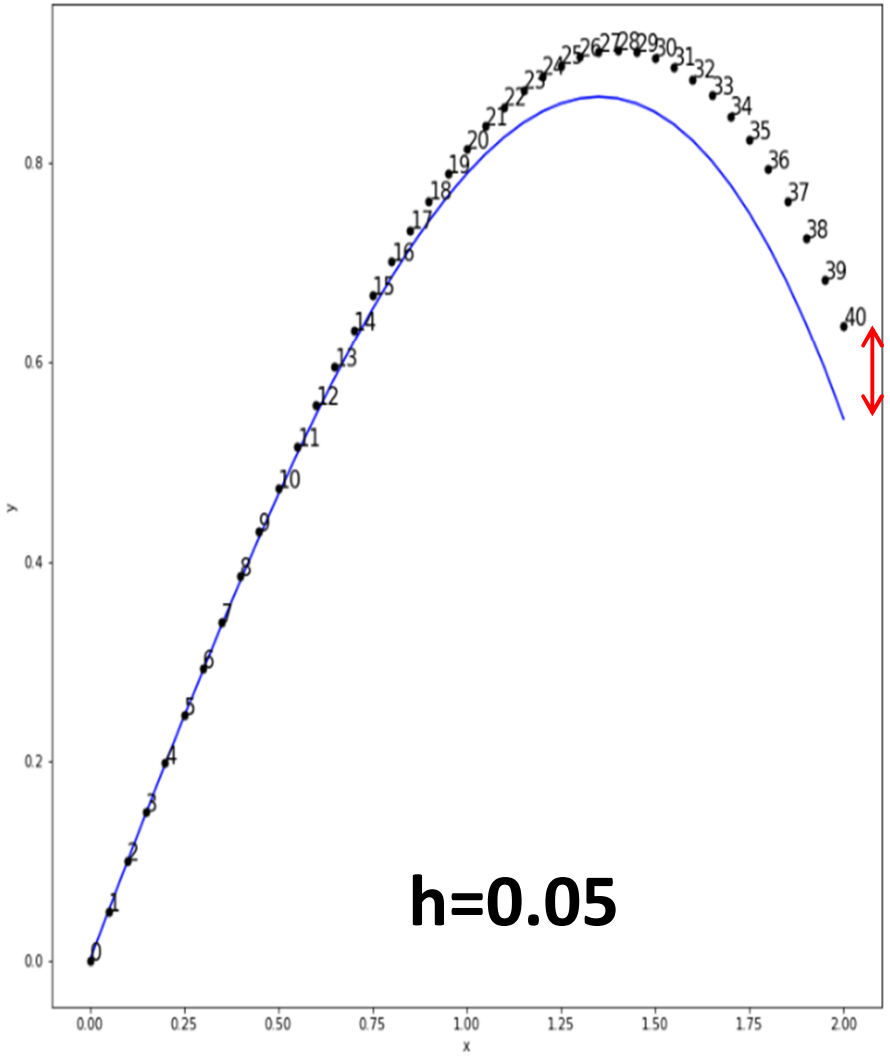
- **Solution:**
  - Using the notation $x_i = ih$ and $y_i = y(x_i)$
  - Euler Formula:

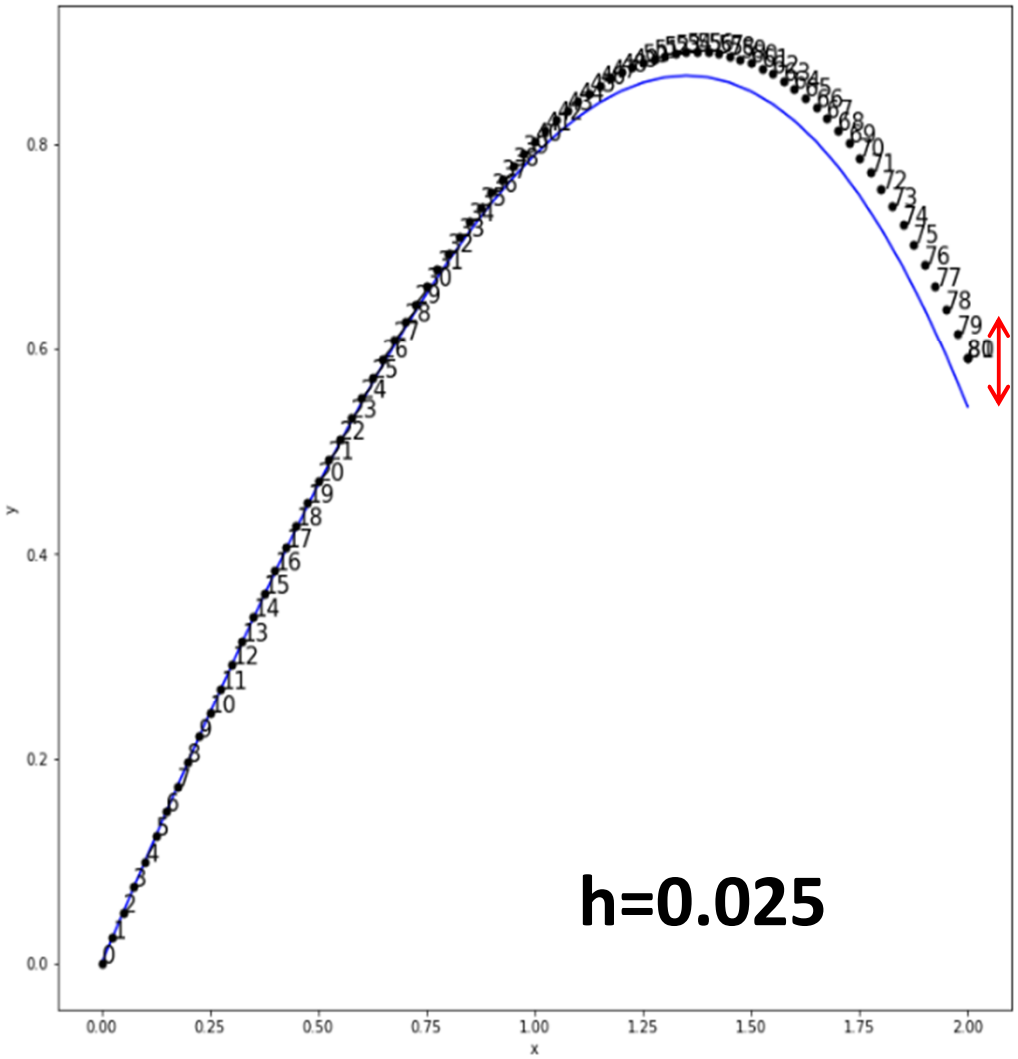
$$\mathbf{F}(x, \mathbf{y}) = \begin{bmatrix} y_0' \\ y_1' \end{bmatrix} = \begin{bmatrix} y_1 \\ -0.1y_1 - x \end{bmatrix} \qquad \mathbf{y}(0) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$



Example 2 (3)

- **Same solution with h→h/2=0.025**





Runge Kutta

Second Order Taylor Expansion

- The local error in Euler's method is of $O(h^2)$
  - Accumulated error is $O(h)$

- To reduce it, we need to go one step further in Taylor Expansion.
- $\boldsymbol{y}(x+h) \approx \boldsymbol{y}(x) + \boldsymbol{y}'(x)h + \boldsymbol{y}''(x)\frac{h^2}{2!} + O(h^3)$
- $\boldsymbol{y}(x+h) \approx \boldsymbol{y}(x) + \boldsymbol{F}(x,\boldsymbol{y})h + \boldsymbol{F}'(x,\boldsymbol{y})\frac{h^2}{2!} + O(h^3)$
  - $with\ F'(x,\boldsymbol{y}) = \frac{\boldsymbol{\partial F}}{\partial x} + \sum_{i=0}^{n-1}\frac{\boldsymbol{\partial F}}{\partial y_i}y'_i = \frac{\boldsymbol{\partial F}}{\partial x} + \sum_{i=0}^{n-1}\frac{\boldsymbol{\partial F}}{\partial y_i}F_i(x,\boldsymbol{y})$
- $\boldsymbol{y}(x+h) \approx \boldsymbol{y}(x) + \boldsymbol{F}(x,\boldsymbol{y})h + \frac{h^2}{2!}\left(\frac{\boldsymbol{\partial F}}{\partial x} + \sum_{i=0}^{n-1}\frac{\boldsymbol{\partial F}}{\partial y_i}F_i(x,\boldsymbol{y})\right) + O(h^3)$

- Not very convenient, because we need to derivate $\boldsymbol{F}$ with respect to several variables

Runge-Kutta (2nd order)

- In Runge-Kutta of 2nd order (RK2), we assume that:
  - $\boldsymbol{y}(x+h) \approx \boldsymbol{y}(x) + c_0 h\boldsymbol{F}(x,\boldsymbol{y}) + \boldsymbol{c_1} h\boldsymbol{F}\big(x+ph\,, \boldsymbol{y} + qh\,\boldsymbol{F}(x,\boldsymbol{y})\big)$

- Remark: a Taylor expansion of the last term is:
  - $\boldsymbol{F}\big(x+ph\,, \boldsymbol{y} + qh\,\boldsymbol{F}(x,\boldsymbol{y})\big) = \boldsymbol{F}(x,\boldsymbol{y}) + \frac{\boldsymbol{\partial F}}{\partial x} ph + qh \sum_{i=0}^{n-1} \frac{\boldsymbol{\partial F}}{\partial y_i} F_i(x,\boldsymbol{y})$
- So we can rewrite RK2 as:
  - $\boldsymbol{y}(x+h) \approx \boldsymbol{y}(x) + c_0 h\boldsymbol{F}(x,\boldsymbol{y}) + \boldsymbol{c_1} h\left[\boldsymbol{F}(x,\boldsymbol{y}) + \frac{\boldsymbol{\partial F}}{\partial x} ph + qh \sum_{i=0}^{n-1} \frac{\boldsymbol{\partial F}}{\partial y_i} F_i(x,\boldsymbol{y})\right]$

- RK2 coincide with 2nd order taylor expansion of $y(x+h)$:
  - $\boldsymbol{y}(x+h) \approx \boldsymbol{y}(x) + \boldsymbol{F}(x,\boldsymbol{y})h + \frac{h^2}{2!}\left(\frac{\boldsymbol{\partial F}}{\partial x} + \sum_{i=0}^{n-1} \frac{\boldsymbol{\partial F}}{\partial y_i} F_i(x,\boldsymbol{y})\right) + O(h^3)$
- IF:

$$c_0 + c_1 = 1 \quad ; \quad c_1 p = \frac{1}{2} \quad ; \quad c_1 q = \frac{1}{2}$$

Runge-Kutta (2nd order)

$$c_0 + c_1 = 1 \quad ; \quad c_1 p = \frac{1}{2} \quad ; \quad c_1 q = \frac{1}{2}$$

- This system as an infinite number of solutions (4unknowns and 3eq.)
- Some popular choices for the parameters are:

$c_0 = 0$	$c_1 = 1$	$p = 1/2$	$q = 1/2$	Modified Euler's method
$c_0 = 1/2$	$c_1 = 1/2$	$p = 1$	$q = 1$	Heun's method
$c_0 = 1/3$	$c_1 = 2/3$	$p = 3/4$	$q = 3/4$	Ralston's method

- All these methods are of the second order
  - They are all equivalent
  - They all have a local error that is of $O(h^3)$

Modified Euler's method

- *RK2:*
  - $\boldsymbol{y}(x+h) \approx \boldsymbol{y}(x) + c_0 h\boldsymbol{F}(x,\boldsymbol{y}) + \boldsymbol{c_1} h\boldsymbol{F}\left(x+ph\,,\boldsymbol{y}+qh\,\boldsymbol{F}(x,\boldsymbol{y})\right)$
- *Modified Euler's method:*

$$c_0 = 0 \quad ; \quad c_1 = 1 \quad ; \quad p = \frac{1}{2} \quad ; \quad q = \frac{1}{2}$$

  - $\boldsymbol{y}(x+h) \approx \boldsymbol{y}(x) + h\boldsymbol{F}\left(x+\frac{h}{2}\,,\boldsymbol{y}+\frac{h}{2}\,\boldsymbol{F}(x,\boldsymbol{y})\right)$
  - *Also called MidPoint method*
  - Take the tangent of F at midpoint to propagate y
  - Pseudo Algorithm:

$$\mathbf{K}_0 = h\mathbf{F}(x, \mathbf{y})$$

$$\mathbf{K}_1 = h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{1}{2}\mathbf{K}_0\right)$$

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \mathbf{K}_1$$

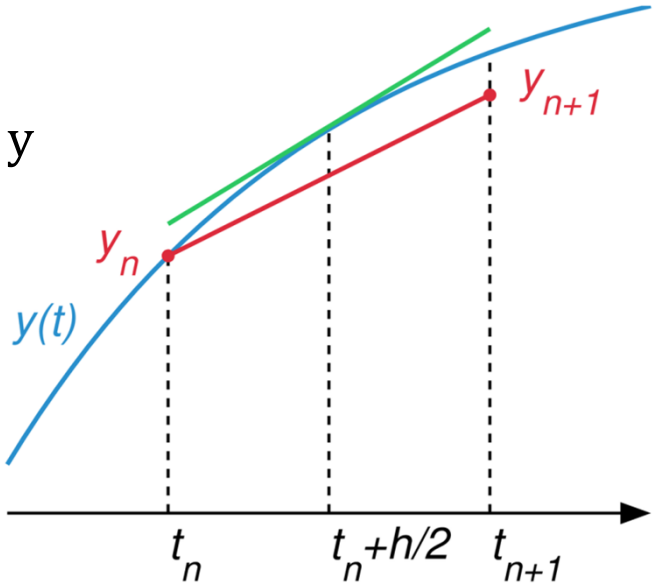


Illustration from Wikipedia

Relation with Integration method

- In the special case where $F(x, y)$ does not depend on y
  - $F(x, y) = F(x)$
- The problem simplifies to:
  - $y'(x) = F(x)$
  - $y(x_n) = y_n$
- The solution to the problem is given by:
  - $y(x_{n+1}) = y(x_n) + \int_{x_n}^{x_{n+1}} f(x)dx$

- If we compare with the RK2 solution in the case $F(x, y) = F(x)$:
  - Modified Euler's method:
    - $y(x_{n+1}) \approx y(x_n) + hF\left(x_n + \frac{h}{2}\right)$ **Midpoint integral**
  - *Heun's method*
    - $y(x_{n+1}) \approx y(x_n) + \frac{h}{2}[F(x_n) + F(x_n + h)]$ **Trapezoid integral**

Runge-Kutta (4th order)

- Runge-Kutta of 4th order (RK4) is obtain like we did for RK2
  - Matching to order 4 Taylor expansion
  - This requires a lot of algebra and is not particularly interesting
  - Let's take it for granted:

$$\mathbf{K}_0 = h\mathbf{F}(x, \mathbf{y})$$

Slope at the beginning of the interval

$$\mathbf{K}_1 = h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{\mathbf{K}_0}{2}\right)$$

Slope at the midpoint of the interval (y is calculated from k0)

$$\mathbf{K}_2 = h\mathbf{F}\left(x + \frac{h}{2}, y + \frac{\mathbf{K}_1}{2}\right)$$

Slope at the midpoint of the interval (y is calculated from k1)

$$\mathbf{K}_3 = h\mathbf{F}(x + h, \mathbf{y} + \mathbf{K}_2)$$

Slope at the end of the interval (y is calculated from k2)

$$\mathbf{y}(x + h) = \mathbf{y}(x) + \frac{1}{6}(\mathbf{K}_0 + 2\mathbf{K}_1 + 2\mathbf{K}_2 + \mathbf{K}_3)$$

- The local truncation error is of $O(h^5)$

Runge-Kutta (4th order)

$$\mathbf{K}_0 = h\mathbf{F}(x, \mathbf{y})$$

$$\mathbf{K}_1 = h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{\mathbf{K}_0}{2}\right)$$

$$\mathbf{K}_2 = h\mathbf{F}\left(x + \frac{h}{2}, y + \frac{\mathbf{K}_1}{2}\right)$$

$$\mathbf{K}_3 = h\mathbf{F}(x + h, \mathbf{y} + \mathbf{K}_2)$$

$$\mathbf{y}(x + h) = \mathbf{y}(x) + \frac{1}{6}(\mathbf{K}_0 + 2\mathbf{K}_1 + 2\mathbf{K}_2 + \mathbf{K}_3)$$

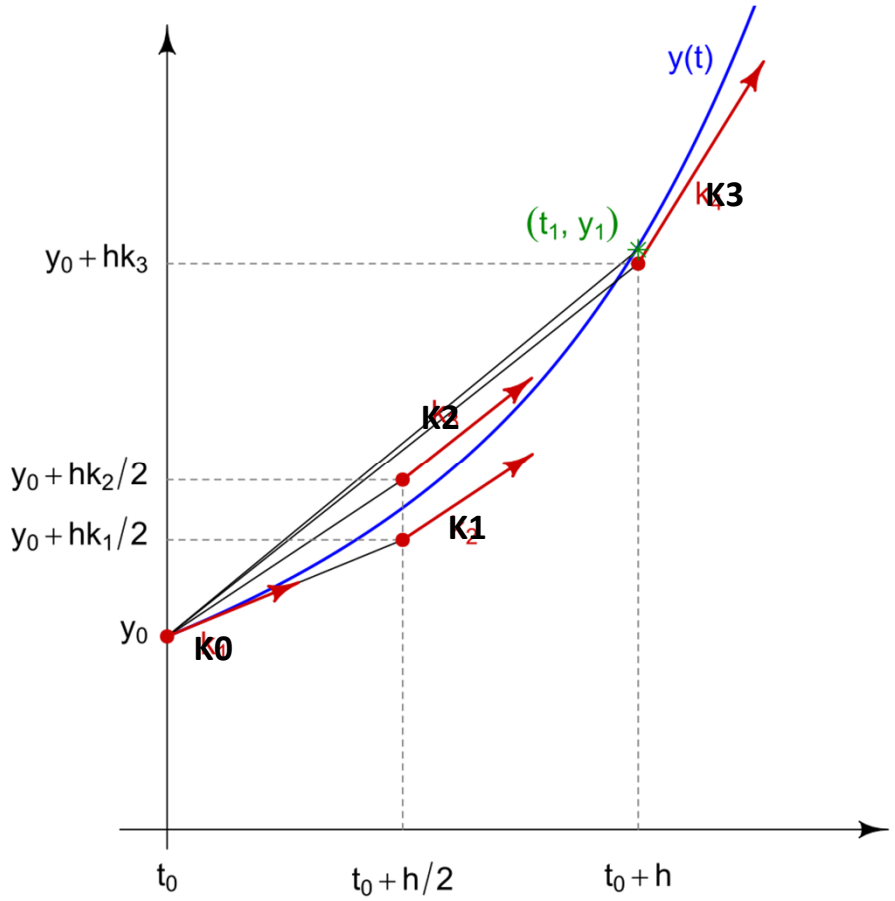


Illustration from Wikipedia

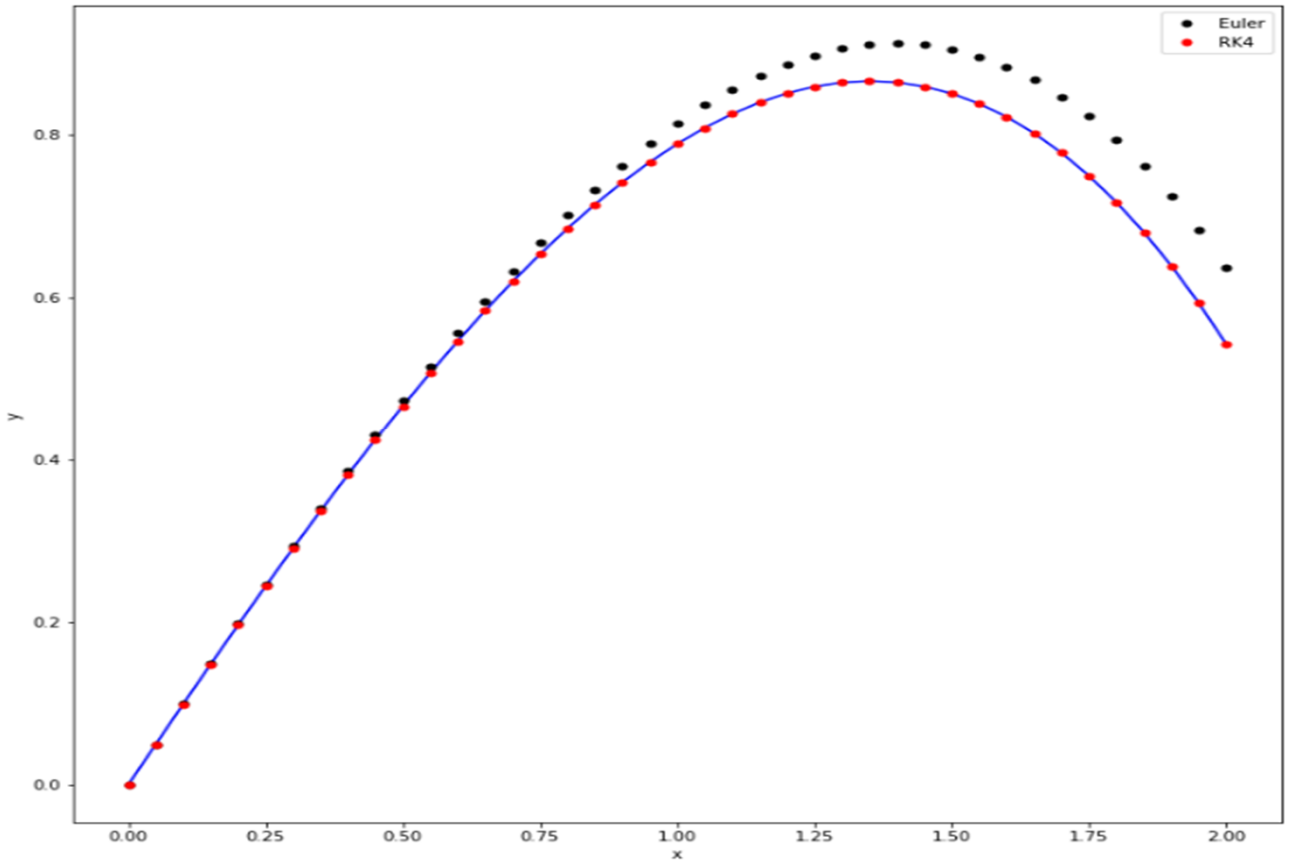
Algorithm

- **Can you implement the RK4 Algorithm ?**

```
import numpy as np
def integrate(F,x,y,xStop,h):

    def run_kut4(F,x,y,h):
        K0 = h*F(x,y)
        K1 = h*F(x + h/2.0, y + K0/2.0)
        K2 = h*F(x + h/2.0, y + K1/2.0)
        K3 = h*F(x + h, y + K2)
        return (K0 + 2.0*K1 + 2.0*K2 + K3)/6.0

    X = []
    Y = []
    X.append(x)
    Y.append(y)
    while x < xStop:
        h = min(h,xStop - x)
        y = y + run_kut4(F,x,y,h)
        x = x + h
        X.append(x)
        Y.append(y)
    return np.array(X),np.array(Y)
```

Example

- Solve the initial value problem given by:
- $y'' = 0.1y' - x$ and $y(0) = 0$ and $y'(0) = 1$
- Use RK4 in steps of h=0.05 from x=0 to 2



Example 2 (1)

- Solve the initial value problem given by:
- $y' = 3y - 4e^{-x}$ and $y(0) = 1$
- Use RK4 in steps of h=0.1 from x=0 to 10
- Compare with the analytical solution $y = e^{-x}$

```
def F(x,y):
    F = np.zeros(1)
    F[0] = 3.0*y[0] - 4.0*exp(-x)
    return F
```

RK4 →

x	y[0]
0.0000e+000	1.0000e+000
2.0000e+000	1.3250e-001
4.0000e+000	-1.1237e+000
6.0000e+000	-4.6056e+002
8.0000e+000	-1.8575e+005
1.0000e+001	-7.4912e+007

- From the analytical solution, we would expect $y \to 0$ for $x \to \infty$

- **Something goes wrong!**

Example 2 (2)

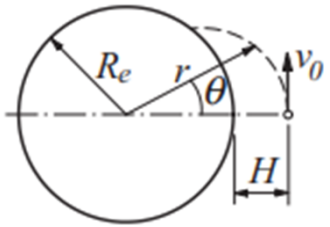
- Actually $y = e^{-x}$ is **not the only solution** of ODE $y' = 3y - 4e^{-x}$
- The general form of solution is:
  - $y = Ce^{3x} + e^{-x}$
  - From derivate: $y' = 3Ce^{3x} - e^{-x}$
  - From ODE: $y' = 3y - 4e^{-x} = 3(Ce^{3x} + e^{-x}) - 4e^{-x} = 3Ce^{3x} - e^{-x}$

- Initial condition: $y(0) = 1$
- $\rightarrow C = 0$ → So the only solution of the IVP is indeed $y = e^{-x}$
- But imagine we have a small error (roundoff) in the initial condition
  - $y(0) = 1 + \epsilon$

- In that case, the solution is $y = \epsilon e^{3x} + e^{-x}$
  - When x get large, the term in ϵ become dominant
- We are victim of **numerical instabilities** due to sensibility to initial condition

→ **DO NOT TRUST BLINDLY THE RESULTS OF AN IVP**

Example 3 (1)



A spacecraft is launched at the altitude $H = 772$ km above sea level with the speed $v_0 = 6700$ m/s in the direction shown. The differential equations describing the motion of the spacecraft are

$$\ddot{r} = r\dot{\theta}^2 - \frac{GM_e}{r^2} \qquad \ddot{\theta} = -\frac{2\dot{r}\dot{\theta}}{r}$$

where r and θ are the polar coordinates of the spacecraft. The constants involved in the motion are

$$G = 6.672 \times 10^{-11}\ \text{m}^3\,\text{kg}^{-1}\text{s}^{-2} = \text{universal gravitational constant}$$

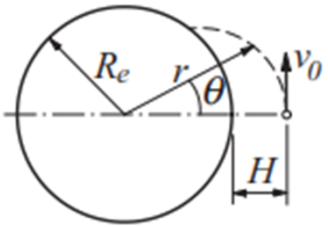
$$M_e = 5.9742 \times 10^{24}\ \text{kg} = \text{mass of the earth}$$

$$R_e = 6378.14\ \text{km} = \text{radius of the earth at sea level}$$

(1) Derive the first-order differential equations and the initial conditions of the form $\dot{\mathbf{y}} = \mathbf{F}(t, \mathbf{y})$, $\mathbf{y}(0) = \mathbf{b}$. (2) Use the fourth-order Runge-Kutta method to integrate the equations from the time of launch until the spacecraft hits the earth. Determine θ at the impact site.

Example 2 (2)

$$GM_e = \left(6.672 \times 10^{-11}\right)\left(5.9742 \times 10^{24}\right) = 3.9860 \times 10^{14}\ \mathrm{m^3\,s^{-2}}$$



$$\mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} r \\ \dot{r} \\ \theta \\ \dot{\theta} \end{bmatrix}$$

Notes that we store the two variables and their derivate in the vector y

$$\mathbf{F}(t, \mathbf{y}) = \begin{bmatrix} \dot{y}_0 \\ \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_0 y_3^2 - 3.9860 \times 10^{14}/y_0^2 \\ y_3 \\ -2y_1 y_3/y_0 \end{bmatrix}$$

This is the ODE

Initial Conditions:

$$r(0) = R_e + H = (6378.14 + 772) \times 10^3 = 7.15014 \times 10^6\ \mathrm{m}$$

$$\dot{r}(0) = 0$$

$$\theta(0) = 0$$

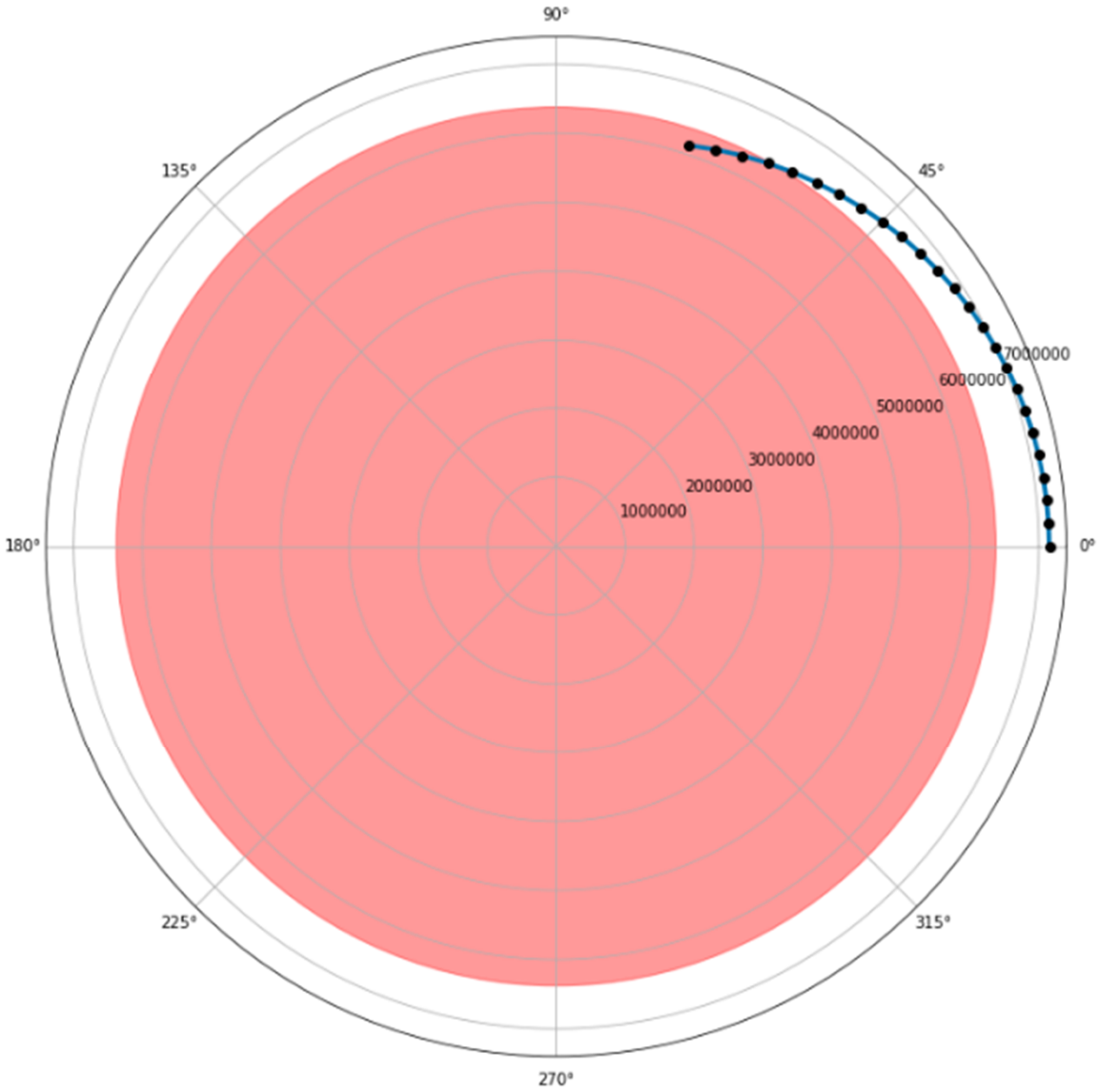
$$\dot{\theta}(0) = v_0/r(0) = (6700)\,/(7.15014 \times 10^6) = 0.937045 \times 10^{-3}\ \mathrm{rad/s}$$

$$\mathbf{y}(0) = \begin{bmatrix} 7.15014 \times 10^6 \\ 0 \\ 0 \\ 0.937045 \times 10^{-3} \end{bmatrix}$$

Example 3 (3)

x	y[0]	y[1]	y[2]	y[3]
0.0000e+000	7.1501e+006	0.0000e+000	0.0000e+000	9.3704e-004
1.0000e+002	7.1426e+006	-1.5173e+002	9.3771e-002	9.3904e-004
2.0000e+002	7.1198e+006	-3.0276e+002	1.8794e-001	9.4504e-004
3.0000e+002	7.0820e+006	-4.5236e+002	2.8292e-001	9.5515e-004
4.0000e+002	7.0294e+006	-5.9973e+002	3.7911e-001	9.6951e-004
5.0000e+002	6.9622e+006	-7.4393e+002	4.7697e-001	9.8832e-004
6.0000e+002	6.8808e+006	-8.8389e+002	5.7693e-001	1.0118e-003
7.0000e+002	6.7856e+006	-1.0183e+003	6.7950e-001	1.0404e-003
8.0000e+002	6.6773e+006	-1.1456e+003	7.8520e-001	1.0744e-003
9.0000e+002	6.5568e+006	-1.2639e+003	8.9459e-001	1.1143e-003
1.0000e+003	6.4250e+006	-1.3708e+003	1.0083e+000	1.1605e-003
1.1000e+003	6.2831e+006	-1.4634e+003	1.1269e+000	1.2135e-003
1.2000e+003	6.1329e+006	-1.5384e+003	1.2512e+000	1.2737e-003

Example 3 (4)


90°
135°
45°
180°
0°
225°
315°
270°
1000000
2000000
3000000
4000000
5000000
6000000
7000000

Example 3 (5)

x	y[0]	y[1]	y[2]	y[3]
0.0000e+000	7.1501e+006	0.0000e+000	0.0000e+000	9.3704e-004
1.0000e+002	7.1426e+006	-1.5173e+002	9.3771e-002	9.3904e-004
2.0000e+002	7.1198e+006	-3.0276e+002	1.8794e-001	9.4504e-004
3.0000e+002	7.0820e+006	-4.5236e+002	2.8292e-001	9.5515e-004
4.0000e+002	7.0294e+006	-5.9973e+002	3.7911e-001	9.6951e-004
5.0000e+002	6.9622e+006	-7.4393e+002	4.7697e-001	9.8832e-004
6.0000e+002	6.8808e+006	-8.8389e+002	5.7693e-001	1.0118e-003
7.0000e+002	6.7856e+006	-1.0183e+003	6.7950e-001	1.0404e-003
8.0000e+002	6.6773e+006	-1.1456e+003	7.8520e-001	1.0744e-003
9.0000e+002	6.5568e+006	-1.2639e+003	8.9459e-001	1.1143e-003
1.0000e+003	6.4250e+006	-1.3708e+003	1.0083e+000	1.1605e-003
1.1000e+003	6.2831e+006	-1.4634e+003	1.1269e+000	1.2135e-003
1.2000e+003	6.1329e+006	-1.5384e+003	1.2512e+000	1.2737e-003

The spacecraft hits the earth when r equals $R_e = 6.378\,14 \times 10^6$ m. This occurs between $t = 1000$ and 1100 s. A more accurate value of t can be obtained by polynomial interpolation. If no great precision is needed, linear interpolation will do. Letting $1000 + \Delta t$ be the time of impact, we can write

$$r(1000 + \Delta t) = R_e$$

Expanding r in a two-term Taylor series, we get

$$r(1000) + r'(1000)\Delta t = R_e$$

$$6.4250 \times 10^6 + \left(-1.3708 \times 10^3\right) x = 6378.14 \times 10^3$$

from which

$$\Delta t = 34.184 \text{ s}$$

N-Body problem (1)

- One typical use-case of ODE / IVP is the N-Body problem
- Movement of N-Bodies with a given mass and Gravity interactions
- Could be solved analytically for N=2, but not easily solvable for N>2

- ODE is given by Newton's gravity law (as in example 3)
  - Each body (i) is attracted by the other bodies ($j \neq i$) with a force in the direction of the body

$$F_{i,j} = \frac{G\, m_i m_j}{{d_{i,j}}^2}$$

  - Because of Newton Law ($F = ma$), the total acceleration of body i is :

$$a_i = \sum_{j \neq i} G \frac{m_j}{{d_{i,j}}^2}$$

N-Body problem (2)

- The ODE is then defined by

$$\dot{x}_i = v_i$$

$$\dot{v}_i = a_i = \sum_{j \neq i} G \frac{m_j}{{d_{i,j}}^2}$$

- The IVP takes an initial condition on x_i and v_i in addition

- We can play with this in 2 dimensions,
  where the position and speed are defined by two Cartesian components

- And see evolution as a function of time.

2-Body simulation (1)

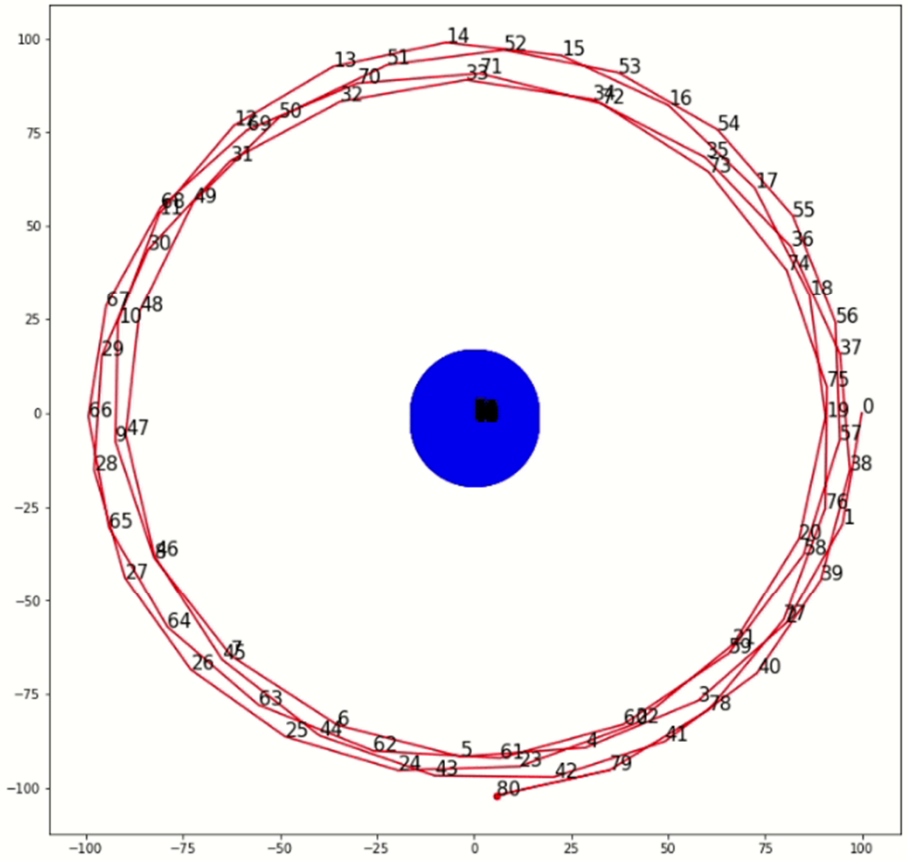
```
#y = (x_0, y_0, x_1, y_1, dx_0, dy_0, dx_1, dy_1)

G   = 1
m_0 = 10.0
m_1 = 0.01

def F(t,y):
    x_0, y_0, x_1, y_1 = y[0:4]
    dist = np.sqrt((x_0-x_1)**2 + (y_0-y_1)**2 )

    F = np.zeros(8)
    F[0:4] = y[4:8]
    F[4]   = -G*m_1*(x_0-x_1)/dist**2
    F[5]   = -G*m_1*(y_0-y_1)/dist**2
    F[6]   = -G*m_0*(x_1-x_0)/dist**2
    F[7]   = -G*m_0*(y_1-y_0)/dist**2
    return F

#x = time
x_start = 0.0
x_stop  = 800.0
y_start = np.array([   0.0,  0.0, #0 x-y position
                     100.0,  0.0, #1 x-y position
                       0.0,  0.0, #0 x-y speed
                       0.0, -3.0  #1 x-y speed
                   ])
h       = 10.0
results_RK4 = integrate_RK4  (F, x_start, y_start, x_stop, h)
showResults(results_RK4, m_0, m_1, animate=True)
```

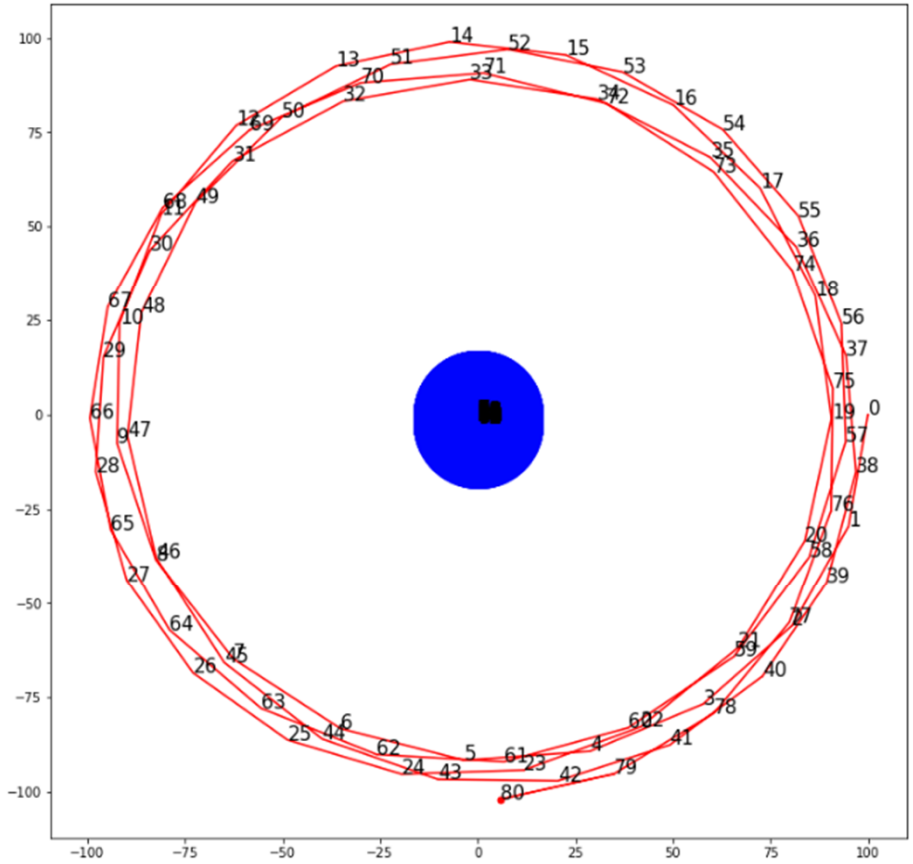
- Store all the pos and speed in y

```
y = [x_0, y_0,       #body 0 pos
     x_1, y_1,       #body 1 pos
     vx_0, vy_0,     #body 0 speed
     vx_1, vy_1]     #body 1 speed
```

- F is given by the ODE (movement equations)
  - G = 1.0 #dummy unit
  - m_0=10.0 #body 0 mass
  - m_1=0.01 #body 1 mass
- Initial conditions
- Simulate from t=0 to 800 in steps of 10

2-Body simulation (1)





m0 = 10.0 **m1 = 0.1**

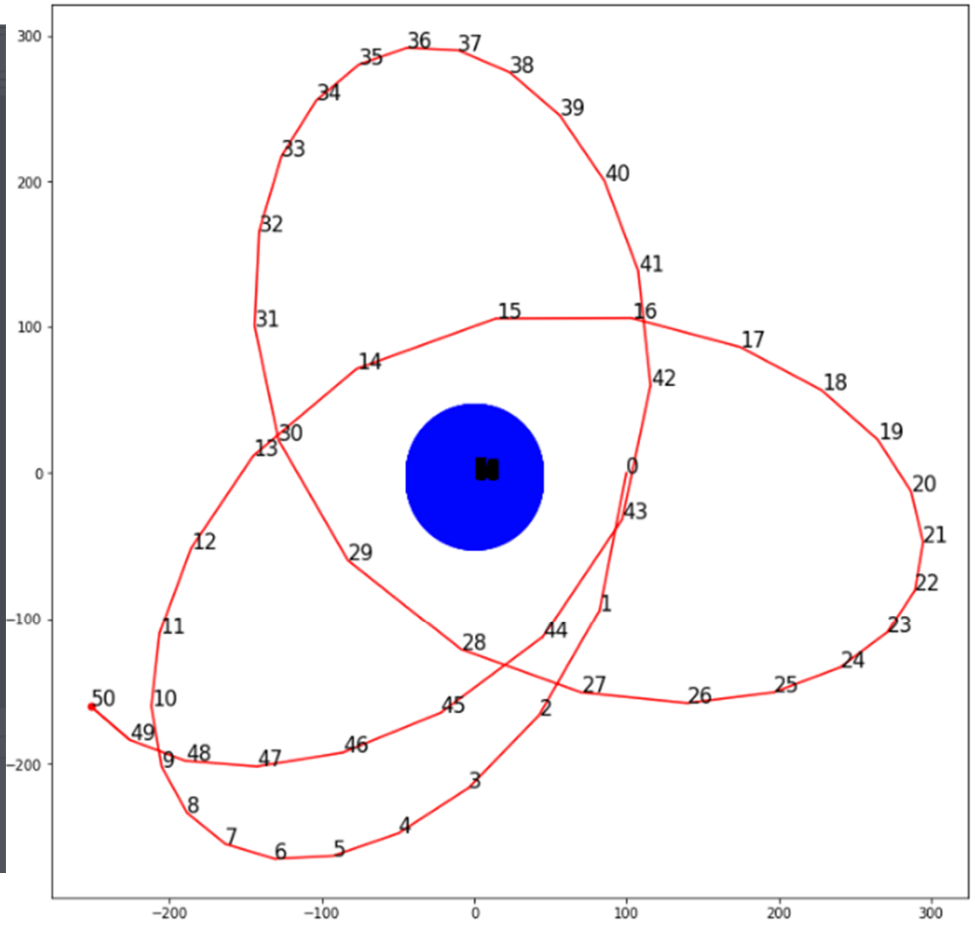
Similar to a Solar System: planet orbiting around a star

2-Body simulation (3)

```
## deux corps (même masse en orbite)

#y = (x_0, y_0, x_1, y_1, dx_0, dy_0, dx_1, dy_1)

G   = 1
m_0 = 1.0
m_1 = 1.0

def F(t,y):
    x_0, y_0, x_1, y_1 = y[0:4]
    dist = np.sqrt((x_0-x_1)**2 + (y_0-y_1)**2 )

    F = np.zeros(8)
    F[0:4] = y[4:8]
    F[4]   = -G*m_1*(x_0-x_1)/dist**2
    F[5]   = -G*m_1*(y_0-y_1)/dist**2
    F[6]   = -G*m_0*(x_1-x_0)/dist**2
    F[7]   = -G*m_0*(y_1-y_0)/dist**2
    return F

#x = time
x_start = 0.0
x_stop  = 2000.0
y_start = np.array([-100.0,  0.0, #0 x-y position
                    100.0,  0.0, #1 x-y position
                      0.0, 0.70, #0 x-y speed
                      0.0,-0.70  #1 x-y speed
                   ])
h       = 20.0
freq    = 2

results_RK4 = integrate_RK4  (F, x_start, y_start, x_stop, h)
showResults(results_RK4, m_0, m_1, animate=True)
```

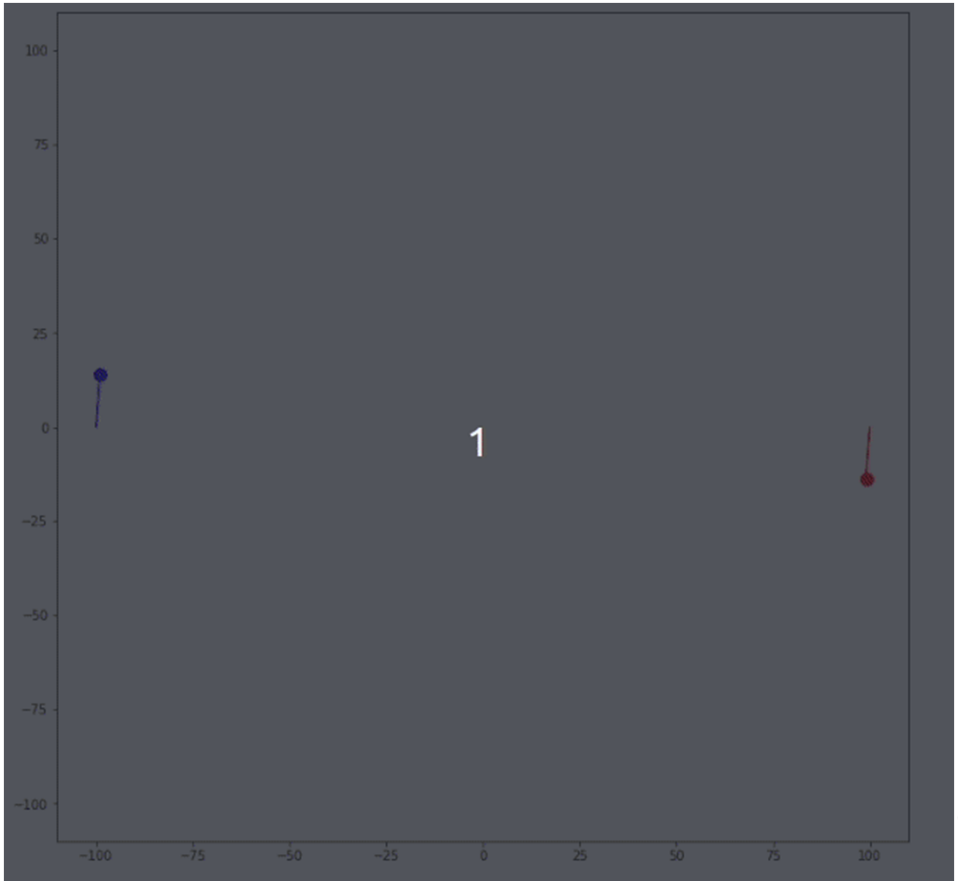
1

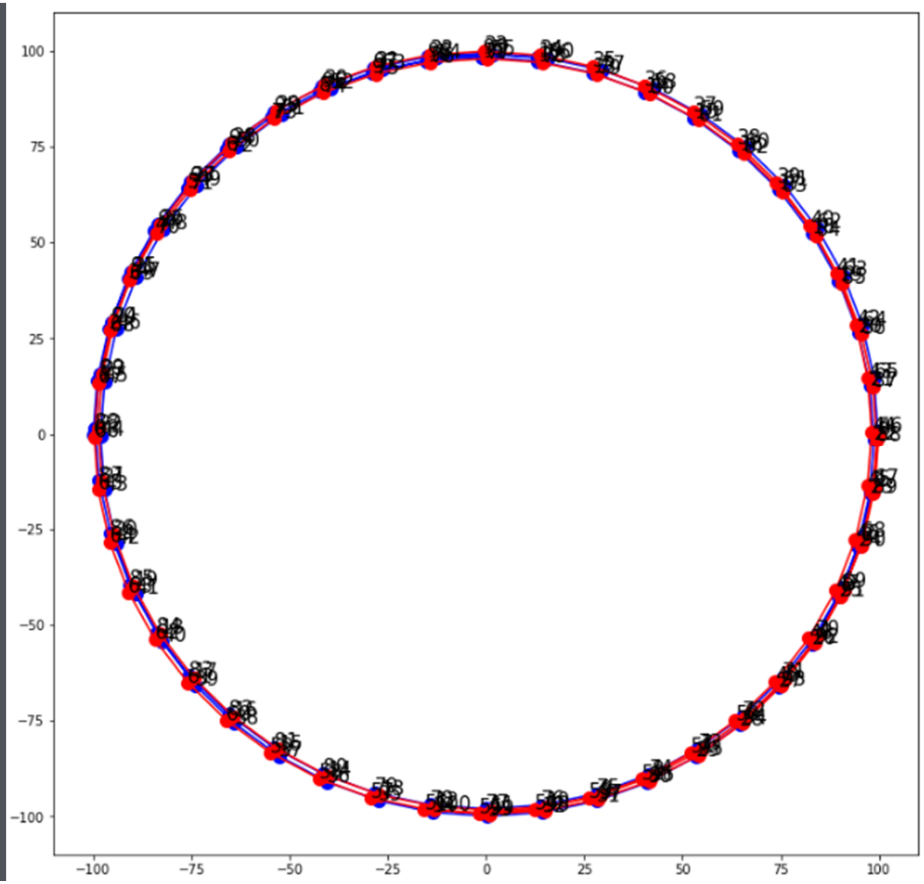
executed in 34.4s, finished 08:11:15 2019-11-15



Changing a bit the initial condition → make the system less trivial
Now we are more in an eliptic movement (like a comet)

2-Body simulation (4)



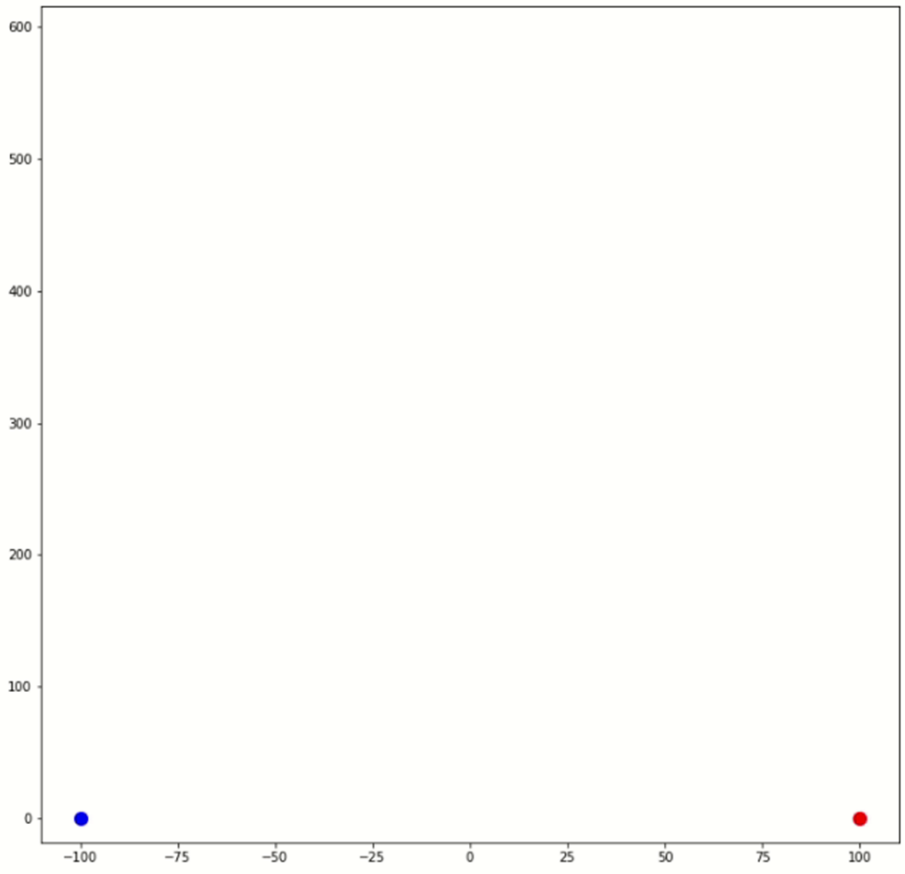


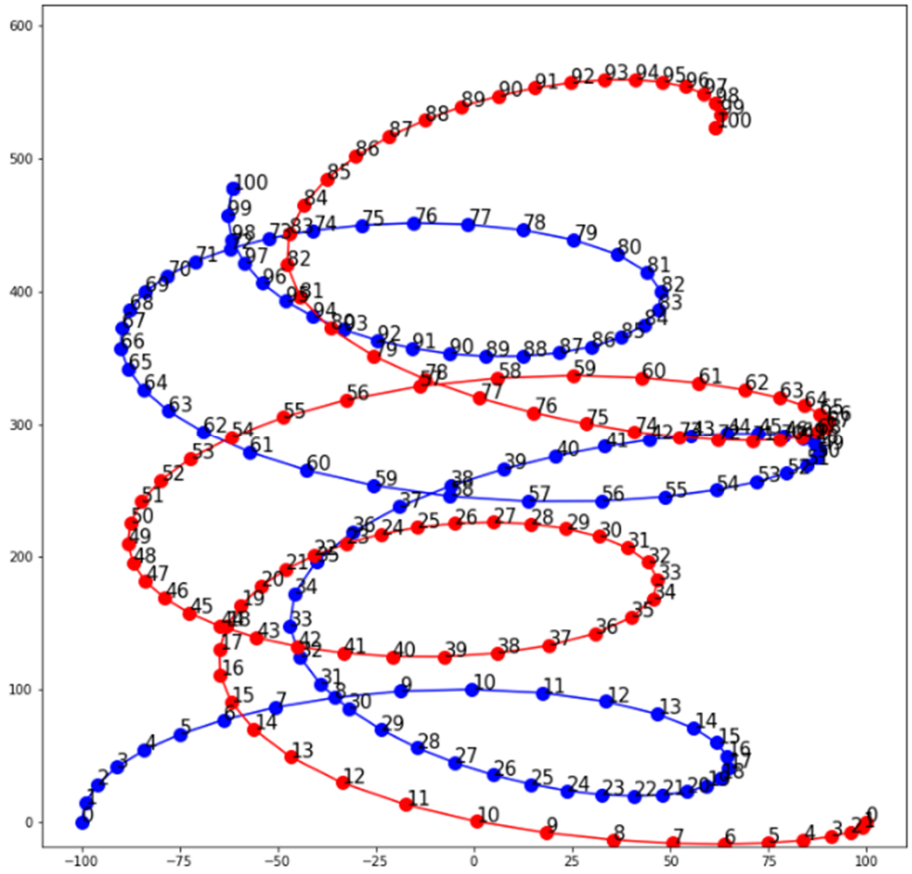
m0 = 1.0 **m1 = 1.0**

Let's try with symmetric masses and positions (and appropriate initial speed conditions)

→ Binary stars/blackholes system

2-Body simulation (5)





m0 = 1.0 **m1 = 1.0**

Same mass, but less ad-hoc initial conditions

→ Probably what's happening most of the time in space… chaos…

3-Body simulation (1)

```
G   = 1
m_0 = 10.0
m_1 = 0.01
m_2 = 0.01

def F(t,y):
    x_0, y_0, x_1, y_1, x_2, y_2 = y[0:6]
    dist_01 = np.sqrt((x_0-x_1)**2 + (y_0-y_1)**2 )
    dist_02 = np.sqrt((x_0-x_2)**2 + (y_0-y_2)**2 )
    dist_12 = np.sqrt((x_1-x_2)**2 + (y_1-y_2)**2 )

    F = np.zeros(12)
    F[0:6] = y[6:12]
    F[6]   = -G*m_1*(x_0-x_1)/dist_01**2 + -G*m_2*(x_0-x_2)/dist_02**2
    F[7]   = -G*m_1*(y_0-y_1)/dist_01**2 + -G*m_2*(y_0-y_2)/dist_02**2
    F[8]   = -G*m_0*(x_1-x_0)/dist_01**2 + -G*m_2*(x_1-x_2)/dist_02**2
    F[9]   = -G*m_0*(y_1-y_0)/dist_01**2 + -G*m_2*(y_1-y_2)/dist_02**2
    F[10]  = -G*m_0*(x_2-x_0)/dist_02**2 + -G*m_1*(x_2-x_1)/dist_12**2
    F[11]  = -G*m_0*(y_2-y_0)/dist_02**2 + -G*m_1*(y_2-y_1)/dist_12**2
    return F

#x = time
x_start = 0.0
x_stop  = 800.0
y_start = np.array([   0.0,  0.0, #0 x-y position
                     100.0,  0.0, #1 x-y position
                     300.0,  0.0, #2 x-y position
                       0.0,  0.0, #0 x-y speed
                       0.0, -3.0, #1 x-y speed
                       0.0, +3.0  #2 x-y speed
                   ])
h       = 10.0
results_RK4 = integrate_RK4  (F, x_start, y_start, x_stop, h)
showResults(results_RK4, m_0, m_1, m_2, animate=True)
```

- Store all the pos and speed in y

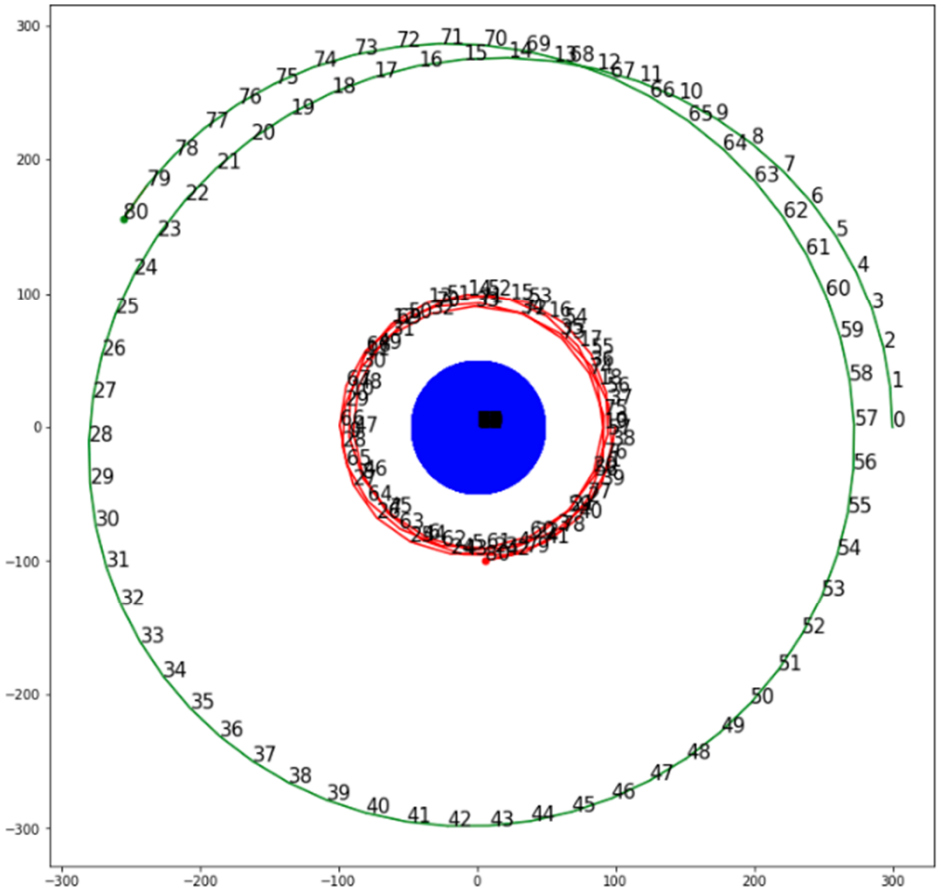
```
y = [x_0, y_0,      #body 0 pos
     x_1, y_1,      #body 1 pos
     x_2, y_2,      #body 2 pos
     vx_0, vy_0,    #body 0 speed
     vx_1, vy_1,    #body 1 speed
     vx_2, vy_2]    #body 2 speed
```

- F is given by the ODE (movement equations)
  - Note that they are **<u>3 distances</u>** involves now
- Initial conditions
- Simulate from t=0 to 800 in steps of 10

3-Body simulation (2)

```
m_0 = 10.0
m_1 = 0.01
m_2 = 0.01

def F(t,y):
    x_0, y_0, x_1, y_1, x_2, y_2 = y[0:6]
    dist_01 = np.sqrt((x_0-x_1)**2 + (y_0-y_1)**2 )
    dist_02 = np.sqrt((x_0-x_2)**2 + (y_0-y_2)**2 )
    dist_12 = np.sqrt((x_1-x_2)**2 + (y_1-y_2)**2 )

    F = np.zeros(12)
    F[0:6] = y[6:12]
    F[6]   = -G*m_1*(x_0-x_1)/dist_01**2 + -G*m_2*(x_0-x_2)/dist_02**2
    F[7]   = -G*m_1*(y_0-y_1)/dist_01**2 + -G*m_2*(y_0-y_2)/dist_02**2
    F[8]   = -G*m_0*(x_1-x_0)/dist_01**2 + -G*m_2*(x_1-x_2)/dist_02**2
    F[9]   = -G*m_0*(y_1-y_0)/dist_01**2 + -G*m_2*(y_1-y_2)/dist_02**2
    F[10]  = -G*m_0*(x_2-x_0)/dist_02**2 + -G*m_1*(x_2-x_1)/dist_12**2
    F[11]  = -G*m_0*(y_2-y_0)/dist_02**2 + -G*m_1*(y_2-y_1)/dist_12**2
    return F

#x = time
x_start = 0.0
x_stop  = 800.0
y_start = np.array([   0.0,  0.0, #0 x-y position
                     100.0,  0.0, #1 x-y position
                     300.0,  0.0, #2 x-y position
                       0.0,  0.0, #0 x-y speed
                       0.0, -3.0, #1 x-y speed
                       0.0, +3.0  #2 x-y speed
                   ])
h       = 10.0
freq    = 2

results_RK4 = integrate_RK4  (F, x_start, y_start, x_stop, h)
showResults(results_RK4, m_0, m_1, m_2, animate=True)
```

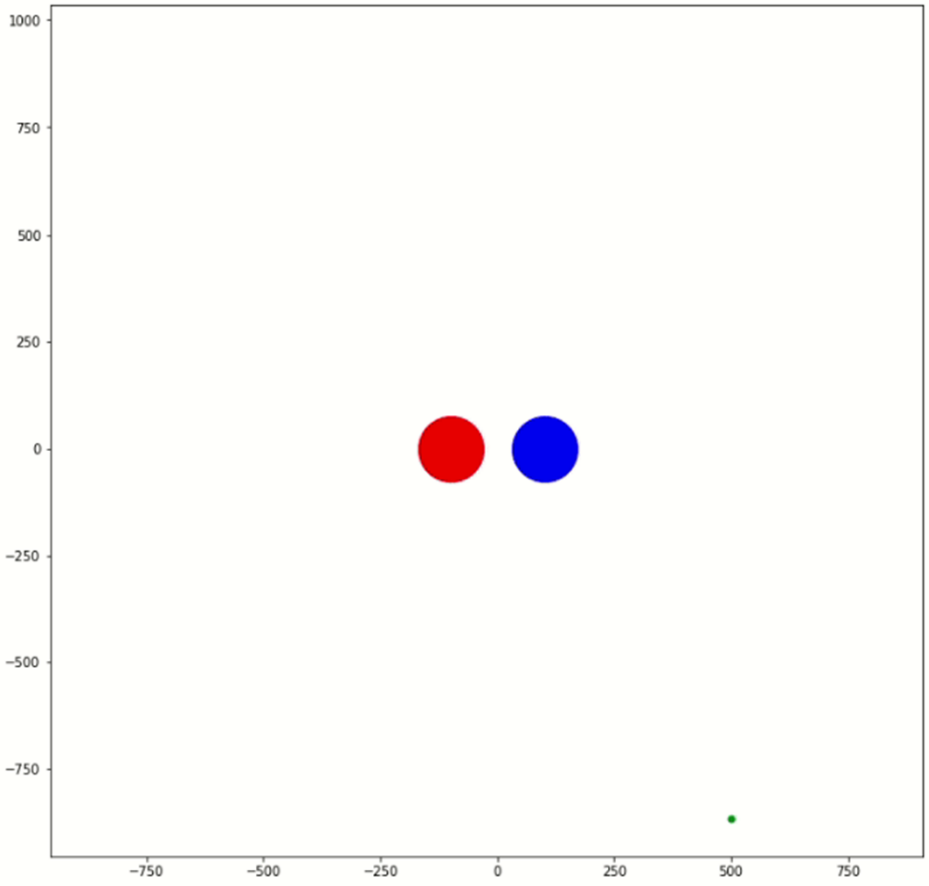
execution queued 08:48:52 2019-11-15

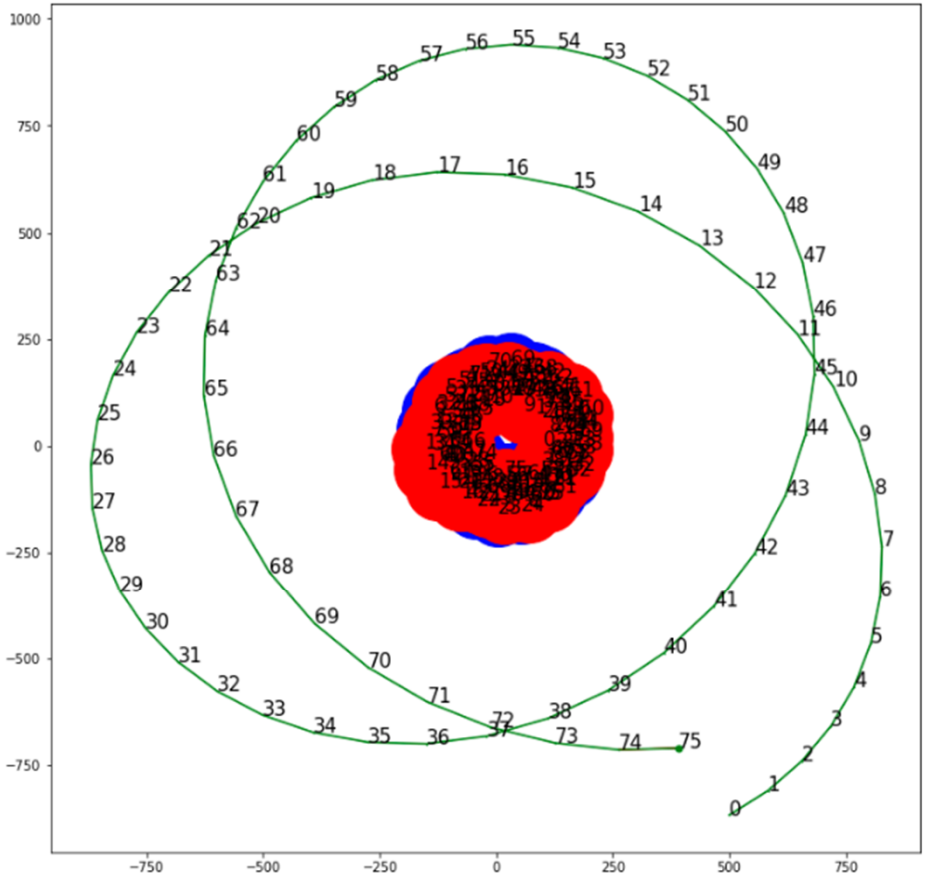


m0 = 10.0 **m1 = 0.1** **m2 = 0.1**

→ Orbital system with sun and two planets on distinct orbits

3-Body simulation (3)

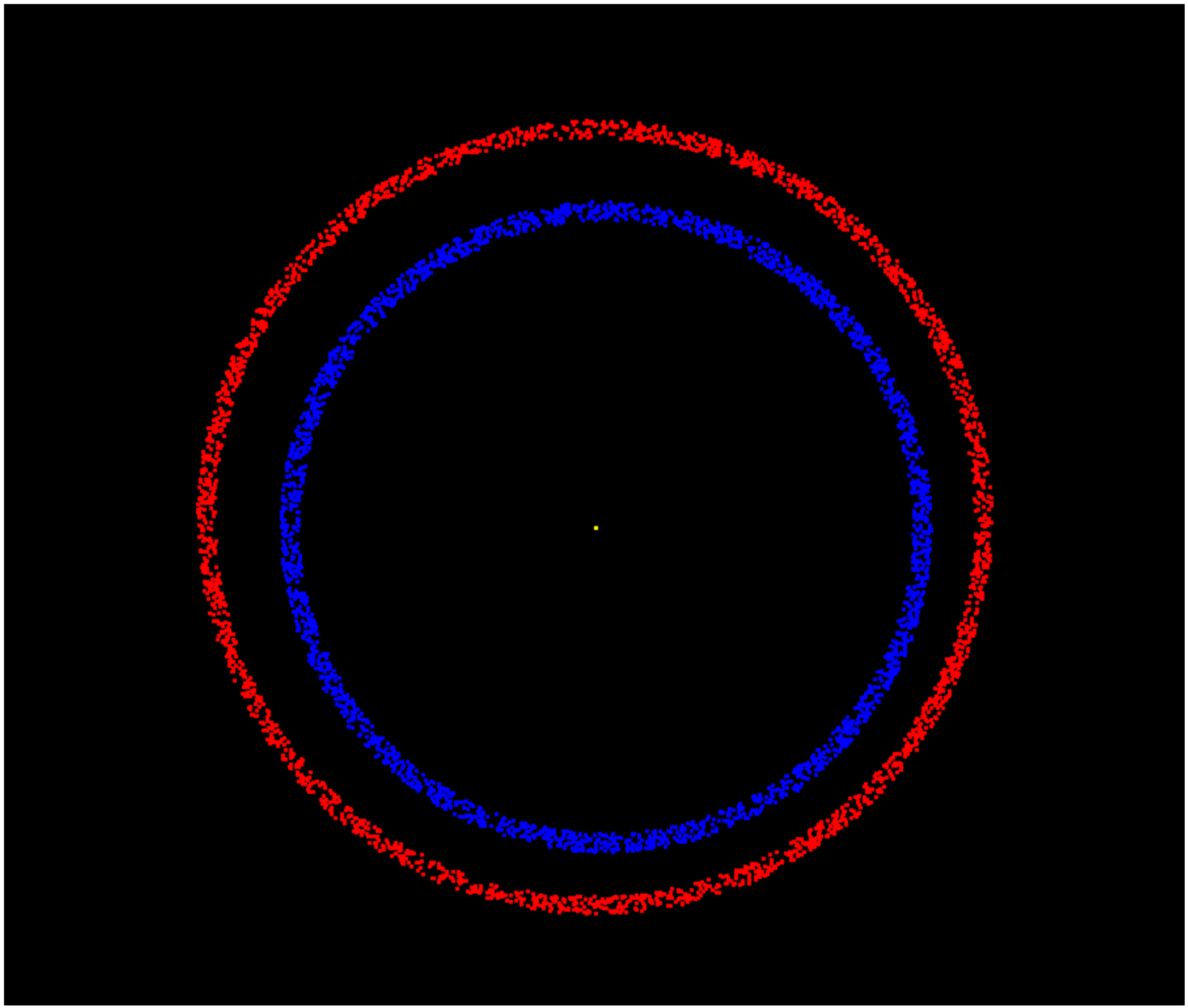




m0 = 5.0 **m1 = 5.0** **m2 = 0.05**

→ One planet orbiting around a binary star system

1+5000-Body simulation



→ **Two asteroid belts orbiting around a star (with imperfect initial conditions)**

Stability and Stiffness

Stability

- A method of numerical integration is **stable** if
  - The effects of local errors do not accumulate catastrophically;
  - The global error remains bounded.
  - If the method is unstable, the global error will increase exponentially, eventually causing numerical overflow.

- Stability has nothing to do with accuracy;
  - An inaccurate method can be very stable.

- Stability is determined by three factors:
  - the differential equations,
  - the method of solution,
  - the value of the increment h.

- Unfortunately, it is not easy to determine stability beforehand, unless the differential equation is linear

Stability of Euler's Method

- Consider the IVP:

$$\boldsymbol{y' = -\lambda\, y} \text{ and } \boldsymbol{y(0) = \beta} \quad \text{with } \boldsymbol{\lambda > 0}$$

- The analytical solution is given by

$$\boldsymbol{y(x) = \beta e^{-\lambda x}}$$

- Let solve the system via Euler's method:
  - $y(x + h) = y(x) + hy'(x)$
  - Using $y' = -\lambda\, y$, we get:
  - $\boldsymbol{y(x + h) = y(x)(1 - \lambda h)}$

- If $|1 - \lambda h| > 1$ the method is clearly unstable, because y(x) increases at every iteration

- **Euler method is stable only if $\boldsymbol{h \leq \frac{2}{\lambda}}$**

Stiffness

- An IVP is **stiff** if
  - some terms in the solution vector $\boldsymbol{y}(x)$ much more rapidly with x than others.

- Stiffness can be easily predicted for the differential equations $\boldsymbol{y'} = -\boldsymbol{\Lambda y}$ with constant coefficient matrix $\boldsymbol{\Lambda}$.
  - The solution of these equations is $\boldsymbol{y(x)} = \sum_i C_i \boldsymbol{v_i} \exp(-\boldsymbol{\lambda_i} x)$, where λ_i are the eigenvalues of $\boldsymbol{\Lambda}$ and $\boldsymbol{v_i}$ are the corresponding eigenvectors.
  - It is evident that the problem is stiff if there is a large disparity in the magnitudes of the positive eigenvalues.
- Numerical integration of stiff equations requires special care.
- The step size h needed for stability is determined by the largest eigenvalue $\boldsymbol{\lambda_{max}}$, even if the terms exp($-\boldsymbol{\lambda_{max}} x$) in the solution decay very rapidly and become insignificant as we move away from the origin

Stiffness Example

- Consider the ODE
  - $y'' + 1001y' + 1000' = 0$
  - Equivalent to:

$$\boldsymbol{y}' = \begin{pmatrix} y_1 \\ -1000y_0 - 1001\, y_1 \end{pmatrix}$$

- In this case:

$$\boldsymbol{\Lambda} = \begin{pmatrix} \mathbf{0} & \mathbf{-1} \\ \mathbf{1000} & \mathbf{1001} \end{pmatrix}$$

- The eigen values are the roots of

$$|\boldsymbol{\Lambda} - \lambda \boldsymbol{I}| = \begin{vmatrix} \boldsymbol{-\lambda} & \mathbf{-1} \\ \mathbf{1000} & \mathbf{1001} - \boldsymbol{\lambda} \end{vmatrix} = \mathbf{0}$$

- The determinant is given by:

$$-\lambda(1001 - \lambda) + 1000 = 0$$

- Solutions are : $\lambda_1 = 1$ and $\lambda_2 = 1000$
- For stability we would need to take $h \leq 2/\lambda_2 = 0.002$
- Problem:
  - This is a very small h → not quite usable in practice (for both Euler and RK)
- They are dedicated techniques and solvers for Stiff IVP
  - Outside the scope of this course