Tutorial-RK_Butcher_Table_Validation

May 12, 2023

-1 Validating Runge Kutta Butcher tables using Truncated Taylor Series

-1.1 Authors: Zach Etienne, Brandon Clark, & Gabriel M Steward

-1.2 This tutorial notebook is designed to validate the Butcher tables contained within the Butcher dictionary constructed in the RK Butcher Table Dictionary NRPy+ module.

-1.2.1 NRPy+ Source Code for this module:

- MoLtimestepping/RK_Butcher_Table_Validation.py stores the `Validate` function for validating convergence orders for Runge Kutta methods.
- MoLtimestepping/RK_Butcher_Table_Dictionary.py **[tutorial]** accesses the Butcher table dictionary `Butcher_dict` for known explicit Runge Kutta methods.

-1.3 Introduction:

Starting with the ODE (ordinary differential equation) initial value problem:

$$y'(t) = f(y,t) \quad y\left(t=0\right) = y_0,$$

for various choices of $f(y,t)$, this module validates the Runge Kutta (RK) methods coded in RK_Butcher_Table_Dictionary.py **tutorial notebook** as follows.

Given y_0 and a smooth $f(y,t)$, all explicit RK methods provide an estimate for $y_1 = y\left(\Delta t\right)$, with an error term that is proportional to $\left(\Delta t\right)^m$, where m is an integer typically greater than zero. This error term corresponds to the *local* truncation error. For RK4, for example, while the *total accumulated truncation error* (i.e., the accumulated error at a fixed final time t_f) is proportional to $\left(\Delta t\right)^4$, the *local* truncation error (i.e., the error after one arbitrarily chosen timestep Δt) is proportional to $\left(\Delta t\right)^5$.

If the exact solution $y(t)$ is known as a closed-form expression, then $y\left(\Delta t\right)$ can be *separately* written as a Taylor expansion about $y(t=0)$:

$$y\left(\Delta t\right) = \sum_{n=0}^{\infty} \frac{y^{(n)}(t=0)}{n!} \left(\Delta t\right)^n,$$

where $y^{(n)}(t=0)$ is the nth derivative of $y(t)$ evaluated at $t=0$.

The above expression will be known exactly. Furthermore, if one chooses a numerical value for y_0 *and leaves* Δt *unspecified*, any explicit RK method will provide an estimate for $y\left(\Delta t\right)$ of the form

$$y\left(\Delta t\right) = \sum_{n=0}^{\infty} a_n \left(\Delta t\right)^n,$$

where a_n *must* match the Taylor expansion of the *exact* solution at least up to and including terms proportional to $(\Delta t)^m$, where m is the order of the local truncation error. If this is *not* the case, then the Butcher table was almost certainly *not* typed correctly.

Therefore, comparing the numerical result with unspecified Δt against the exact Taylor series provides a convenient (though not perfectly robust) means to verify that the Butcher table for a given RK method was typed correctly. Multiple typos in the Butcher tables were found using this approach.

Example from Z. Etienne's MATH 521 (Numerical Analysis) lecture notes:

Consider the ODE

$$y' = y - 2te^{-2t}, \quad y(0) = y(t_0) = 0.$$

- Solve this ODE exactly, then Taylor expand the solution about $t = 0$ to approximate the solution at $y(t = \Delta t)$ to fifth order in Δt.
- Next, solve this ODE using Heun's method (second order in total accumulated truncation error, third order in local truncation error) *{by hand} with a step size of* Δt *to find* $y(\Delta t)$. *Confirm that the solution obtained when using Heun's method has an error term that is at worst* $\mathcal{O}\left((\Delta t)^3\right)$. *If the dominant error is proportional to a higher power of* Δt, *explain the discrepancy.*
- *Finally, solve this ODE using the Ralston method {by hand} with a step size of* Δt *to find* $y(\Delta t)$. *Is the coefficient on the dominant error term closer to the exact solution than Heun's method?*

We can solve this equation via the method of integrating factors, which states that ODEs of the form:

$$y'(t) + p(t)y(t) = g(t)$$

are solved via

$$y(t) = \frac{1}{\mu(t)}\left[\int \mu(s)g(s)ds + c\right],$$

where the integrating factor $\mu(t)$ is given by

$$\mu(t) = \exp\left(\int p(t)dt\right).$$

Here, $p(t) = -1$ and $g(t) = -2te^{-2t}$. Then

$$\mu(t) = \exp\left(-\int dt\right) = e^{-t+c} = ke^{-t} \tag{1}$$

and

$$y(t) = e^t/k\left[\int ke^{-s}(-2se^{-2s})ds + c\right] = -2e^t\left[\int se^{-3s}ds + c'\right] \tag{2}$$

$$= -2e^t\left[e^{-3t}\left(-\frac{t}{3} - \frac{1}{9}\right) + c'\right] = -2e^{-2t}\left(-\frac{t}{3} - \frac{1}{9}\right) - 2c'e^t \tag{3}$$

$$= e^{-2t}\left(2\frac{t}{3} + \frac{2}{9}\right) + c''e^t. \tag{4}$$

$$\tag{5}$$

If $y(0) = 0$ then we can compute the integration constant c'', and $y(t)$ becomes

$$y(t) = \frac{2}{9} e^{-2t} \left(3t + 1 - e^{3t}\right).$$

The Taylor Series expansion of the exact solution about $t = 0$ evaluated at $y(\Delta t)$ yields

$$y(\Delta t) = -(\Delta t)^2 + (\Delta t)^3 - \frac{3(\Delta t)^4}{4} + \frac{23(\Delta t)^5}{60} - \frac{19(\Delta t)^6}{120} + O\left((\Delta t)^7\right).$$

Next we evaluate $y(\Delta t)$ using Heun's method. We know $y(0) = y_0 = 0$ and $f(y,t) = y - 2te^{-2t}$, so

$$\begin{aligned}
k_1 &= \Delta t f(y(0), 0) && (6)\\
&= \Delta t \times 0 && (7)\\
&= 0 && (8)\\
k_2 &= \Delta t f(y(0) + k_1, 0 + \Delta t) && (9)\\
&= \Delta t f(y(0) + 0, 0 + \Delta t) && (10)\\
&= \Delta t(-2\Delta t e^{-2\Delta t}) && (11)\\
&= -2(\Delta t)^2 e^{-2\Delta t} && (12)\\
y(\Delta t) &= y_0 + \frac{1}{2}(k_1 + k_2) + \mathcal{O}\left((\Delta t)^3\right) && (13)\\
&= 0 - (\Delta t)^2 e^{-2\Delta t} && (14)\\
&= -(\Delta t)^2(1 - 2\Delta t + 2(\Delta t)^2 + ...) && (15)\\
&= -(\Delta t)^2 + 2(\Delta t)^3 + \mathcal{O}\left((\Delta t)^4\right). && (16)
\end{aligned}$$

Thus the coefficient on the $(\Delta t)^3$ term is wrong, but this is completely consistent with the fact that our stepping scheme is only third-order accurate in Δt.

In the below approach, the RK result is subtracted from the exact Taylor series result, as a check to determine whether the RK Butcher table was coded correctly; if it was not, then the odds are good that the RK results will not match to the expected local truncation error order. Multiple $f(y,t)$ are coded below to improve the robustness of this test.

As NRPy+'s butcher tables also contain the information to run Adams-Bashforth (AB) methods, those too shall also be validated. Fortunately the exact same validation method that we use for the RK methods also functions for the AB methods, all that needs to change is the specific implementation code.

0 Table of Contents

This notebook is organized as follows

1 Step 1: Initialize needed Python/NRPy+ modules [Back to top]

Let's start by importing all the needed modules from Python/NRPy+:

[1]:
```
import sympy as sp              # SymPy: The Python computer algebra package upon which NRPy+ depends
import numpy as np              # NumPy: A numerical methods module for Python
from MoLtimestepping.RK_Butcher_Table_Dictionary import Butcher_dict
```

2 Step 2: Validate Convergence Order of Butcher Tables [Back to top]

Each Butcher table/Runge Kutta method is tested by solving an ODE. Comparing the Taylor series expansions of the exact solution and the numerical solution as discussed in the **Introduction** above will confirm whether the method converges to the appropriate order.

2.1 Step 2.a: Defining the right-hand side of the ODE [Back to top]

Consider the form of ODE $y' = f(y,t)$. The following begins to construct a dictionary `rhs_dict` of right-hand side functions for us to validate explicit Runge Kutta methods. The most up-to-date catalog of functions stored in `rhs_dict` can be found in the RK_Butcher_Table_Validation.py module.

[2]:
```
def fypt(y,t): # Yields expected convergence order for all cases
    #            except DP6 which converge to higher order (7, respectively)
    return y+t

def fy(y,t): # Yields expected convergence order for all cases
    return y

def feypt(y,t): # Yields expected convergence order for all cases
    return sp.exp(1.0*(y+t))

def ftpoly6(y,t): # Yields expected convergence order for all cases, L6 has 0 error
    return 2*t**6-389*t**5+15*t**4-22*t**3+81*t**2-t+42
rhs_dict = {'ypt':fypt, 'y':fy, 'eypt':feypt, 'tpoly6':ftpoly6}
```

2.2 Step 2.b: Defining a Validation Function [Back to top]

To validate each Butcher table we compare the exact solutions to ODEs with the numerical solutions using the Runge Kutta scheme built into each Butcher table. The following is a function that

1. solves the ODE exactly,
2. solves the ODE numerically for a given Butcher table, and
3. compares the two solutions and checks for the order of convergence by returning their difference.

The `Validate()` function inputs a specified `Butcher_key`, the starting guess solution and time `y_n`, `t_n` and the right-hand side of the ODE corresponding to a specified initial value problem, `rhs_key`.

[3]:
```
from MoLtimestepping.RK_Butcher_Table_Dictionary import Butcher_dict
def Validate(Butcher_key, yn, tn, rhs_key):
    # 1. First we solve the ODE exactly
    y = sp.Function('y')
    sol = sp.dsolve(sp.Eq(y(t).diff(t), rhs_dict[rhs_key](y(t), t)), y(t)).rhs
    constants = sp.solve([sol.subs(t,tn)-yn])
    exact = sol.subs(constants)

    # 2. Now we solve the ODE numerically using specified Butcher table

    # Access the requested Butcher table
    Butcher = Butcher_dict[Butcher_key][0]
    # Determine number of predictor-corrector steps
    L = len(Butcher)-1
    # Set a temporary array for update values
    k = np.zeros(L, dtype=object)
    # Initialize intermediate variable
    yhat = 0
    # Initialize the updated solution
    ynp1 = 0
    for i in range(L):
        #Initialize and approximate update for solution
        yhat = yn
        for j in range(i):
            # Update yhat for solution using a_ij Butcher table coefficients
            yhat += Butcher[i][j+1]*k[j]
            if Butcher_key == "DP8" or Butcher_key == "L6":
                yhat = 1.0*sp.N(yhat,20) # Otherwise the adding of fractions kills performance.
        # Determine the next corrector variable k_i using c_i Butcher table coefficients
        k[i] = dt*rhs_dict[rhs_key](yhat, tn + Butcher[i][0]*dt)
        # Update the solution at the next iteration ynp1 using Butcher table coefficients
        ynp1 += Butcher[L][i+1]*k[i]
```

```
    # Finish determining the solution for the next iteration
    ynp1 += yn

    # Determine the order of the RK method
    order = Butcher_dict[Butcher_key][1]+2
    # Produces Taylor series of exact solution at t=tn about t = 0 with the specified order
    exact_series = sp.series(exact.subs(t, dt),dt, 0, order)
    num_series = sp.series(ynp1, dt, 0, order)
    diff = exact_series-num_series
    return diff
```

2.3 Step 2.c: Validating RK Methods against ODEs [Back to top]

The following makes use of the `Validate()` function above to demonstrate that each method within the Butcher table dictionary converges to the expected order for the given right-hand side expression.

[4]:
```
t, dt = sp.symbols('t dt')
# Set initial conditions
t0 = 0
y0 = 1
# Set RHS of ODE
function = 'ypt'# This can be changed, just be careful that the initial conditions are satisfied
for key,value in Butcher_dict.items():
    if key not in {"AHE", "ABS", "ARKF", "ACK", "ADP5", "ADP8", "AB"}:
        print("RK method: \""+str(key)+"\".")
        y = sp.Function('y')
        print(" When solving y'(t) = "+str(rhs_dict[function](y(t),t))+", y("+str(t0)+")="+str(y0)+",")
        local_truncation_order = list(value)[1]+1
        print(" the first nonzero term should have local truncation error proportional to O(dt^"+str(local_truncation_order)+") or a␣
 ↪higher power of dt.")
        print("Subtracting the numerical result from the exact Taylor expansion, we find a local truncation error of:")
        sp.pretty_print(Validate(key, y0, t0, function))
    #     print("\n")
        print(" (Coefficients of order 1e-15 or less may generally be ignored, as these are at roundoff error.)\n")
        if key == "DP8":
            break # Keep this code from trying to validate Adaptive and Adams-Bashforth methods
            # They need to be read differently.
```

```
RK method: "Euler".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^2) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
```

```
local truncation error of:
  2    [U+239B]  3[U+239E]
dt  + O[U+239D]dt [U+23A0]
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "RK2 Heun".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^3) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
  3
dt     [U+239B]  4[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 3
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "RK2 MP".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^3) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
  3
dt     [U+239B]  4[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 3
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "RK2 Ralston".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^3) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
  3
dt     [U+239B]  4[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 3
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)
```

```
RK method: "RK3".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^4) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
  4
dt     [U+239B]  5[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 12
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "RK3 Heun".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^4) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
  4
dt     [U+239B]  5[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 12
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "RK3 Ralston".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^4) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
  4
dt     [U+239B]  5[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 12
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "SSPRK3".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^4) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
```

```
  4
dt     [U+239B]  5[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 12
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "RK4".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^5) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
  5
dt     [U+239B]  6[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 60
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "DP5".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^6) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
    6
  dt      [U+239B]  7[U+239E]
- [U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
  1800
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "DP5alt".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^6) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
     6
13·dt     [U+239B]  7[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
231000
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)
```

```
RK method: "CK5".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^6) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
  6
dt      [U+239B]  7[U+239E]
[U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
3600
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "DP6".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^7) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
 [U+239B]  8[U+239E]
O[U+239D]dt [U+23A0]
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "L6".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^7) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
                              5                                   6
- 1.0587911840678754238e-22·dt  - 2.6469779601696885596e-23·dt  + 0.0013227513

               7    [U+239B]  8[U+239E]
227513227513·dt  + O[U+239D]dt [U+23A0]
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "DP8".
 When solving y'(t) = t + y(t), y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^9) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
```

```
                                                                  2
3.6854403535034607753e-18·dt + 5.8394451383711465375e-18·dt  + 3.7764963953332

             3                                4                                   5
980617e-18·dt  + 9.542884942003761195e-19·dt  + 1.2718729098615353529e-19·dt

                               6                                  7
+ 3.9082629581905451582e-20·dt  + 4.8075737201581968464e-21·dt  + 5.1688448526

                 8                                9    [U+239B]  10[U+239E]
907316834e-22·dt  + 7.2078645877627939543e-9·dt  + 0[U+239D]dt  [U+23A0]
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)
```

2.4 Step 2.d: Validating Inherently Adaptive RK Methods against ODEs [Back to top]

The following code validates inherently adaptive RK methods. Each method has two validation checks as each method has, individually, two different methods inside of it. One should have the first error term's order be equal to that of the method's order, and the other check should have the firs terror term's order be the method's order plus one.

[5]:
```python
def ValidateARK1(Butcher_key, yn, tn, rhs_key):
    # 1. First we solve the ODE exactly
    y = sp.Function('y')
    sol = sp.dsolve(sp.Eq(y(t).diff(t), rhs_dict[rhs_key](y(t), t)), y(t)).rhs
    constants = sp.solve([sol.subs(t,tn)-yn])
    exact = sol.subs(constants)

    # 2. Now we solve the ODE numerically using specified Butcher table

    # Access the requested Butcher table
    Butcher = Butcher_dict[Butcher_key][0]
    # Determine number of predictor-corrector steps
    L = len(Butcher)-1
    # Set a temporary array for update values
    k = np.zeros(L, dtype=object)
    # Initialize intermediate variable
    yhat = 0
    # Initialize the updated solution
    ynp1 = 0
    for i in range(L-1):
        #Initialize and approximate update for solution
        yhat = yn
```

```
        for j in range(i):
            # Update yhat for solution using a_ij Butcher table coefficients
            yhat += Butcher[i][j+1]*k[j]
            if Butcher_key == "DP8" or Butcher_key == "L6":
                yhat = 1.0*sp.N(yhat,20) # Otherwise the adding of fractions kills performance.
        # Determine the next corrector variable k_i using c_i Butcher table coefficients
        k[i] = dt*rhs_dict[rhs_key](yhat, tn + Butcher[i][0]*dt)
        # Update the solution at the next iteration ynp1 using Butcher table coefficients
        ynp1 += Butcher[L][i+1]*k[i]
    # Finish determining the solution for the next iteration
    ynp1 += yn

    # Determine the order of the RK method
    order = Butcher_dict[Butcher_key][1]+2
    # Produces Taylor series of exact solution at t=tn about t = 0 with the specified order
    exact_series = sp.series(exact.subs(t, dt),dt, 0, order)
    num_series = sp.series(ynp1, dt, 0, order)
    diff = exact_series-num_series
    return diff

def ValidateARK2(Butcher_key, yn, tn, rhs_key):
    # 1. First we solve the ODE exactly
    y = sp.Function('y')
    sol = sp.dsolve(sp.Eq(y(t).diff(t), rhs_dict[rhs_key](y(t), t)), y(t)).rhs
    constants = sp.solve([sol.subs(t,tn)-yn])
    exact = sol.subs(constants)

    # 2. Now we solve the ODE numerically using specified Butcher table

    # Access the requested Butcher table
    Butcher = Butcher_dict[Butcher_key][0]
    # Determine number of predictor-corrector steps
    L = len(Butcher)-1
    # Set a temporary array for update values
    k = np.zeros(L, dtype=object)
    # Initialize intermediate variable
    yhat = 0
    # Initialize the updated solution
    ynp1 = 0
    for i in range(L-1):
        #Initialize and approximate update for solution
        yhat = yn
        for j in range(i):
            # Update yhat for solution using a_ij Butcher table coefficients
```

```
            yhat += Butcher[i][j+1]*k[j]
            if Butcher_key == "DP8" or Butcher_key == "L6":
                yhat = 1.0*sp.N(yhat,20) # Otherwise the adding of fractions kills performance.
        # Determine the next corrector variable k_i using c_i Butcher table coefficients
        k[i] = dt*rhs_dict[rhs_key](yhat, tn + Butcher[i][0]*dt)
        # Update the solution at the next iteration ynp1 using Butcher table coefficients
        ynp1 += Butcher[L-1][i+1]*k[i]
    # Finish determining the solution for the next iteration
    ynp1 += yn

    # Determine the order of the RK method
    order = Butcher_dict[Butcher_key][1]+2
    # Produces Taylor series of exact solution at t=tn about t = 0 with the specified order
    exact_series = sp.series(exact.subs(t, dt),dt, 0, order)
    num_series = sp.series(ynp1, dt, 0, order)
    diff = exact_series-num_series
    return diff
```

[6]:
```
t, dt = sp.symbols('t dt')
# Set initial conditions
t0 = 0
y0 = 1
# Set RHS of ODE
toggle = 0 # This is a bookkeeping device for knowing when we reached the Inherently Adaptive methods
for key,value in Butcher_dict.items():
    if key in {"AHE", "ABS", "ARKF", "ACK", "ADP5", "ADP8"}:
        if (key == "AHE"):
            toggle = 1
        if (toggle == 1): #only do anything once we actually hit adaptive methods.
            print("RK method: \""+str(key)+"\".")
            y = sp.Function('y')
            print(" When solving y'(t) = "+str(rhs_dict[function](y(t),t))+", y("+str(t0)+")="+str(y0)+",")
            local_truncation_order = list(value)[1]+1
            print(" the first calculation's first nonzero term should have local truncation error proportional to␣
 ↪O(dt^"+str(local_truncation_order-1)+") or a higher power of dt.")
            print(" the second calculation's first nonzero term should have local truncation error proportional to␣
 ↪O(dt^"+str(local_truncation_order)+") or a higher power of dt.")
            print("Subtracting the numerical result from the exact Taylor expansion, we find a local truncation error of:")
            if key == "ADP8": #ADP8 takes up too much of the screen in analytic, we need to print as decimals.
                sp.pretty_print(ValidateARK1(key, y0, t0, function).evalf())
                sp.pretty_print(ValidateARK2(key, y0, t0, function).evalf())
            else:
                sp.pretty_print(ValidateARK1(key, y0, t0, function))
                sp.pretty_print(ValidateARK2(key, y0, t0, function))
```

```
        #     print("\n")
            print(" (Coefficients of order 1e-15 or less may generally be ignored, as these are at roundoff error.)\n")
```

```
RK method: "AHE".
 When solving y'(t) = t + y(t), y(0)=1,
 the first calculation's first nonzero term should have local truncation error
proportional to O(dt^2) or a higher power of dt.
 the second calculation's first nonzero term should have local truncation error
proportional to O(dt^3) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
        3
  2   dt     [U+239B]  4[U+239E]
dt  + [U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
       3
  3
dt     [U+239B]  4[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 3
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "ABS".
 When solving y'(t) = t + y(t), y(0)=1,
 the first calculation's first nonzero term should have local truncation error
proportional to O(dt^3) or a higher power of dt.
 the second calculation's first nonzero term should have local truncation error
proportional to O(dt^4) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
    3     4
  dt    dt     [U+239B]  5[U+239E]
- [U+2500][U+2500][U+2500] + [U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
   24    24
  4
dt     [U+239B]  5[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 12
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "ARKF".
 When solving y'(t) = t + y(t), y(0)=1,
 the first calculation's first nonzero term should have local truncation error
proportional to O(dt^5) or a higher power of dt.
```

```
 the second calculation's first nonzero term should have local truncation error
proportional to O(dt^6) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
    5     6
  dt    dt     [U+239B]  7[U+239E]
- [U+2500][U+2500][U+2500] + [U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
  390   360
     6
17·dt     [U+239B]  7[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 9360
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "ACK".
 When solving y'(t) = t + y(t), y(0)=1,
 the first calculation's first nonzero term should have local truncation error
proportional to O(dt^5) or a higher power of dt.
 the second calculation's first nonzero term should have local truncation error
proportional to O(dt^6) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
        5          6
  277·dt    4541·dt     [U+239B]  7[U+239E]
- [U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + [U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] +␣
 ↪O[U+239D]dt [U+23A0]
   614400   7372800
  6
dt      [U+239B]  7[U+239E]
[U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
3600
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "ADP5".
 When solving y'(t) = t + y(t), y(0)=1,
 the first calculation's first nonzero term should have local truncation error
proportional to O(dt^5) or a higher power of dt.
 the second calculation's first nonzero term should have local truncation error
proportional to O(dt^6) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
       5        6
  97·dt    17·dt     [U+239B]  7[U+239E]
```

```
- [U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + [U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
  60000    180000
    6
  dt      [U+239B]  7[U+239E]
- [U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
  1800
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

RK method: "ADP8".
 When solving y'(t) = t + y(t), y(0)=1,
 the first calculation's first nonzero term should have local truncation error
proportional to O(dt^8) or a higher power of dt.
 the second calculation's first nonzero term should have local truncation error
proportional to O(dt^9) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
                                                 2                           3
-7.71097267488672e-19·dt + 2.91023006428162e-18·dt  + 5.29778138968287e-19·dt

                          4                           5
 - 1.5349356222021e-19·dt  + 1.65528504999405e-19·dt  + 4.19247608610368e-20·d

 6                          7                           8
t  + 6.27844437078994e-21·dt  - 4.85333183539141e-7·dt  + 3.49344710134931e-7·

  9    [U+239B]  10[U+239E]
dt  + O[U+239D]dt  [U+23A0]
                                               2                           3
3.68531467298237e-18·dt + 5.84980541251108e-18·dt  + 3.78072846284061e-18·dt

                          4                           5
+ 9.53606673846474e-19·dt  + 1.26972675407126e-19·dt  + 3.90778437343018e-20·d

 6                          7                           8
t  + 4.80748915688373e-21·dt  + 5.17189453562972e-22·dt  + 7.20786458776279e-9

   9    [U+239B]  10[U+239E]
·dt  + O[U+239D]dt  [U+23A0]
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)
```

2.5 Step 2.e: Validating AB Methods against ODEs [Back to top]

The following code validates the AB methods, all of them from order 1 to 19. Their first remaining error term should be precisely one order higher than the method order.

Since AB methods have to validate using multiple past values, the validation function is hard-coded in, since we need exact solutions at those past points for this to work. Fortunately $y' = y$ is suitable for this purpose, as it sufficiently tests all the orders up to 19.

[7]:
```
#This is taken almost directly from the butcher validation from NRPy+

def ValidateAB(ABorder, yn, tn, rhs_key): # custom function for validating AB methods.

    order = ABorder

    # 1. First we solve the ODE exactly
    y = sp.Function('y')
    sol = sp.dsolve(sp.Eq(y(t).diff(t), fy(y(t), t)), y(t)).rhs
    constants = sp.solve([sol.subs(t,tn)-yn])
    exact = sol.subs(constants)
    exact_series = sp.series(exact.subs(t, dt),dt, 0, order+2)

    # 2. Now we solve the ODE numerically using specified Butcher table

    # Access the requested Butcher table
    Butcher = Butcher_dict.get("AB")[0]
    # Determine number of predictor-corrector steps
    L = len(Butcher)-1 #set to -2 for adaptive methods.
    # Set a temporary array for update values
    k = np.zeros(L, dtype=object)
    # Initialize intermediate variable
    yhat = 1
    # Initialize the updated solution
    ynp1 = 0
    for i in range(ABorder):
        # Initialize and approximate update for solution
        # Update yhat for solution using "past values",
        # Which means evaluating the exact answer at x=(-i*dt).
        # If the user wishes to validate another ODE, the respective section
        # In the below line will have to be changed.
        yhat += dt*Butcher[ABorder-1][i]*sp.exp((-i)*dt)
    # Finish determining the solution for the next iteration
    num_series = sp.series(yhat, dt, 0, order+2)

    # Produces Taylor series of exact solution at t=tn about t = 0 with the specified order
```

```
    diff = exact_series-num_series
    return diff
```

[8]:
```
t, dt = sp.symbols('t dt')
# Set initial conditions
t0 = 0
y0 = 1

i = 1
while i < 20:
    print("AB method order: \""+str(i)+"\".")
    y = sp.Function('y')
    print(" When solving y'(t) = y, y(0)=1,") # make this adaptable to all possible inputs.
    print(" the first nonzero term should have local truncation error proportional to O(dt^"+str(i+1)+") or a higher power of dt.")
    print("Subtracting the numerical result from the exact Taylor expansion, we find a local truncation error of:")
    sp.pretty_print(ValidateAB(i, y0, t0, function))
#     print("\n")
    print(" (Coefficients of order 1e-15 or less may generally be ignored, as these are at roundoff error.)\n")
    i = i+1
```

```
AB method order: "1".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^2) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
  2
dt     [U+239B]  3[U+239E]
[U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 2
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "2".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^3) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
    3
5·dt     [U+239B]  4[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
  12
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
```

```
roundoff error.)

AB method order: "3".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^4) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
    4
3·dt     [U+239B]  5[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
  8
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "4".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^5) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
      5
251·dt     [U+239B]  6[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
  720
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "5".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^6) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
     6
95·dt     [U+239B]  7[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 288
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "6".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^7) or a higher power of dt.
```

```
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
        7
19087·dt      [U+239B]   8[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
  60480
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "7".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^8) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
       8
5257·dt      [U+239B]   9[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt [U+23A0]
 17280
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "8".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^9) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
          9
1070017·dt      [U+239B]   10[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt  [U+23A0]
  3628800
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "9".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^10) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
        10
25713·dt      [U+239B]   11[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt  [U+23A0]
  89600
```

```
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "10".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^11) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
           11
26842253·dt      [U+239B]  12[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt  [U+23A0]
   95800320
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "11".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^12) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
          12
4777223·dt      [U+239B]  13[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt  [U+23A0]
  17418240
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "12".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^13) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
               13
703604254357·dt      [U+239B]  14[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500]␣
 ↪+ O[U+239D]dt  [U+23A0]
  2615348736000
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "13".
 When solving y'(t) = y, y(0)=1,
```

```
 the first nonzero term should have local truncation error proportional to
O(dt^14) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
                14
106364763817·dt      [U+239B]  15[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500]␣
 ↪+ O[U+239D]dt  [U+23A0]
   402361344000
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "14".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^15) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
                 15
1166309819657·dt      [U+239B]  16[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500]␣
 ↪+ O[U+239D]dt  [U+23A0]
  4483454976000
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "15".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^16) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
           16
25221445·dt      [U+239B]  17[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500] + O[U+239D]dt  [U+23A0]
   98402304
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "16".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^17) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
```

```
                    17
8092989203533249·dt      [U+239B]  18[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500]
 ↪+ O[U+239D]dt  [U+23A0]
  32011868528640000
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "17".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^18) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
                   18
85455477715379·dt      [U+239B]  19[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500]
 ↪+ O[U+239D]dt  [U+23A0]
  342372925440000
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "18".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^19) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
                        19
1260046723604275659·dt      [U+239B]  20[U+239E]
[U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500][U+2500]
 ↪+ O[U+239D]dt  [U+23A0]
   51090942171709440000
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)

AB method order: "19".
 When solving y'(t) = y, y(0)=1,
 the first nonzero term should have local truncation error proportional to
O(dt^20) or a higher power of dt.
Subtracting the numerical result from the exact Taylor expansion, we find a
local truncation error of:
                       20
1311546499957236437·dt      [U+239B]  21[U+239E]
```

```
[U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500] [U+2500]
 ↪+ O[U+239D]dt  [U+23A0]
  5377993912811520000
 (Coefficients of order 1e-15 or less may generally be ignored, as these are at
roundoff error.)
```

3 Step 3: Output this notebook to LaTeX-formatted PDF file [Back to top]

The following code cell converts this Jupyter notebook into a proper, clickable LaTeX-formatted PDF file. After the cell is successfully run, the generated PDF may be found in the root NRPy+ tutorial directory, with filename Tutorial-RK_Butcher_Table_Validation.pdf. (Note that clicking on this link may not work; you may need to open the PDF file through another means.)

[9]:
```python
import cmdline_helper as cmd    # NRPy+: Multi-platform Python command-line interface
cmd.output_Jupyter_notebook_to_LaTeXed_PDF("Tutorial-RK_Butcher_Table_Validation")
```

```
Created Tutorial-RK_Butcher_Table_Validation.tex, and compiled LaTeX file
    to PDF file Tutorial-RK_Butcher_Table_Validation.pdf
```