PETR 5313: CRN 38950, Fall 2017
Numerical Application in Petroleum Engineering, Lesson 10: ODE

Ekarit Panacharoensawad, PhD
Terry Fuller Petroleum Engineering Research Building Room 236
ekarit.panacharoensawad@ttu.edu

Cite as: Panacharoensawad, E. (2017) "Numerical application in petroleum engineering, Lesson 10: ODE", presentation slide for Texas Tech PETR 5313 Fall 2017

Outline

- Analytical solution via Sympy
- Numerical method
  - Explicit Euler
  - Implicit Euler
  - Runge-Kutta 4th Order
  - Butcher Tableau
  - Backward Differentiation Formula
  - Lobatto IIIC
  - Dormand-Prince 5(4)
- System of ODEs / Higher order ODE / BC problem
- Scipy automatic functions

Analytical Solution: Question 1

Solve $\dfrac{dy}{dx} = y$ for x = 0 to x = 10 at x = 10 y = 1

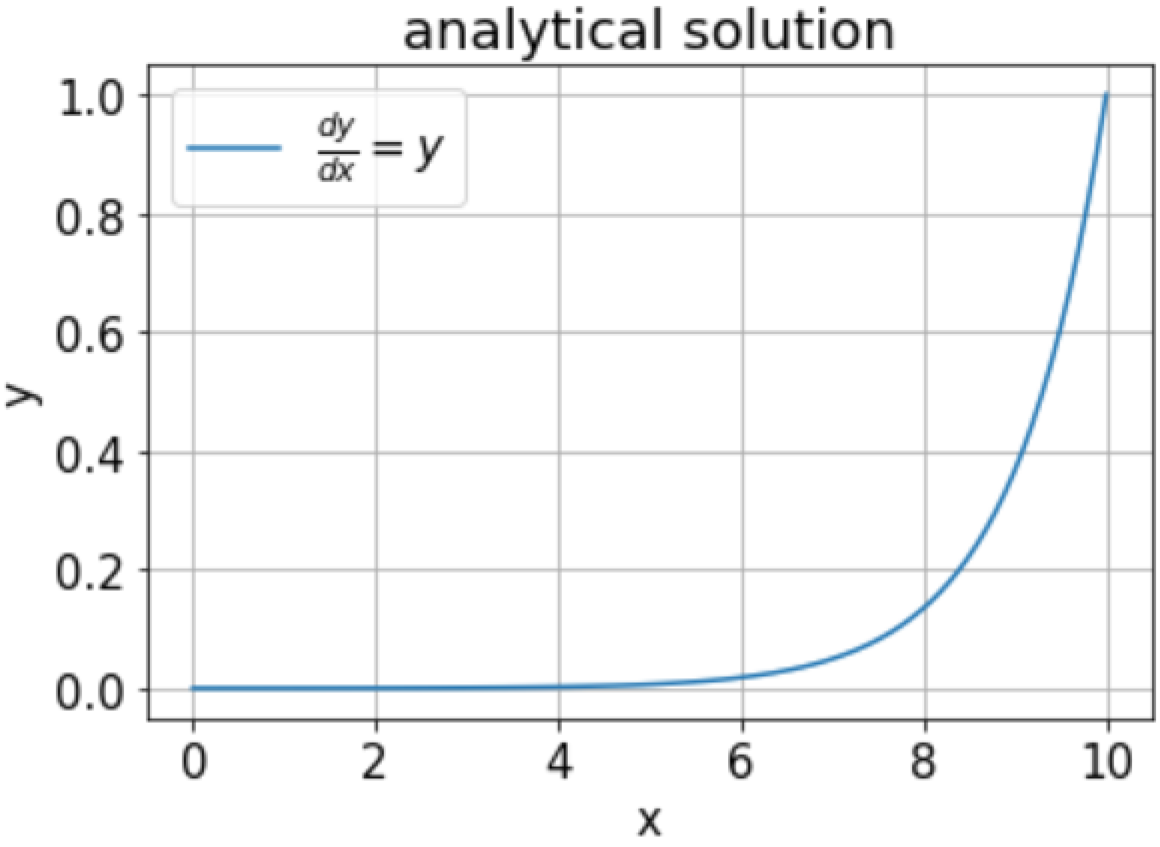
$$\int \frac{dy}{y} = \int dx$$

$$ln(y) = x + \text{constant}$$

$$y = Ce^x$$

$$1 = Ce^{10} \text{ or } C = e^{-10}$$

$$y = e^{-10}e^x$$



Sympy Analytical Solution: Solving ODE

```
import sympy as sm                          #import statement
xs, ys = sm.symbols('x y')                  #define symbol x to be var xs
fs = sm.Function('f')                       #define function f to be var fs
y_sm = sm.dsolve(sm.Derivative(fs(xs),xs)-fs(xs), fs(xs))
```

To solve for $f(x)$

$$\boxed{\frac{\partial}{\partial x} f(x)} - f(x) = 0 \quad \text{or} \quad \frac{d}{dx} y = y$$

Sympy Analytical Solution: Result

Result from `sm.dsolve`

`y_sm` is `Eq(f(x), C1*exp(x))`

`y_sm.args` is `(f(x), C1*exp(x))`

.args give tuple of symbols. They are l.h.s. and r.h.s. of the symbolic computation result

Sympy Analytical Solution: Solving for Constants

```
C1 = sm.symbols('C1')
C1_N = sm.solve(y_sm.args[1].subs(xs,10)-1,C1)
y_sol = y_sm.subs(C1,C1_N[0])
```

`.subs(xs, 10)` is for substituting x with 10

`sm.solve` is for symbolically solving equation(s)

`sm.solve` take

- 1) equation to be solve
- 2) symbol to be solved for

Sympy for Second Order Homogeneous Linear ODE

Question 2: Solve

$$\frac{d^2y}{dx^2} + \frac{dy}{dx} - 6y = 0$$

At x = 0, y = 4, y'' = 5. Solve y for x = 0 to 1

```
E2 = sm.Derivative(fs(xs),xs,xs)
        + sm.Derivative(fs(xs),xs) – 6*fs(xs)
y2_sm = sm.dsolve(E2,fs(xs))
y2_sm
```

Output: $$f(x) = \overbrace{C_1 e^{-3x} + C_2 e^{2x}}^{\texttt{y2\_sm.args[1]}}$$

Sympy Question2: Find constant values

At x = 0, y = 4, y' = 5

```
con1_q2 = sm.Eq(y2_sm.args[1].subs({xs:0}),4)
con2_q2 = sm.Eq(y2_sm.args[1].diff(xs)
                .subs({xs:0}),5)
C1C2_N = sm.solve([con1_q2,con2_q2],[C1,C2])
C1C2_N
```

Output: $\left\{C_1 : \frac{3}{5},\quad C_2 : \frac{17}{5}\right\}$

Sympy Boundary Value Problem: Question 3

Question 3: $$\frac{d^2y}{dx^2} + \frac{dy}{dx} - 6y = sin(x) + x$$

BC: At x = 0, y = 1 and at x = 1, y' = 1

```
E3 = (sm.Derivative(fs(xs),xs,xs)
          + sm.Derivative(fs(xs),xs)
          - 6*fs(xs) - sm.sin(xs) - xs)
y3_sm = sm.dsolve(E3,fs(xs)).rhs
```

Sympy Q3: Solve for constants

Passing list of equations and list of symbols to be solved for into sm.solve to get dictionary of answer (symbols : answer).

- The difference here is that we take derivative first (`sm.diff`), before substitute the right boundary condition

```
con1_q3 = sm.Eq(y3_sm.subs({xs:0}),1)
con2_q3 = sm.Eq(sm.diff(y3_sm,xs).subs({xs:1})
                ,1)
C1C2_q3 = sm.solve([con1_q3,con2_q3],[C1,C2])
y3_sol = y3_sm.subs(C1C2_q3)
```

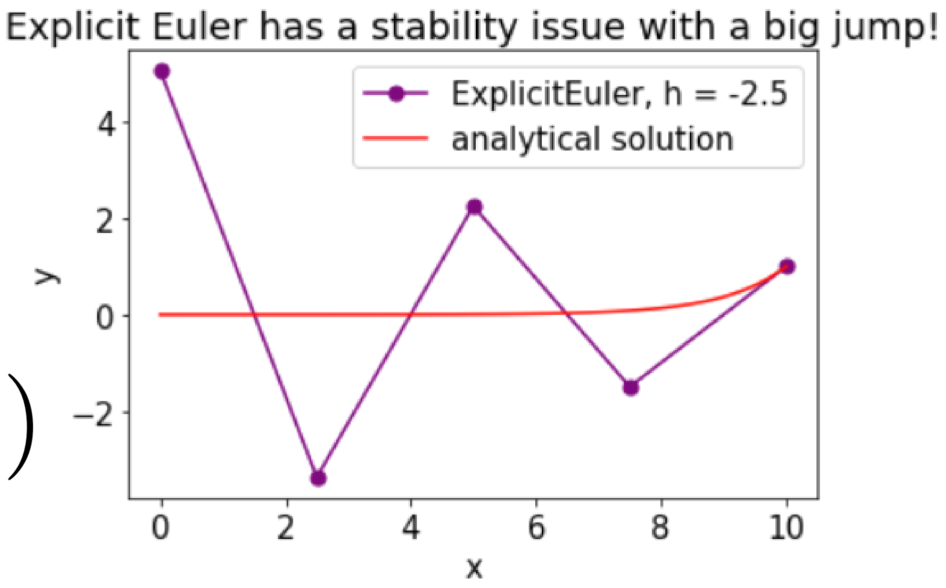
Explicit Euler Method

Use slope at the current point to estimate the next point.

- The next point is at the distance of h away from the current point.

$$\frac{dy}{dx} = f(x, y)$$

$$y_{n+1} = y_n + h \ \ f(x_n, y_n)$$



Implicit Euler Method

Use the slope at the next point to calculate the next point

$$\frac{dy}{dx} = f(x, y)$$

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$$

$$g(y_{n+1}) = -y_{n+1} + y_n + hf(x_{n+1}, y_{n+1})$$

If we find the right y_{n+1}, then $g(y_{n+1})$ become zero

Newton method can be used to solve this 1 eq 1 unknown

Stability of Implicit Euler

Use the test function of $\frac{dy}{dx} = ky$

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$$

$$y_{n+1} = y_n + hky_{n+1}$$

$$y_{n+1} - hky_{n+1} = y_n$$

$$(1 - hk)y_{n+1} = y_n$$

https://en.wikipedia.org/wiki/Stiff_equation#A-stability

Stability of Implicit Euler

$$y_{n+1} = \frac{1}{1 - hk} y_n$$

$$y_{n+1} = \phi(z) \cdot y_n \qquad \text{where} \quad \phi(z) = \frac{1}{1 - z} \quad \text{and} \quad z = hk$$

$$y_n = (\phi(z))^n \cdot y_0 \qquad \text{For any } z < 0, \text{ we have} \quad |\phi(z)| < 1$$

Thus, this method is A-stable (stability region is the entire left half-plane)

As $z \to \infty, \phi(z) \to 0$ Thus, this method is also L-stable

Explicit vs Implicit Euler

$\phi(z)$ Is called the stability function

For simplicity, stability of ODE numerical method can be categorized as A-stable and L-stable

$\text{For any } z < 0, \text{ we have } \quad |\phi(z)| < 1$

, then the method is A-stable

$\text{As } z \to \infty, \phi(z) \to 0$, then the method is L-stable

Implicit Euler is A- and L-stable (L-stable means very stable)

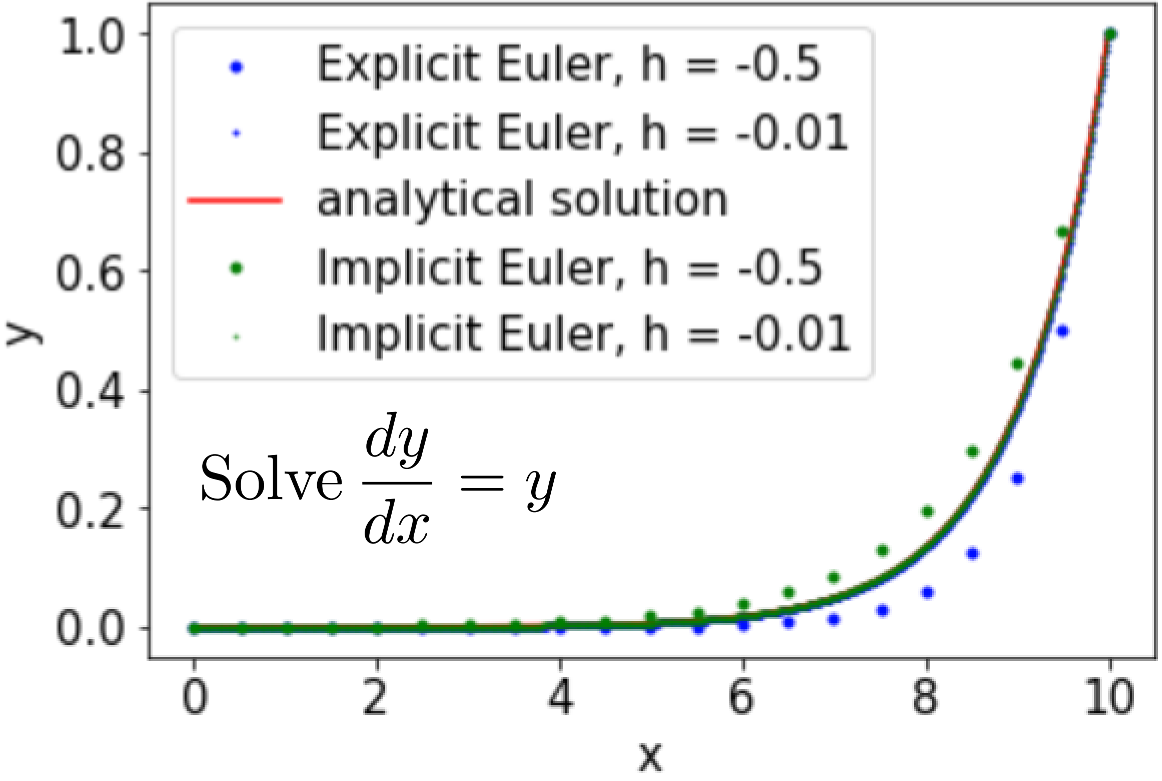
Explicit Euler is not A-stable (not quite stable)

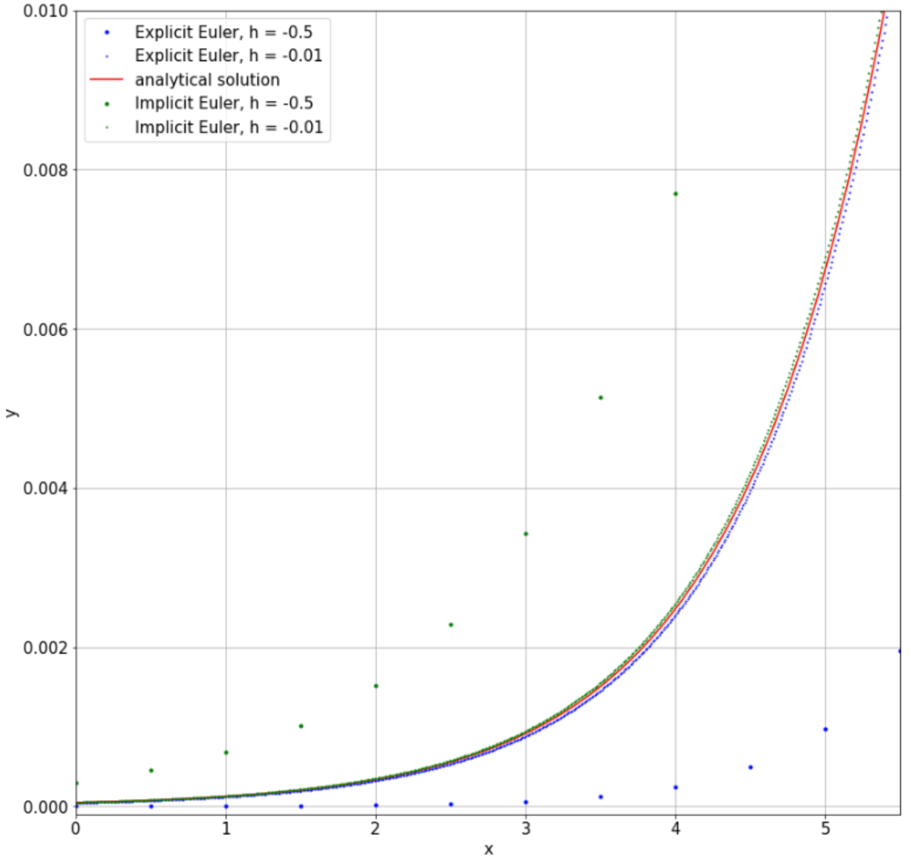
Not stable means as h increases, answer may diverge

Explicit vs Implicit Euler (1^{st} order accuracy)

Stable does not mean accurate!

Both are not accurate





Runge-Kutta 4th order (RK4 , Explicit Method)

The next point is from the average slope of ends and middle points. RK4 is not A-stable method. Quick and accurate!

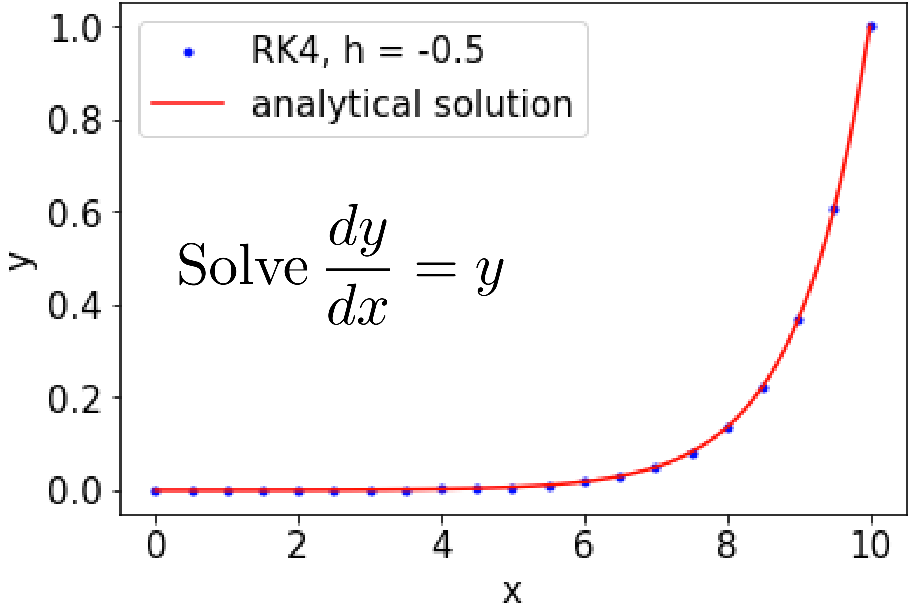
$$\frac{dy}{dx} = f(x, y) \qquad y_{n+1} = y_n + \frac{h}{6}(k_1 + 2\,k_2 + 2\,k_3 + k_4)$$

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

$$k_2 = f(x_n + 0.5\,h, y_n + 0.5\,h\,k_1)$$

$$k_3 = f(x_n + 0.5\,h, y_n + 0.5\,h\,k_2)$$

$$k_4 = f(x_n + h, y_n + hk_3)$$



Butcher Tableau

From $\dfrac{dy}{dt} = f(t, y)$

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

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

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

$$k_3 = f(t_n + c_3 h, y_n + h(a_{31} k_1 + a_{32} k_2)),$$

$$k_i = f\Big(t_n + c_i h, y_n + h \sum_{j=1}^{i-1} a_{ij} k_j\Big),$$

The corresponding Butcher Tableau is

$$\begin{array}{c|cccc}
c_1 & a_{11} & a_{12} & \dots & a_{1s} \\
c_2 & a_{21} & a_{22} & \dots & a_{2s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\
\hline
 & b_1 & b_2 & \dots & b_s
\end{array}$$

Butcher Tableau...

Explicit Euler

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$$

Implicit Euler

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

Runge-Kutta 4th Order

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

https://en.wikipedia.org/wiki/List_of_Runge%E2%80%93Kutta_methods#Forward_Euler

6-Step Backward Differentiation Formula (6-step BDF)

BDF is an implicit method

- Each step can be solved separately.
- For fully implicit BDF (need to solve everything simultaneously) see Akinfenwa et al 2013 (not cover)
- First step in BDF is the same as backward Euler (implicit Euler)
- Not A-stable, but still good for stiff ODE.

https://en.wikipedia.org/wiki/Backward_differentiation_formula

Akinfenwa, O.A., Jator, S.N. and Yao, N.M. (2013) Continuous block backward differntiation formula for solving stiff ordinary differential equations. Computer and Mathematics with Applications 65, pp. 996-1005

6-Step Backward Differentiation Formula

$$y_{n+1} - y_n = hf(t_{n+1}, y_{n+1})$$

$$y_{n+2} - \frac{4}{3}y_{n+1} + \frac{1}{3}y_n = \frac{2}{3}hf(t_{n+2}, y_{n+2})$$

$$y_{n+3} - \frac{18}{11}y_{n+2} + \frac{9}{11}y_{n+1} - \frac{2}{11}y_n = \frac{6}{11}hf(t_{n+3}, y_{n+3})$$

$$y_{n+4} - \frac{48}{25}y_{n+3} + \frac{36}{25}y_{n+2} - \frac{16}{25}y_{n+1} + \frac{3}{25}y_n = \frac{12}{25}hf(t_{n+4}, y_{n+4})$$

$$\begin{aligned} y_{n+5} - \frac{300}{137}y_{n+4} + \frac{300}{137}y_{n+3} - \frac{200}{137}y_{n+2} + \frac{75}{137}y_{n+1} - \frac{12}{137}y_n \\ = \frac{60}{137}hf(t_{n+5}, y_{n+5}) \end{aligned}$$

$$\begin{aligned} y_{n+6} - \frac{360}{147}y_{n+5} + \frac{450}{147}y_{n+4} - \frac{400}{147}y_{n+3} + \frac{225}{147}y_{n+2} - \frac{72}{147}y_{n+1} \\ + \frac{10}{147}y_n = \frac{60}{147}hf(t_{n+6}, y_{n+6}) \end{aligned}$$

6-Step BDF

Use Newton-Raphson to solve for the only unknown in each step

- 1st step, y_{n+1} is the only unknown.
- 2nd step, y_{n+2} is the only unknown.
- ...
- 6th step, y_{n+6} is the only unknown.

Get $O(h^6)$ order of accuracy ($h = x_{n+6} - x_n$).

- Accuracy can be less if Error from Newton-Raphson is high
  - To get accurate result, more iteration in Newton-Raphson is needed, but this makes it slower.

Lobatto IIIC

Fully Implicit: L-stable good for stiff ODEs

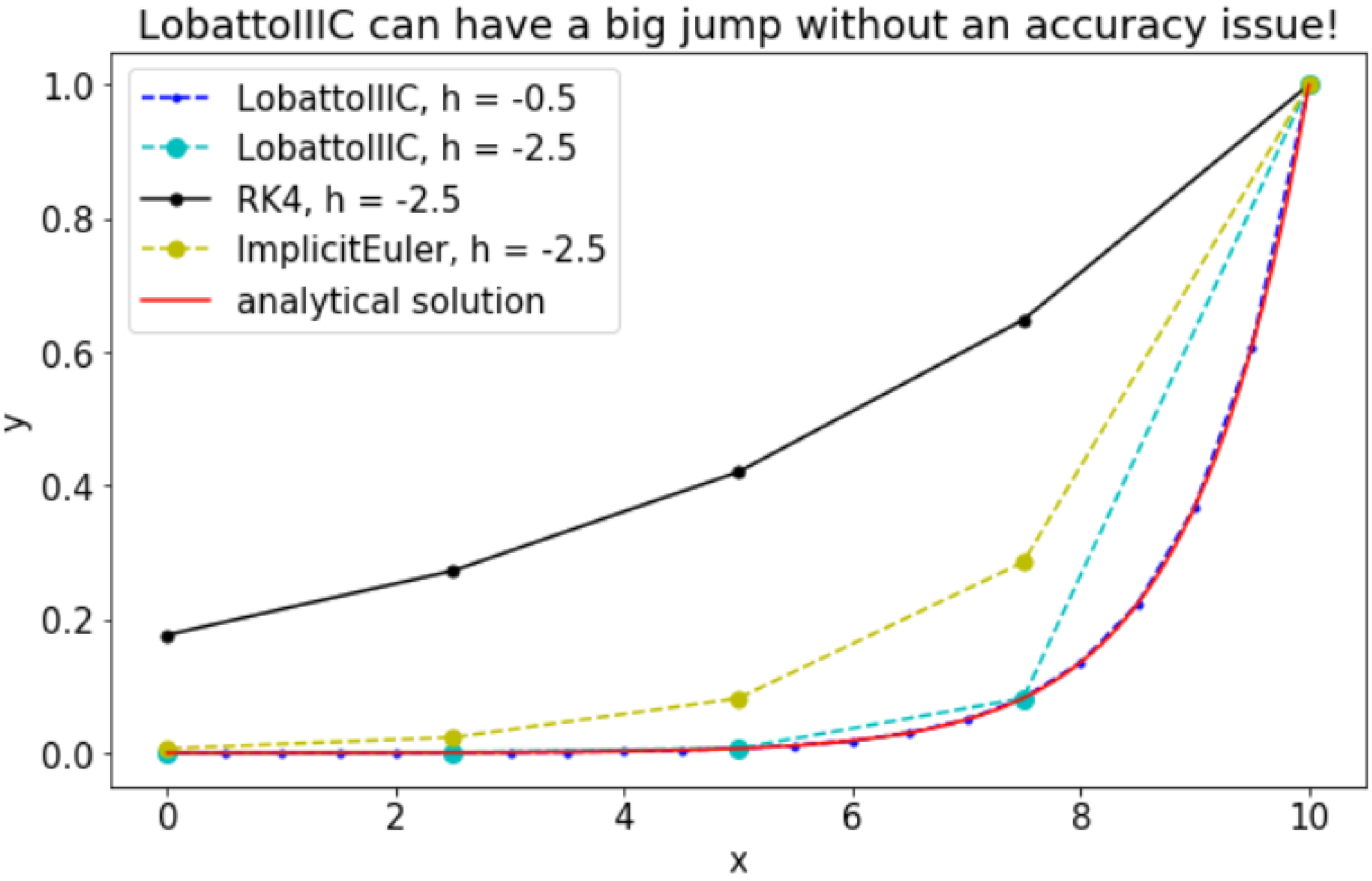
$$
\begin{array}{c|cccc}
0 & \frac{1}{12} & \frac{-\sqrt{5}}{12} & \frac{\sqrt{5}}{12} & \frac{-1}{12} \\
\frac{1}{2} - \frac{\sqrt{5}}{10} & \frac{1}{12} & \frac{1}{4} & \frac{10-7\sqrt{5}}{60} & \frac{\sqrt{5}}{60} \\
\frac{1}{2} + \frac{\sqrt{5}}{10} & \frac{1}{12} & \frac{10+7\sqrt{5}}{60} & \frac{1}{4} & \frac{-\sqrt{5}}{60} \\
1 & \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12} \\
\hline
 & \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12}
\end{array}
$$

http://homepage.math.uiowa.edu/~ljay/publications.dir/Lobatto.pdf

Lobatto IIIC Method (Fully Implicit)

Newton-Jacobian is needed to solve Lobatto IIIC equations

- Solve all 4 equations simultaneously
- Lobatto IIIC get less impact from a big step-size



Dormand-Prince 5(4): Explicit Method

Compare 5th and 4th answer to adjust the step size

- Use 5th order answer to get the next y, so that we have 5th order accuracy

$$
\begin{array}{c|ccccccc}
0 & & & & & & & \\
1/5 & 1/5 & & & & & & \\
3/10 & 3/40 & 9/40 & & & & & \\
4/5 & 44/45 & -56/15 & 32/9 & & & & \\
8/9 & 19372/6561 & -25360/2187 & 64448/6561 & -212/729 & & & \\
1 & 9017/3168 & -355/33 & 46732/5247 & 49/176 & -5103/18656 & & \\
1 & 35/384 & 0 & 500/1113 & 125/192 & -2187/6784 & 11/84 & \\
\hline
y_4 & 35/384 & 0 & 500/1113 & 125/192 & -2187/6784 & 11/84 & 0 \\
y_5 & 5179/57600 & 0 & 7571/16695 & 393/640 & -92097/339200 & 187/2100 & 1/40
\end{array}
$$

```
err = abs(y5 - y4)
s = (abs(eps*h/2/err))**0.2 #step ratio
```

Dormand-Prince 5(4): Explicit Method

Step ratio is used to decide if h is too big or not

$$err = |y_5 - y_4|$$

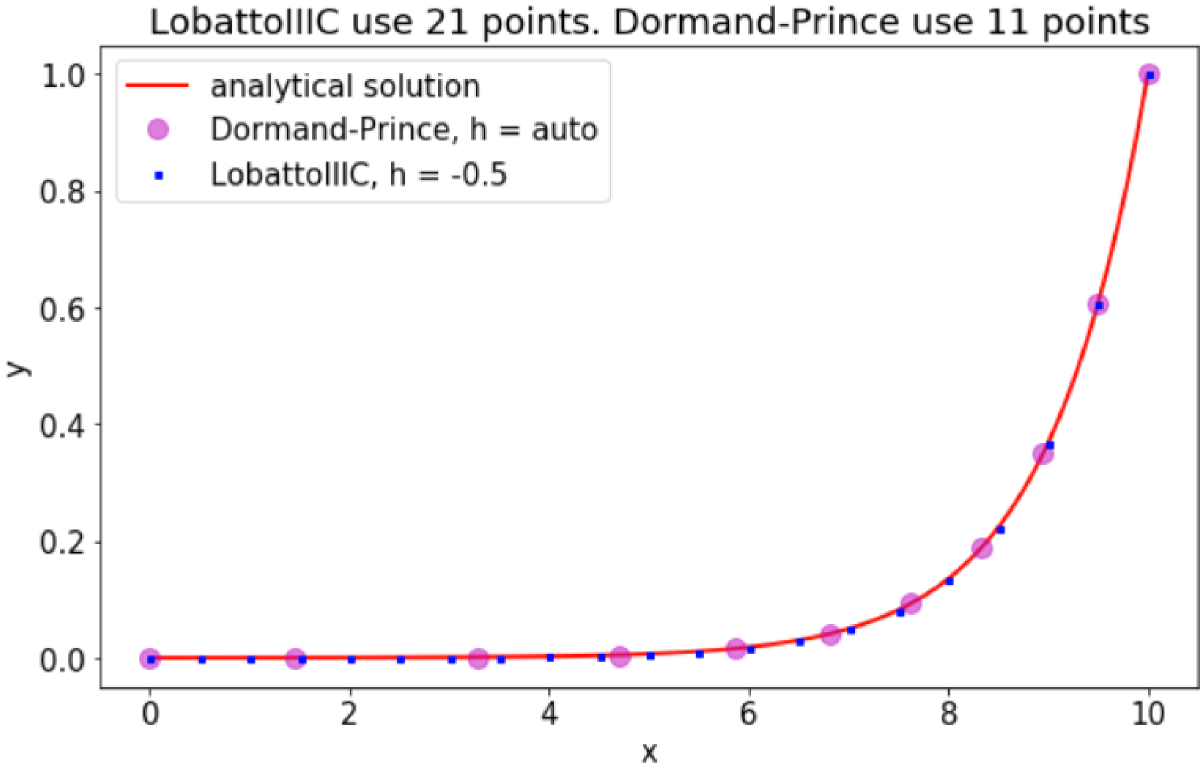
$$s = \left| \frac{\varepsilon * h}{2 * err} \right|^{1/5}$$

If s > a (a certain constant, e.g. 1), means that the selected h is small enough and we can move faster. Thus, the obtained result should be accepted.

If s < a (e.g. 1), the selected h is to large. h should be updated with h*s and the y5 need to be re-calculate with the new h.

Dormand-Prince 5(4) uses less point and less error

- DP5 is better than LBIIIC in this case.
- Faster and more accurate because there is no Newton-Jacobian involve.
- DP5 discretizes more where it is needed.




```
average error from Dormand-Prince 5(4) =  -2.97720218779e-05
average error from LobattoIIIC =  -7.43312233914e-05
Notice that even though the first step is the same
Dormand-Prince method use less points (more efficient)
```

System of First Order ODEs

Question 3: $$\frac{d^2y}{dx^2} + \frac{dy}{dx} - 6y = sin(x) + x$$

BC: At x = 0, y = 1 and at x = 1, y' = 1

Numerical method cannot directly solve higher order ODE

$$\frac{dy_0}{dx} = y_1$$

$$\frac{dy_1}{dx} = sin(x) + x + 6y_0 - y_1$$

Solving Boundary Value Problem: Shooting method

At x = 0, y = 1 At x = 1, y’ = 1

At x = 0, y’ = c

Define a function f as a black box solver.

- Take initial guess of y’ at x = 0
- Give y’ at x = 1, as an output

$$f(c) = y'_{\mathrm{at}\ x=1} \qquad g(c) = f(c) - 1$$

- where 1 is the y’ value at the right boundary
- Solve g(c) by Newton method or Bisection method

Creating ODE for shooting method with RK4

BC: At x = 0, y = 1 and at x = 1, y' = 1

$$\frac{dy_0}{dx} = y_1 \qquad \frac{dy_1}{dx} = sin(x) + x + 6y_0 - y_1$$

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

$$k1 = \begin{bmatrix} k1_0 \\ k1_1 \end{bmatrix} \qquad fn = \begin{bmatrix} fn_0(x,y) \\ fn_1(x,y) \end{bmatrix} \qquad yi = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$$

xi is a scalar quantity

$$k1 = fn(xi, yi)$$

$$k2 = fn(xi + 0.5 * h, yi + 0.5 * h * k1)$$

$$k3 = fn(xi + 0.5 * h, yi + 0.5 * h * k2)$$

$$k4 = fn(xi + 1.0 * h, yi + 1.0 * h * k3)$$

RK4 2nd Order ODE: Calculation Steps

$$\frac{dy_0}{dx} = y_1 \qquad \frac{dy_1}{dx} = sin(x) + x + 6y_0 - y_1$$

$$k0_1 = -1$$

$$k1_1 = sin(0) + 0 + 6 * 1 - (-1)$$

$$k0_2 = -1 + 0.5 * 0.1 * k1_1$$

$$k1_2 = sin(0.05) + 0.05 + 6(1 + 0.05k0_1) - (-1 + 0.05 * k1_1)$$

$$k0_3 = -1 + 0.5 * 0.1 * k1_2$$

$$k1_2 = sin(0.05) + 0.05 + 6(1 + 0.05k0_2) - (-1 + 0.05 * k1_2)$$

$$k0_4 = -1 + 0.1 * k1_3$$

$$k1_2 = sin(0.1) + 0.1 + 6(1 + 0.1k0_3) - (-1 + 0.1 * k1_3)$$

RK4 2nd Order ODE: Calculation Steps

- First, $k0_1$ and $k1_1$ must be calculated
  - (does not matter which one first)
- Then, $k0_2$ and $k1_2$ can be calculated
  - (does not matter which one first)
- Then, $k0_3$ and $k1_3$ can be calculated
  - (does not matter which one first)
- Then, $k0_4$ and $k1_4$ can be calculated
  - (does not matter which one first)

When calculate $k0_3$ and $k1_3$ **everything** must be specified at

- $xi + 0.5h$ and $yi + 0.5hk_2$

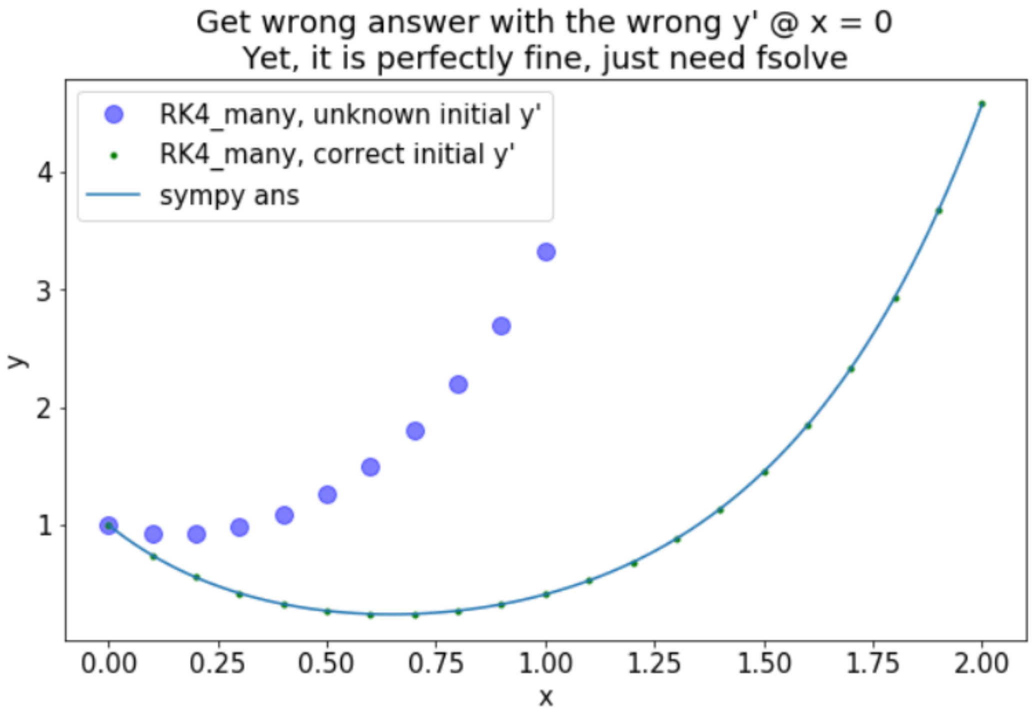
2^{nd} ODE BVP result from RK4

BC: At x = 0, y = 1 and at x = 1, y' = 1

$$\frac{d^2y}{dx^2} + \frac{dy}{dx} - 6y = sin(x) + x$$

Large blue-purple dots are from RK4 without Newton-Raphson

Green small dots are from from RK4 with Newton-Raphson to find the right initial y' at x = 0



Higher Order ODE with LobattoIIIC

$$\vec{y} = [y_0, y_1, \ldots]$$

$$\vec{y}_{n+1} = \vec{y}_n + \vec{f}(x_{n+1}, \vec{y}_{n+1})$$

$$\vec{f}(x, \vec{y}) = [\ f_0(x, \vec{y}),\ \ f_1(x, \vec{y})\ ]$$

$$\begin{array}{c|cccc}
0 & \frac{1}{12} & \frac{-\sqrt{5}}{12} & \frac{\sqrt{5}}{12} & \frac{-1}{12} \\
\frac{1}{2} - \frac{\sqrt{5}}{10} & \frac{1}{12} & \frac{1}{4} & \frac{10-7\sqrt{5}}{60} & \frac{\sqrt{5}}{60} \\
\frac{1}{2} + \frac{\sqrt{5}}{10} & \frac{1}{12} & \frac{10+7\sqrt{5}}{60} & \frac{1}{4} & \frac{-\sqrt{5}}{60} \\
1 & \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12} \\
\hline
 & \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12}
\end{array}$$

For 2nd Order, we have two sets of k1, k2, k3, and k4

- Need to solve 8 non-linear equation simultaneously

$$k_i = f\Big(t_n + c_i h, y_n + h \sum_{j=1}^{i-1} a_{ij} k_j\Big)$$

$$k_i = \begin{bmatrix} ki_0 \\ ki_1 \end{bmatrix} \quad f = \begin{bmatrix} f_0(x, y) \\ f_1(x, y) \end{bmatrix}$$

Higher Order ODE with LobattoIIIC

$$Eq1 = k1 - f\left(t_n + c_1 h, y_n + h\sum_{j=1}^{i-1} a_{ij} k1\right)$$

$$\vdots \quad = \vdots - f(\qquad \vdots \qquad)$$

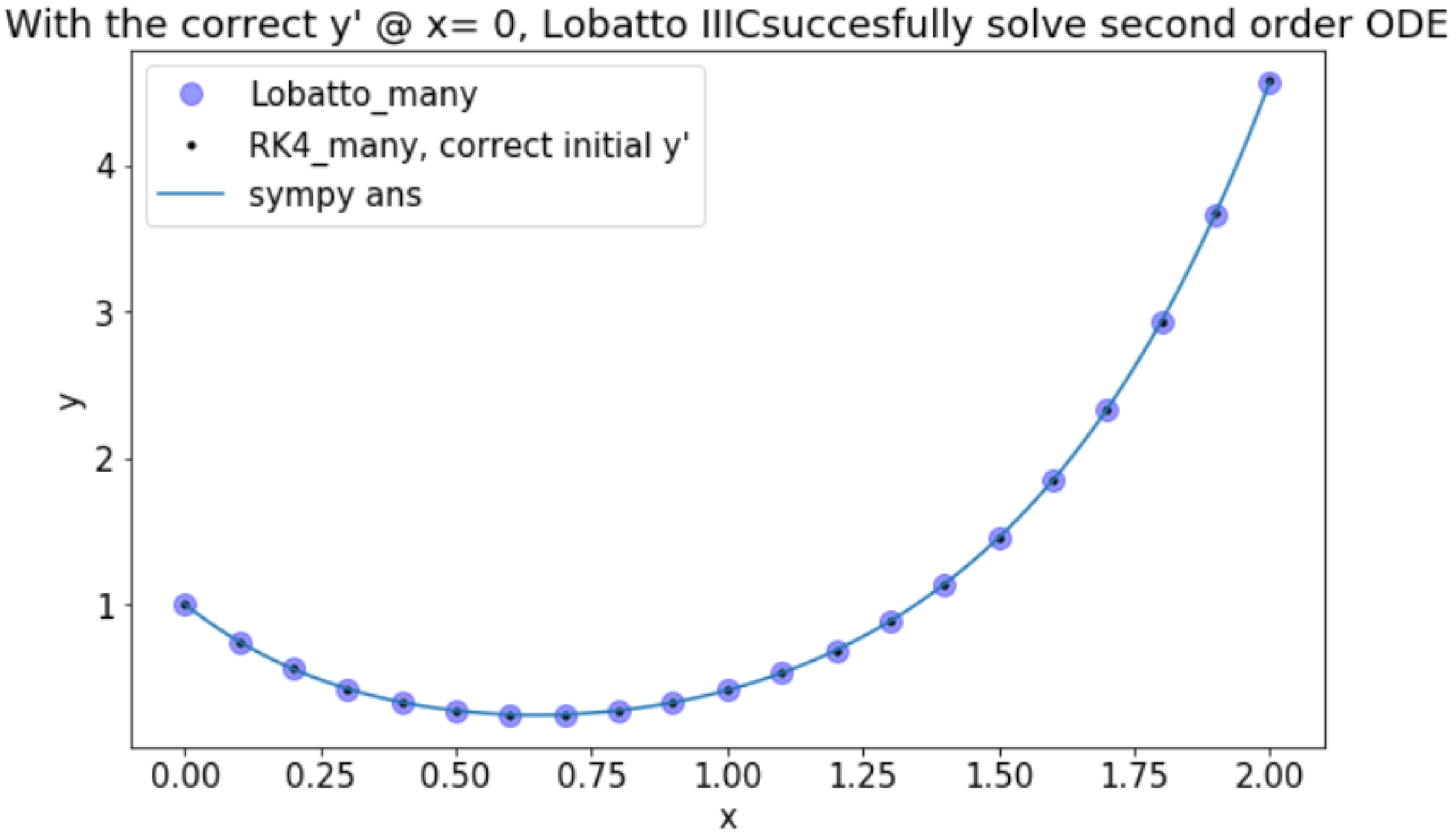
$$Eq4 = k4 - f\left(t_n + c_4 h, y_n + h\sum_{j=1}^{i-1} a_{ij} k4\right)$$

Each Eqi is a 2x1 column vector (we totally have 8 equations) with 8 unknown which are $k1_1$, $k1_2$, …, $k4_1$, $k4_2$

- Use numpy and scipy.optimize.fsolve to easily solve it

Higher Order ODE with LobattoIIIC

Lobatto IIIC is good for stiff ODEs



Dormand-Prince with higher order ODE

Calculate k1 to k6 as usual, but this time do it for f0 and f1

Get error for y[0] and y[1]

Adjust the step according to the term that provide a high error

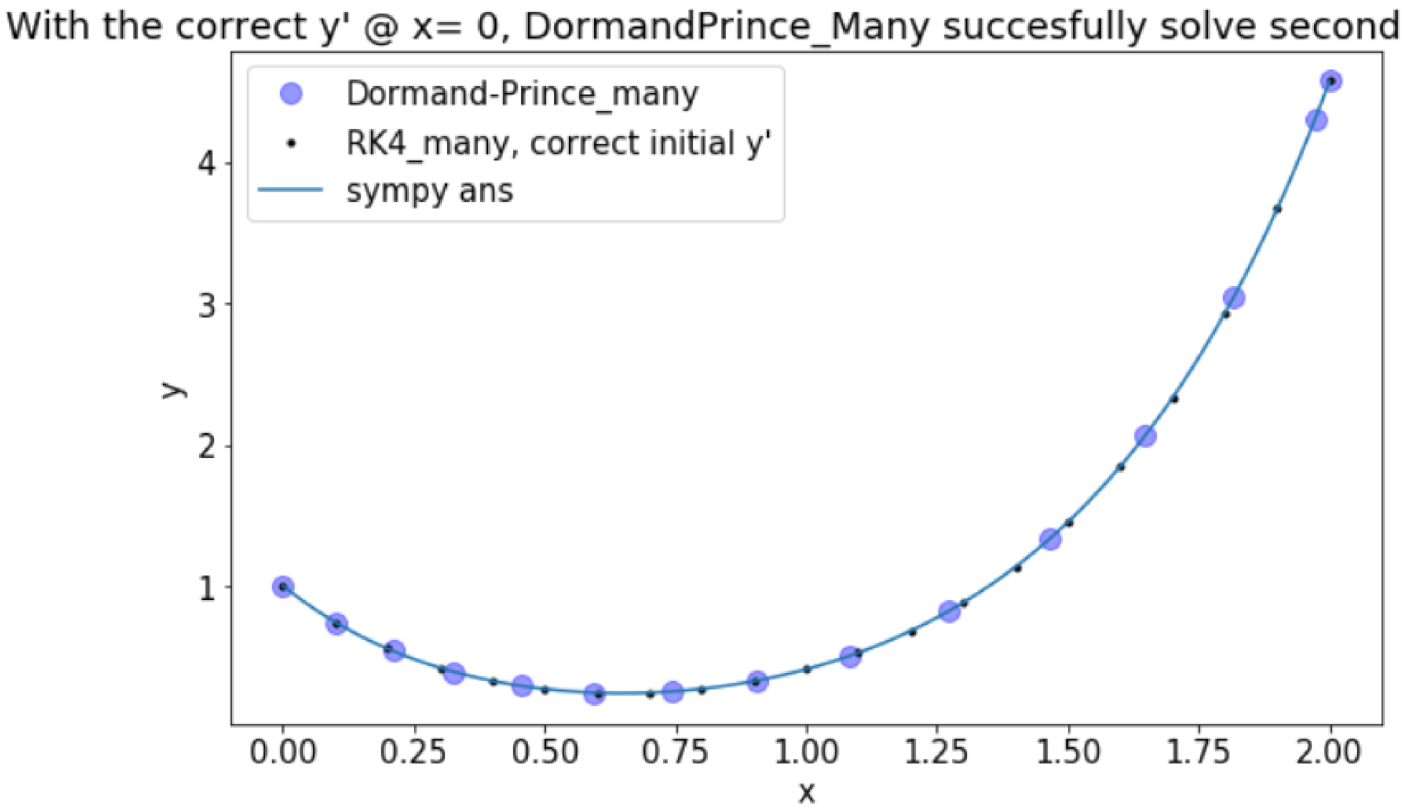
- `err = abs(y5 – y4).max()`

For second order ODE,

- k, y, y4, y5 become 2x1 column vector
- fn become a function taking x and vector y as an input. fn returns 2x1 column vector

Dormand-Prince with higher order ODE...

dopri5(4) is the efficient method. It uses less points and gives better result. For stiff method, decrease epsilon in the model



Scipy Function for solving ODEs: dopri5

Steps

- Create scipy.integrate.ode object
  - Specify the equation to be solved during the creation
- Set integration method for the created object
- Set initial condition for the new object
- Create user define function to retrieve intermediate data
- Set the function to be called during the run
  - If we don’t have this, we will just get the final result, not the non-uniform distribution of the data

Scipy Function: Solving ODE

```
fnp1 = lambda x, y: y
sol = []
def solout(t,y):
    sol.append([t,*y])
    return None
ode_solver = integrate.ode(fnp1)
ode_solver.set_integrator('dopri5', atol = 0.01)
ode_solver.set_initial_value(1,10)
ode_solver.set_solout(solout)
ode_solver.integrate(0)
data = np.array(sol)
```

Next slides for step-by-step explanation

Scipy Function: Solving ODE

Create function to be solved

```
fnp1 = lambda x, y: y
```

Create function to be called at each iteration to store intermediate answer

```
sol = []
def solout(t,y):
    sol.append([t,*y])
    return None
```

Scipy Function: Solving ODE

Creating object called `ode_solver`. Set the right-hand-side of ODE to be function `fnp1`

- `ode_solver = integrate.ode(fnp1)`

Set the method to be Dormand-Prince 5(4)

- `ode_solver.set_integrator('dopri5', atol = 0.01)`

Set the initial value of y to be 1 at x = 10

- `ode_solver.set_initial_value(1,10)`

Set the function to be called at each calculation step to be function name `solout`

- `ode_solver.set_solout(solout)`

Scipy Function: Solving ODE...

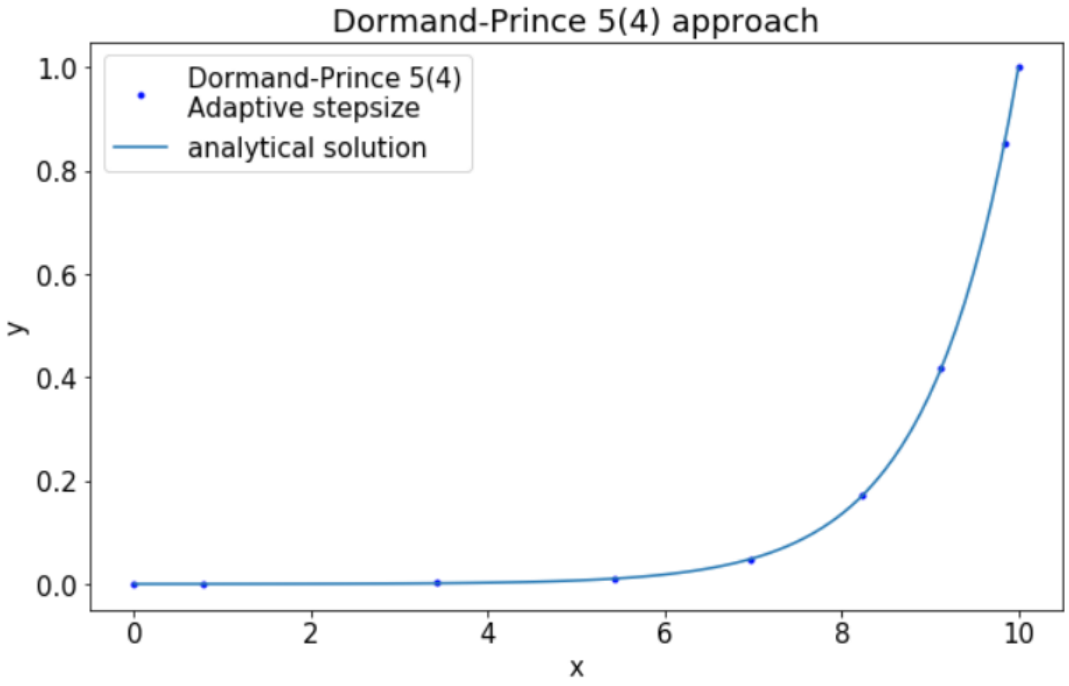
Perform Dormand-Prince method until x reach x = 0

- ➢ `ode_solver.integrate(0)`

Retrieve the data from list `sol` to numpy array ‘`data`’

- ➢ `data = np.array(sol)`

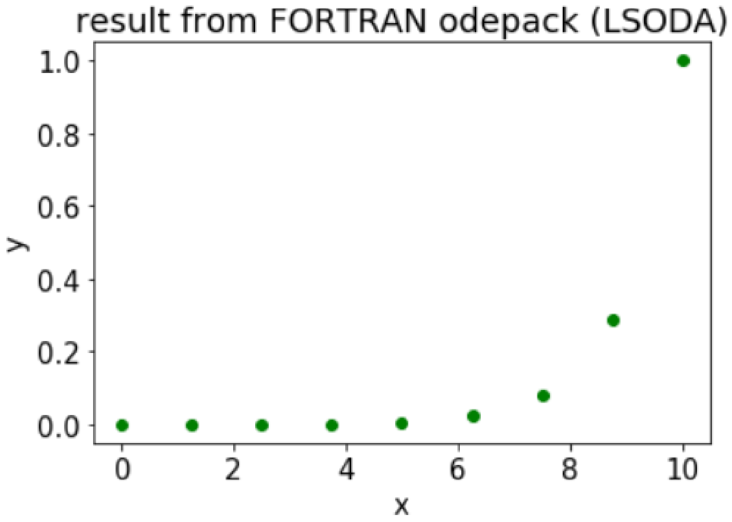
Result from scipy:



scipy.integrate.odeint

Odeint is quick to program, but we have less control. It will use LSODA library (automatic switching between implicit Adams and BDF). It is a constant step-size method.

```
f_odeint1 = lambda y, x: y
x_odeint1 = np.linspace(10,0,9)
y_result1 = integrate.odeint(f_odeint1, 1, x_odeint1)
plt.plot(x_odeint1, y_result1,'og')
```



Scipy Second Order ODE

Trick: Make function take y as a vector and return vector (same dimension as y)

```
sol2 = []
def solout2(t,y):
    sol2.append([t,*y])
    return None
f12q2 = lambda x,y: [y[1], 6 * y[0] - y[1]]
q2_ode = integrate.ode(f12q2)
q2_ode.set_integrator('dopri5')
q2_ode.set_initial_value([4,5],0)
q2_ode.set_solout(solout2)
q2_ode.integrate(5)
```

Scipy Second Order ODE...

Create function to retrieve intermediate calculation values

```
def solout2(t,y):
    sol2.append([t,*y])
```

Create function to be solved (y = vector, output = vector)

```
f12g2 = lambda x,y: [y[1], 6 * y[0] - y[1]]
```

$$\frac{dy_0}{dx} = y_1 \qquad \frac{dy_1}{x} = 6y_0 - y_1 \qquad \frac{d^2y}{dx^2} + \frac{dy}{dx} - 6y = 0$$

Set initial condition: y[0] = 4 and y[1] = 5 at x = 0

```
q2_ode.set_initial_value([4,5],0)
```

Scipy ODE: Boundary Value Problem

$$\frac{d^2y}{dx^2} + \frac{dy}{dx} - 6y = sin(x) + x$$

BC: @ x = 0, y = 1; @ x = 1, y' = 1

$$\frac{dy_0}{dx} = y_1$$

$$\frac{dy_1}{dx} = sin(x) + x + 6y_0 - y_1$$

Steps: 1) Create BC function, 2) Create dy/dx r.h.s. function, 3) create initial guess, 4) use `integrate.solve_bvp`

Scipy ODE: Boundary Value Problem...

Specify boundary conditions

```
def bc_3(ya, yb):
    return np.array([ya[0] - 1, yb[1] - 1])
```

Specify ODEs

```
def fun_3(x,y):
    return np.vstack((y[1], np.sin(x) + x + 6 *
                                  y[0]-y[1]))
```

Create domain x and initial guess y, then solve!

```
x_3 = np.linspace(0,1,100)
y_3 = np.ones((2,x_3.size))
sol_3 = integrate.solve_bvp(fun_3,bc_3,x_3,y_3)
```

Scipy ODE: Boundary Value Problem...

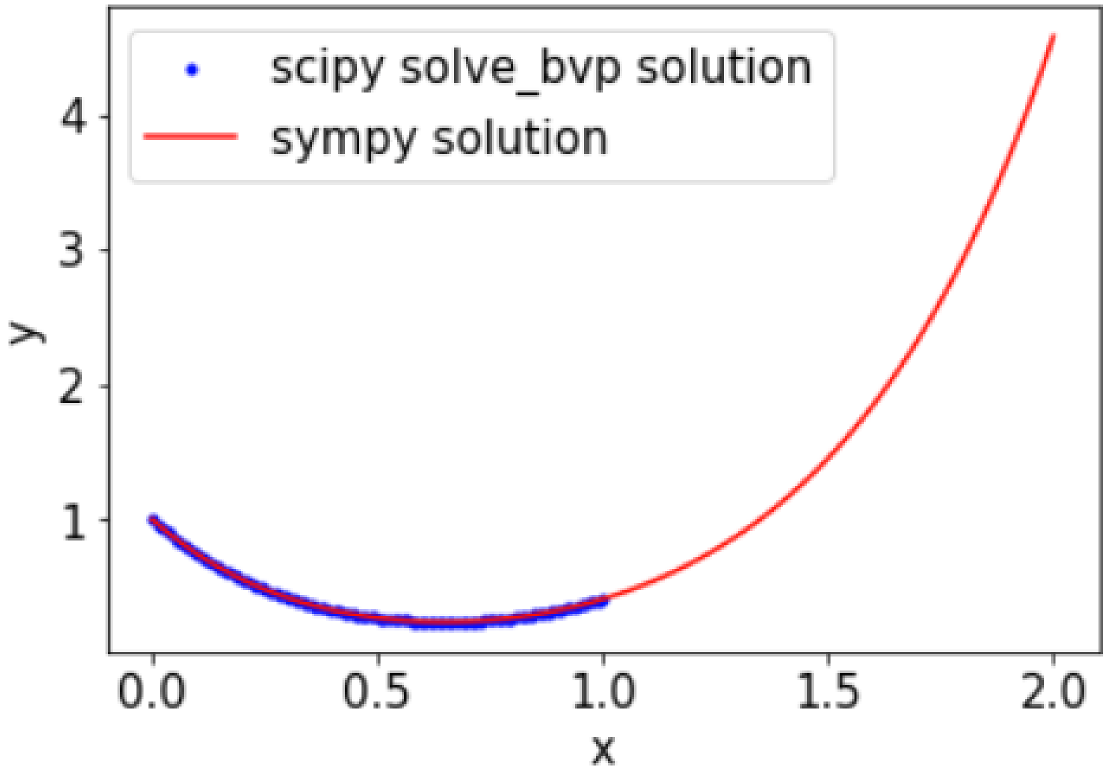
sol_3 output is an object contain 1) x, 2) y, 3) y', and 4) residual

#y = array[y[0], y[1]]

#y[1] is dy0/dx

#yp = array[[yp[0], yp[1]]

#yp[0] is dy0/dx




```
plt.plot(x_3,sol_3.y[0])
```

Scipy ODE: Boundary Value Problem...

Once Boundary Value Problem is solved, we obtain the right initial condition (y' at x = 0) and both y' and y and x = 1.

Next step is to use y and y' at x = 1 as IC to solve for y from 1 to 2 (use integrate.ode as usual)

```
ode3 = integrate.ode(fun_3)
ode3.set_integrator('dopri5')
ode3.set_initial_value([sol_3.y[0][-1],
                        sol_3.y[1][-1]], 1)
ode3.set_solout(solout3)
```