Integration Of ODEs Homework 3

MinWook Kang

February 2, 2023

1 Introduction Of Some ODE Solvers

1.1 Euler's Method

상미분 방정식(ODE)의 문제는 1차 미분 방정식의 집합으로 축소된다. 아래의 예를 들면,

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x) \tag{1}$$

의 문제의 경우, 이는 치환을 통하여 두개의 1차 방정식으로 써질 수 있다.

$$\begin{aligned} \frac{dy}{dx} &= z \\ \frac{dz}{dx} &= r(x) - q(x)z \end{aligned} \tag{2}$$

여기서 z는 새로운 변수이다. 위의 방식처럼, 우리는 N차 미분방정식을 1차의 미분 방정식 집합으로 축소시킬 수 있다.

$$\frac{dy_i}{dx} = f_i(x, y_0, \ldots, y_{N-1}), \qquad i = 0, \ldots, N-1 \tag{3}$$

그리고 우리는 이것을 벡터화 해서 나타낼 수 있는데, 이를 뒤에 나올 numpy 배열 혹은 리스트를 이용하여 $F(x, y)$함수를 정의하여 문제를 풀어볼 것이다. 우선 위의 식을 벡터화 하면,

$$\mathbf{Y}(x) = \begin{pmatrix} y_0(x) \\ \vdots \\ y_{N-1}(x) \end{pmatrix} \quad \text{and} \quad \mathbf{F}(x, \mathbf{Y}) = \begin{pmatrix} f_0(x, y_0, \ldots, y_{N-1}) \\ \vdots \\ f_{N-1}(x, y_0, \ldots, y_{N-1}) \end{pmatrix} \tag{4}$$

이고 이를, 우리가 위에서 예제로 든 2번식을 벡터화 하게되면 다음과 같다.

$$\mathbf{Y} = \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} y \\ z \end{pmatrix} \quad \text{and} \quad \mathbf{F} = \begin{pmatrix} f_0 \\ f_1 \end{pmatrix} = \begin{pmatrix} z \\ r(x) - q(x)z \end{pmatrix} \tag{5}$$

상미분 방정식을 푸는 문제는 방정식에 의해 완벽하게 서술되지는 않는다. 문제를 수치적으로 풀려면 가장 중요한 것은 문제의 경계조건의 성질을 이해하는 것인데, 이는 다음과 같다.

- 초기 가치 문제에서 모든 y는 일부 시작값 x_s에서 주어지고, 최종점 x_f까지의 목록에서 y를 찾는 것이 바람직하다.

- 두 점 경계 값 문제에서 경계 조건은 둘 이상의 x로 저장된다. 일반적으로 일부 조건은 x_s에서 저장되고, 나머지 조건은 x_f에서 저장된다.

이렇게 초기 가치가 주어진 문제에서, 2차 미분 방정식을 1차 미분 방정식의 집합으로 축소시켜, 문제를 풀어볼 것이다. 상미분 방정식을 푸는 해결책은 근본적으로 Euler's Method를 따르고 있다. 초기 가치가 주어진 문제를 풀기 위해서 우리는 두가지를 따라야 한다.

- dy와 dx들을 유한한 단계인 Δ_y와 Δ_x로 쓴다.
- Δ_x로 방정식을 곱한다.

$$\frac{dy}{dx} = f(x,y) \quad \rightarrow \quad \Delta y = f(x,y)\Delta x \tag{6}$$

즉, $y(x_0) = y_0$의 초기 조건이 주어진 공식에서 오일러 방법의 공식은

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{7}$$

이고, 여기서 $h = x_{n+1} - x_n$이다. 이는 Stepsize라고 부른다. 이때, 독립적인 변수 x가 하나의 Δ_x의 Stepsize에 의해 결정된다. 만약 Stepsize를 매우 작게 만들면 상미분 방정식에 대한 좋은 근사치를 가지게 되는데, 이는 에러값인 $|y_n - y_{x_n}|$가 작아지게 된다. 하지만 h가 작아지면 더 많은 계산처리를 해야한다는 단점이 존재한다. 따라서 오일러 방법은 간격만이 정확도를 발전시킨다.

1.2 Predictor-corrector Method

예측 교정 방법은, 상미분 방정식을 통합하도록 설계된 알고리즘인데, 이러한 모든 알고리즘은 두가지의 단계를 따른다.

- 초기의 예측 단계는 점 집합의 함수값과 미분값에 맞는 함수에서 시작하여 새로운 지점에서 이 함수의 값을 추정한다.
- 교정자 단계는 함수의 예측 값과 동일한 후속 지점에서 알 수 없는 함수의 값을 보간하는 다른 방법을 사용하여 초기 근사치를 구체화 한다.

예측 교정 방법은 상미분 방정식의 수치 해를 고려할 때, 일반적으로 예측자 단계에 대한 명시적인 방법과 교정 단계에 대한 암시적인 방법을 사용한다. Stepsize를 h로 두고 위에서 보인 오일러 방법을 예로 들면,

$$y' = f(t, y), \qquad y(t_0) = y_0 \tag{8}$$

에서 예측 단계와 교정자 단계를 보일 것이다. 먼저 예측 단계는, 현재의 값 y_i에서 시작하여 오일러 방법을 통해 계산한 $y_{i+1}^{\sim}$을 구하는 방법이다.

$$\tilde{y}_{i+1} = y_i + hf(t_i, y_i) \tag{9}$$

이는 이미 알고 있는 y_i에서 유도한 것이다. 교정자 단계에서는 사다리꼴 규칙으로 추측한 값을 개선할 것이다.

$$y_{i+1} = y_i + h\frac{f(t_i, y_i) + f(t_{i+1}, \tilde{y}_{i+1})}{2} \tag{10}$$

그리고 이 교정자 단계는 여러번 반복될 수 있으며, 이를 통하여 초기의 근사치를 보간하는 방식을 가지는 알고리즘이 바로 예측 교정 방법이다.

1.3 Runge-Kutta Method

Runge-Kutta 방법은 오일러 방법을 포함하는 방식이다. 우선 $\frac{dy}{dx} = f(x, y)$식을 다음과 같이 정의할 것이다.

$$y_{n+1} = y_n + ak_1 + bk_2 \tag{11}$$

이때, k_1과 k_2는 각각

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + Ah, y_n + Bk_1) \end{aligned} \tag{12}$$

이다. 이를 테일러 전개를 하면 다음과 같이 되는데,

$$y_{n+1} = y_n + hf(x_n, y_n) + \frac{h^2}{2}f'(x_n, y_n) + O(h^3) \tag{13}$$

이때, f'을 f_x와 같이 마찬가지로 y에 대해서도 간단하게 정리하게 되어주면

$$y_{n+1} = y_n + hf_n + \frac{h^2}{2}(f_x + ff_y)_{x=x_n} \tag{14}$$

11번 식을 12번식으로 대입하여 정리하게 되면,

$$y_{n+1} = y_n + ahf(x_n, y_n) + bhf(x_n + Ah, y_n + Bhf(x_n, y_n)) \tag{15}$$

이고, 여기에 마지막 항을 고차항을 무시하고 테일러 전개하면 다음과 같다.

$$f(x_n + Ah, y_n + Bhf(x_n, y_n)) \approx f(x_n, y_n) + Ahf_x(x_n, y_n) + Bhf(x_n, y_n)f_y(x_n, y_n) \tag{16}$$

이 식을 15번 식의 마지막 항에 대입하면

$$\begin{aligned} y_{n+1} &= y_n + ahf(x_n, y_n) + bh(f(x_n, y_n) + Ahf_x(x_n, y_n) + Bhf(x_n, y_n)f_y(x_n, y_n)) \\ &= y_n + h(a+b)f_n + h^2(Abf_x + Bbff_y)_{x=x_n} \end{aligned} \tag{17}$$

가 나온다. 이때, 14번식과 계수를 비교하면 a와 b에 대한 식을 얻을 수 있는데 따라서 우리는 $a+b=1$, $Ab=\frac{1}{2}$ 그리고 $Bb=\frac{1}{2}$을 얻을 수 있다. 따라서 앞서서 구한 식 11번은

$$y_{n+1} = y_n + \frac{1}{2}(k_1 + k_2) \tag{18}$$

과 같고, 이것이 바로 Euler-Heun의 방법이다. 이때 a의 다른 값을 선택할 수 있는데, 이것이 2차 Runge-Kutta 방법이다. 우리가 앞서서 구한 방식으로 비슷하게 3차에 대해서도 풀어주게 되면,

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3) + O(h^4) \tag{19}$$

가 나오고 여기서 오차는 $O(h^3)$값을 가지게 된다. 이는 우리가 고려하지 않은 나머지 h에 대한 함수를 뜻하는 것으로, 만약 차수를 올리게 되면 이러한 오류 또한 h의 n차만큼 늘 것이고, 이를 우리는 local error라고 부른다.

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1\right) \\ k_3 &= hf\left(x_n + h, y_n + 2k_2 - k_1\right) \end{aligned} \tag{20}$$

이다. 하지만, 지금까지 가장 많이 사용되는 방식은 4차 Runge-Kutta 방법이다. 테일러 전개를 h^4까지 반복하면 마찬가지로,

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5) \tag{21}$$

를 구할 수 있고 여기서 우리는 local error = $O(h^5)$을 가진다. 그리고 global error = $O(h^4)$을 가진다.

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1\right) \\ k_3 &= hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2\right) \\ k_4 &= hf\left(x_n + h, y_n + k_3\right) \end{aligned} \tag{22}$$

1.4 Adaptive Stepsize Control

Adaptive stepsize의 방법은 오류를 제어하기 위해 일부 방법에서 사용된다. 이 방법의 목적은 최소한의 계산 노력으로 솔루션에서 달성하고자 하는 정확성을 가지는 해를 구하기 위한 것이다. Adoptive stepssize를 사용하는 것은 도함수의 크기에 큰 차이가 있을 때 특히 중요하다. $y'(t) = f(t, y(t)), y(a) = y_a$ 식에서 t = b일 때의 근인 $y(b)$를 구할려면, t값에 대한 리스트 t_n을 $t_n = a + nh$로 두고, $y(t_n)$에 대한 추정을 하면 다음과 같다.

$$y_{n+1}^{(0)} = y_n + hf(t_n, y_n) \tag{23}$$

이때, 위의 추정에서 local truncation error은 이렇게 정의된다.

$$\tau_{n+1}^{(0)} = y(t_{n+1}) - y_{n+1}^{(0)} = ch^2 \tag{24}$$

여기서 c는 비례상수이다. 이제 두번째 근사치를 생성하기 위해 다른 단계의 크기로 오일러 방식을 적용하는데, 이때 새로운 stepsize를 원래 stepsize의 반이 되도록 정의하여 적용하면 다음과 같다.

$$\begin{aligned} y_{n+\frac{1}{2}} &= y_n + \frac{h}{2}f(t_n, y_n) \\ y_{n+1}^{(1)} &= y_{n+\frac{1}{2}} + \frac{h}{2}f\left(t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}}\right) \\ \tau_{n+1}^{(1)} &= y(t_{n+1}) - y_{n+1}^{(1)} \end{aligned} \tag{25}$$

이때 오일러 방법을 두번 적용했기 때문에 local error은 극단적인 경우 실제 에러의 절반일 것이다. 따라서 에러는

$$\tau_{n+1}^{(1)} = c\left(\frac{h}{2}\right)^2 + c\left(\frac{h}{2}\right)^2 = 2c\left(\frac{h}{2}\right)^2 = \frac{1}{2}ch^2 = \frac{1}{2}\tau_{n+1}^{(0)} \tag{26}$$

이때 우리가 주목할 점은 local error 추정에서 stepsize h는 원하는 추정치를 달성하게끔 만들 수 있는

데, local tolerance을 정해준 상태에서 h를 아래 처럼 표현할 수 있다.

$$h \to 0.9 \times h \times \min\left(\max\left(\left(\frac{\text{tol}}{2\left|\tau_{n+1}^{(1)}\right|}\right)^{1/2}, 0.3\right), 2\right) \tag{27}$$

그리고 이것은 이후에 나올 Runge-Kutta 방법에서 stepsize를 정의하는 방식으로 h를 우리가 원하는 오차에 맞게끔 적용할 것이다.

1.5 Runge-Kutta Method With Embedded Stepsize Control

4차 Runge-Kutta를 사용하면 위의 예처럼 Step Doubling을 적용할 수 있는데, 우리는 해를 구하는 방식으로 한번은 완전한 단계로, 그리고 그다음은 독립적인 두번의 반 단계의 방식을 거칠 것이다. 이로써, 우리는 원래 Runge-Kutta 단계보다 더 정확한 오류 추정치를 얻을 수 있다. 그리고 Embedded Stepsize Control 알고리즘은 Fehlberg에 의해 발명된 내장된 Runge-Kutta 공식을 기반으로 한다. Fehlberg는 6개의 함수를 평가하는 5차 방법을 발견했다. 5차 Runge-Kutta 공식의 일반적인 형태는 다음과 같다.

$$\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf(x_n + a_2 h, y_n + b_{21} k_1) \\
k_3 &= hf(x_n + a_3 h, y_n + b_{31} k_1 + b_{32} k_2) \\
&\cdots \\
k_6 &= hf(x_n + a_6 h, y_n + b_{61} k_1 + \cdots + b_{65} k_5) \\
y_{n+1} &= y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 + O(h^6)
\end{aligned} \tag{28}$$

그리고 내장된 공식은 다음과 같다.

$$y_{n+1}^* = y_n + c_1^* k_1 + c_2^* k_2 + c_3^* k_3 + c_4^* k_4 + c_5^* k_5 + O(h^5) \tag{29}$$

그리고 오류 추정치는

$$\Delta \equiv y_{n+1} - y_{n+1}^* = \sum_{i=1}^{6} (c_i - c_i^*)\, k_i \tag{30}$$

이다. 여기서 사용하는 다양한 상수 값은 Cash와 Karp에 의해 정의 되어 있다. 우리는 에러의 값이 무엇인지 알기 때문에, 우리가 원하는 오차 범위내에 에러 값이 유지되도록 만들 것이다. 그러면 우리는

아래처럼 추정할 수 있다.

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2} \tag{31}$$

이때 31번 식에서 Δ_0을 우리가 원하는 정확도로 만들기 위해 두가지 방법이 쓰여진다.

- 만약Δ_1값이 Δ_0보다 규모가 크면 31번식은 우리가 현재 단계를 다시 시도할 때 stepsize를 얼마나 줄여야 하는지 알려준다.
- 만약Δ_1값이 Δ_0보다 규모가 작으면 31번식은 우리가 현재 단계를 다시 시도할 때 stepsize를 얼마나 안전하게 늘릴 수 있는지 알려준다.

2 SciPy ODE Solvers

2.1 Module Understanding

이번에는 SciPy 모듈을 통해서, 예제문제를 풀 것이다. Scipy에서 solve_ivp 함수를 사용할 것이다. 이 함수에는 다양한 인수값이 존재하는데, 우선 하나 하나씩 정리해보면 다음과 같다.

```
solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, dense_output=False, events=
    None, vectorized=False, args=None, **options)[source]
```

- f: function 구하고자 하는 함수이다.
- t_span2 : 적분 간격 (t0, tf)

  solver는 $t = t_0$로 시작하여 $t = t_f$에 도달할 때까지 적분한다. t_0과 t_f는 모두 플로트 변환 함수로 해석할 수 있는 부동 소수점 또는 값이어야 한다.

- y_0: initial state

  만약 초기값이 complex가 아닌 real 값이더라도, complex의 domain 문제의 경우 반드시 complex 꼴로 보내야 한다.

- method : method string or Odesolver, 선택사항

  RK45 (기본값) : Order 5의 명시적인 Runge-Kutta Method, 오류는 4차 방법의 정확성을 가정하여 제어되지만, 5차 정확한 공식을 사용하여 단계가 수해오딘다. 복잡한 domain에 적용 될 수 있다.

  RK23 : Order 3의 명시적인 Runge-Kutta Method, 오류는 2차 방법의 정확성을 가정하여 제어되고, 3차 공식을 사용하여 단계를 수행한다.

  DOP853 : Order 8의 Runge-Kutta Method, 7차 순으로 정확한 7차 보간 다항식이 고밀도 출력에 사용된다.

  Radau : Radau 2a 계열의 order 5의 Runge-Kutta method 오류는 3차 embedded 공식으로 제어된다.

  BDF : 미분 근사치에 대한 역방향 분화 공식을 기반으로 한 암시적 다단계 변수 순서 방법(1에서 5)

  등이 존재한다.

외에도 몇가지 설정을 할 수 있는 선택사항 인수들이 존재했는데, 이 문제에서 주로 다룰때 쓸 dense output 기능만 간단히 소개하자면, 이는 지속적인 해결책을 계산할지를 알려주는 설정값이다. 기본값은 bool = False로 되어있기 때문에 앞으로 쓸 해를 구하는 식에서는 True로 사용한다.

이때 이 함수의 return 값을 정리하면 다음과 같다.

```
return (t, y, sol, t_events, y_events, nev, nfev, njev, nlu, status, message,
    success)
```

- t : ndarray 꼴이다. time points를 나타낸다.
- y : ndarray 꼴이다. t의 solution에 대한 value 값들이다.
- sol : Odesoultion 으로 구한 solution을 의미한다. 급격히 증가하거나 감소하는 지역에서 어느 한 지점이 주어지면, 그 지점에 대한 보간법을 통하여 일시적으로 구한 해를 도출해준다. 이 간격을 벗어난 평가의 정확도는 보장되지 않는다. dense output이 False로 설정된 경우는 없다.

외에도 여러가지 return값이 있지만, 문제를 접근하는데 필요한 정보들만 간추려 보았다.

2.2 Problem Recognition

먼저 풀어볼 문제는 아래와 같다.

$$y' = y \tag{32}$$

초기조건은 $x = [0, 2]$ 이고 $y(0) = 1$ 이 주어져 있다. 위 함수의 실제 해는 아래와 같다.

$$y(x) = e^x \tag{33}$$

우리는 문제를 풀기 위해서 Scipy에서 solve_ivp 함수에 필요한 인수 값인 f, xinit, y0, method, dense_output 을 원하는 조건에 맞추어서 수정하여 문제를 풀어볼 것이다. 또한 xinit에 따라서 구해지는 Odesolution 과 실제 해 사이에서의 에러의 크기도 확인해 볼 것이다. 이를 통하여 method에 따른 정확도의 차이에 대해서도 확인해 볼 것이다.

2.3 Development of a solution

먼저 함수는 dy/dx의 우변꼴이 우리가 구하고자 하는 함수이다. 즉 위 함수 $y' = y$ 에서 y에 해당하는 우변이 바로 우리가 구하고자 하는 함수의 function 값이므로, 이를 먼저 정의 한후, 위의 조건식에 맞추어서 xinit 과 y0 값을 수정한다. 방식은 Runge-Kutta 5차식의 방식을 이용할 것이다.

```
sol = solve_ivp(f, [0, 2], [1], method='RK45', dense_output = True)

def f(x, y):
    return y

b = []
b.append(*a.sol(x))

i = 0
y_true = np.exp(x)

x_true = np.linspace(0, 2, 200)
y_true = np.exp(x_true)

plt.figure(figsize=[11, 4])

plt.subplot(1, 3, 1)
x = np.linspace(0, 2, 5)
plt.plot(x_true, y_true, "-b", x, *sol.sol(x), "o:r")
plt.legend(["$e^x$", f"$h = {x[1] - x[0]}$"])
plt.grid()
plt.xlabel("$x$")
plt.ylabel("$y$")

plt.subplot(1, 3, 2)
x = np.linspace(0, 2, 11)
plt.plot(x_true, y_true, "-b", x, *sol.sol(x), "o:r")
plt.legend(["$e^x$", f"$h = {x[1] - x[0]}$"])
plt.grid()
plt.xlabel("$x$")

plt.subplot(1, 3, 3)
x = np.linspace(0, 2, 21)
plt.plot(x_true, y_true, "-b", x, *sol.sol(x), "o:r")
plt.legend(["$e^x$", f"$h = {x[1] - x[0]}$"])
plt.grid()
plt.xlabel("$x$")

plt.tight_layout()
```

```
plt.show()
```

2.4 Execution and Assessment

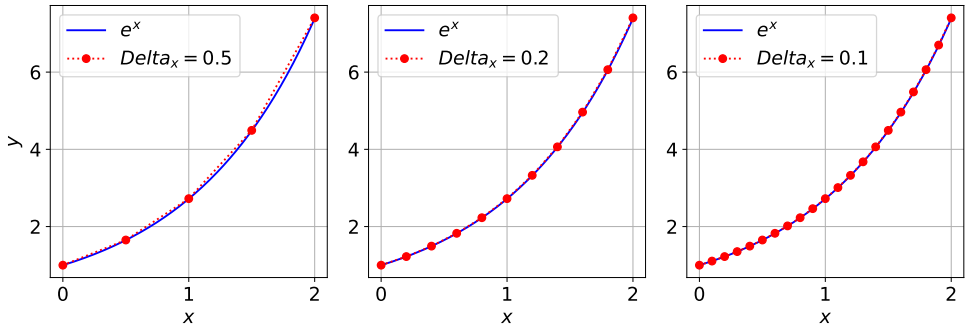

Figure 1: $y' = y$

여기서 sol.sol(x)의 의미는 우리가 지정한 구간에서 구한 sol에서 x의 구간이 촘촘해질수록, 곡률이 더 매끄러워지는 것을 볼 수 있다. 위 sol 함수는 조밀한 출력이라고 하는 다항식 보간을 구현한다. 우리는 원하는 구간내에서의 $y' = y$ 의 상미분 방정식의 해값을 이렇게 구할 수 있다. 이때, 위에서 확인한 Method들은 각기 다른 쓰임이 존재했는데, 고밀도 제어가 필요한 경우의 Method는 오차값이 당연히 작을 것으로 예상되기 때문에 이를 다른 방식을 넣어 오차의 크기를 확인하려고 한다. 먼저 RK45의 경우 오차는 다음과 같다.

```
def f(x, y):
    return y

a = solve_ivp(f, [0, 2], [1], method='RK23', dense_output = True)
t = np.linspace(0, 2, 5)

def error(linspace, real_function, a): #a = solve_ivp(f, [0, 2], [1], method='RK45',
     dense_output = True)
    t = linspace
    i = 0
    y_true = real_function(t)
    b = []
    b.append(*a.sol(t))
    for i, c in enumerate(y_true):
```

```
        print((f"error ({t[0]} ~ {t[-1]}) :{y_true[i] - b[0][i]}"))

print(error(np.linspace(0, 2, 5), lambda x: np.exp(x), a))
print(error(np.linspace(0, 2, 11), lambda x: np.exp(x), a))
print(error(np.linspace(0, 2, 21), lambda x: np.exp(x), a))
```

```
error (0.0 ~ 2.0) :0.0
error (0.0 ~ 2.0) :-4.156935837329456e-06
error (0.0 ~ 2.0) :-2.1237760380099502e-05
error (0.0 ~ 2.0) :-3.1839687372858805e-05
error (0.0 ~ 2.0) :-0.000269366909522617
None
error (0.0 ~ 2.0) :0.0
error (0.0 ~ 2.0) :-3.1015876831963496e-05
error (0.0 ~ 2.0) :-5.8258354160845016e-05
error (0.0 ~ 2.0) :7.074363789305593e-05
error (0.0 ~ 2.0) :0.00013152556591711217
error (0.0 ~ 2.0) :-2.1237760380099502e-05
error (0.0 ~ 2.0) :-0.00017003513837998696
error (0.0 ~ 2.0) :-0.00017410584998689416
error (0.0 ~ 2.0) :0.00011726622438601453
error (0.0 ~ 2.0) :9.377831619072907e-05
error (0.0 ~ 2.0) :-0.000269366909522617
None
error (0.0 ~ 2.0) :0.0
error (0.0 ~ 2.0) :-2.5798541081201165e-10
error (0.0 ~ 2.0) :-3.1015876831963496e-05
error (0.0 ~ 2.0) :-6.603546164218876e-05
error (0.0 ~ 2.0) :-5.8258354160845016e-05
error (0.0 ~ 2.0) :-4.156935837329456e-06
error (0.0 ~ 2.0) :7.074363789305593e-05
error (0.0 ~ 2.0) :0.00012758126396317238
error (0.0 ~ 2.0) :0.00013152556591711217
error (0.0 ~ 2.0) :7.097324756744072e-05
error (0.0 ~ 2.0) :-2.1237760380099502e-05
error (0.0 ~ 2.0) :-6.78728845775467e-05
error (0.0 ~ 2.0) :-0.00017003513837998696
error (0.0 ~ 2.0) :-0.00022855513413411188
```

```
error (0.0 ~ 2.0) :-0.00017410584998689416
error (0.0 ~ 2.0) :-3.1839687372858805e-05
error (0.0 ~ 2.0) :0.00011726622438601453
error (0.0 ~ 2.0) :0.00017913007328118624
error (0.0 ~ 2.0) :9.377831619072907e-05
error (0.0 ~ 2.0) :-0.00011247594008079176
error (0.0 ~ 2.0) :-0.000269366909522617
None
```

DOP853의 Method의 경우 고밀도 출력에서 많이 사용하는 방식이다. Runge-Kutta Method의 7차 보간 다항식을 구한 것이기 때문에, 높은 정확도를 가지고 있음을 알 수 있다. 실제로 RK45와 비교하였을때 오차크기가 어느정도인지를 확인하면 다음과 같은 결과가 나온다.

```
error (0.0 ~ 2.0) :0.0
error (0.0 ~ 2.0) :-3.7969912620727797e-06
error (0.0 ~ 2.0) :-9.106313882867312e-05
error (0.0 ~ 2.0) :0.00011141117893398444
error (0.0 ~ 2.0) :3.745382305098133e-05
None
error (0.0 ~ 2.0) :0.0
error (0.0 ~ 2.0) :1.2968071061436603e-11
error (0.0 ~ 2.0) :-6.698338987032315e-06
error (0.0 ~ 2.0) :-8.925274852522591e-07
error (0.0 ~ 2.0) :-2.8524980141941825e-05
error (0.0 ~ 2.0) :-9.106313882867312e-05
error (0.0 ~ 2.0) :-9.637758558689313e-05
error (0.0 ~ 2.0) :2.360861782157997e-05
error (0.0 ~ 2.0) :0.00018518408810930254
error (0.0 ~ 2.0) :0.000180844920722123
error (0.0 ~ 2.0) :3.745382305098133e-05
None
error (0.0 ~ 2.0) :0.0
error (0.0 ~ 2.0) :-6.2232441422338525e-12
error (0.0 ~ 2.0) :1.2968071061436603e-11
error (0.0 ~ 2.0) :-2.7468468677405156e-06
error (0.0 ~ 2.0) :-6.698338987032315e-06
error (0.0 ~ 2.0) :-3.7969912620727797e-06
error (0.0 ~ 2.0) :-8.925274852522591e-07
```

```
error (0.0 ~ 2.0) :-7.640067075431745e-06
error (0.0 ~ 2.0) :-2.8524980141941825e-05
error (0.0 ~ 2.0) :-5.993031038409313e-05
error (0.0 ~ 2.0) :-9.106313882867312e-05
error (0.0 ~ 2.0) :-0.00010753605664470811
error (0.0 ~ 2.0) :-9.637758558689313e-05
error (0.0 ~ 2.0) :-5.1220706422050455e-05
error (0.0 ~ 2.0) :2.360861782157997e-05
error (0.0 ~ 2.0) :0.00011141117893398444
error (0.0 ~ 2.0) :0.00018518408810930254
error (0.0 ~ 2.0) :0.0002148185506332112
error (0.0 ~ 2.0) :0.000180844920722123
error (0.0 ~ 2.0) :9.618907302844093e-05
error (0.0 ~ 2.0) :3.745382305098133e-05
None
```

예상했던대로, x의 범위중 경계값에 해당하는x_f에 가까울수록 오차가 커지는 경우를 위에서 볼 수 있었는데, DOP853의 경우 경계값에서도 10^{-5}정도로 비교적 작은 값을 가지는 것을 볼 수 있다. 이외에도 RK23을 사용했을시, 당연하게 45보다 정확도가 떨어지는 것을 볼 수 있다.

2.5 Problem Recognition

이번에는 $y' = y^2$을 풀어볼 것이다. 이때 $y(0) = -1$라는 값이 주어져 있다. 위 미분방정식의 해는 다음과 같다.

$$y(x) = \frac{-1}{x+1} \tag{34}$$

2.6 Development of a solution

이번에는 LSODA의 방법으로 풀어볼 것이다. 위와 같은 방법으로 코드를 짜면 다음과 같다.

```
def f(x, y):
    return y ** 2


x = np.linspace(0, 5, 16)
sol = solve_ivp(f, [0, 5], [-1], method='LSODA', dense_output = True)

x_true = np.linspace(0, 5, 100)
y_true = -1/(x_true + 1)
```

```
plt.figure()
plt.plot(x, *sol.sol(x), ".:r", x_true, y_true, "-b")
plt.legend(["Heun", "$-1/(x + 1)$"])
plt.grid()
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.show()

print(error(x, lambda x: -1/(x + 1), sol))
```

```
error (0.0 ~ 5.0) :0.0
error (0.0 ~ 5.0) :0.0005107166633844251
error (0.0 ~ 5.0) :0.0006179231012554132
error (0.0 ~ 5.0) :0.00047276511962957013
error (0.0 ~ 5.0) :0.0004744789574998576
error (0.0 ~ 5.0) :0.00040843393484457646
error (0.0 ~ 5.0) :0.00034521895747735565
error (0.0 ~ 5.0) :0.00035161754804069467
error (0.0 ~ 5.0) :0.0003150987249875281
error (0.0 ~ 5.0) :0.00027771933041087493
error (0.0 ~ 5.0) :0.0002471999955599713
error (0.0 ~ 5.0) :0.00022737869869338123
error (0.0 ~ 5.0) :0.00023806847005869436
error (0.0 ~ 5.0) :0.00021690415046862754
error (0.0 ~ 5.0) :0.000203802342367998
error (0.0 ~ 5.0) :0.00017921160683101456
None
```

2.7 Execution and Assessment

예상했던대로, 실제 값과 매우 정교하게 나온 것을 볼 수 있다. 오차를 보면, LSODA의 경우 같은 방식을 RK45로 한 것 보다 오차가 작게 나왔지만 정교한 작업이 필요로 하는 방식에서는 맞지 않는다는 것을 볼 수 있다. 마찬가지로 DOP853의 방식으로 에러 값을 측정하면 다음과 같이 매우 작은 에러값이 나온다.

```
error (0.0 ~ 5.0) :0.0
```

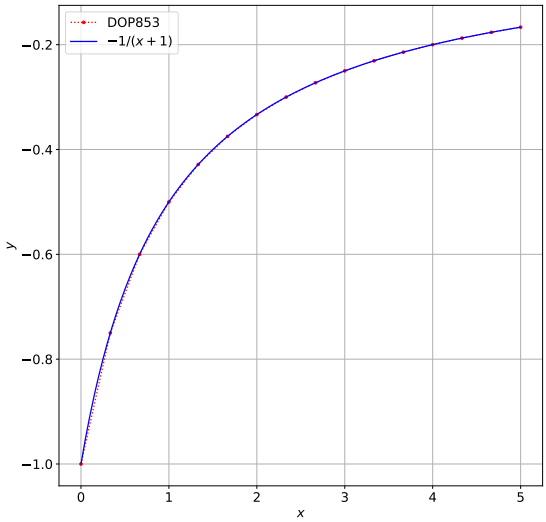

Figure 2: $y' = y$

```
error (0.0 ~ 5.0) :4.2580274439707466e-07
error (0.0 ~ 5.0) :1.0940012791893494e-05
error (0.0 ~ 5.0) :-9.795822744851357e-06
error (0.0 ~ 5.0) :-7.599695994731093e-08
error (0.0 ~ 5.0) :9.940520382389906e-07
error (0.0 ~ 5.0) :-1.6535692408115032e-06
error (0.0 ~ 5.0) :7.0241504279699996e-09
error (0.0 ~ 5.0) :4.0402445056209046e-08
error (0.0 ~ 5.0) :9.497284830239927e-07
error (0.0 ~ 5.0) :6.244511590869362e-07
error (0.0 ~ 5.0) :-1.6603436716611242e-06
error (0.0 ~ 5.0) :-1.0250484894225309e-07
error (0.0 ~ 5.0) :-1.560800641509097e-08
error (0.0 ~ 5.0) :-1.385852790858344e-08
error (0.0 ~ 5.0) :-1.2380110631093899e-08
None
```

2.8 Problem Recognition

이번에는 뉴턴의 두번째 법칙에 대해서 풀어볼 것이다. 중력상수에 대한 운동방정식은 다음과 같다.

$$\frac{dx^2}{dt^2} = -g \tag{35}$$

이 문제의 실제 솔루션은 다음과 같다.

$$x(t) = -\frac{1}{2}gt^2 + v_0t + x_0 \tag{36}$$

이때 $x_0 = 0$이며 $v_0 = 1$이 주어졌을 때, 높이와 속도에 대한 그래프를 구해볼 것이다. 범위는 0부터 2/g일때까지 구해야 한다.

2.9 Development of a solution

예제에 나와있는 코드 처럼 2차 미분방정식을 벡터화 한 후, 각각에 대한 값을 구해준다. 이때, Y[1]에 해당하는 값은 v이다. 이 v는 -g로 인해 1차 미분 방정식으로 구해지는 값이기 때문에, 그값이 구해지면, 그것이 다시 Y 변수로 가게 되면서 순차적으로 x도 적분할 수 있게 된다. 또한, 우리는 그래프를 그려야 하는데, sol.sol(t)로 구해지는 값을 리스트 x, v 에 순차적으로 저장하여 그 값을 그래프로 그려볼 것이다.

```
g = 9.8 # m/s
def F(t, Y): # F = [v, -g], Y = [x, v]
    F = [Y[1], -g]
    return np.array(F)

t    = np.linspace(0, 2/g, 100)
Y_0  = np.array([0, 1])

#Using Zip
x = []
v = []

for a, b in zip(*sol.sol(t)):
    x.append(a)
    v.append(b)

sol = solve_ivp(F, [0, 2/g], Y_0, method='LSODA', dense_output = True)
```

```
19
21 plt.figure(figsize=[11, 4])
23 plt.subplot(1, 2, 1)
   plt.plot(t, -g/2*t**2 + t, '-', t, x, ':k')
25 plt.legend(["exact", "Heun"])
   plt.xlabel("$t$ [s]")
27 plt.ylabel("$x$ [m]")
29 plt.subplot(1, 2, 2)
   plt.plot(t, -g*t + 1, '-', t, v, ':k')
31 plt.legend(["exact", "Heun"])
   plt.xlabel("$t$ [s]")
33 plt.ylabel("$v$ [m/s]")
35 plt.tight_layout()
   plt.show()
```

2.10 Execution and Assessment

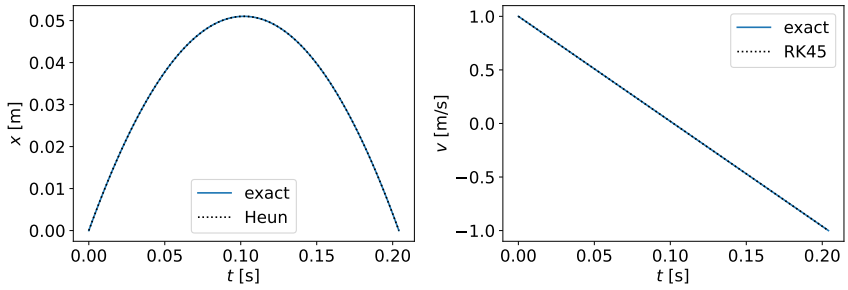

Figure 3: Newton's 2nd Law

exact 값과 상당히 유사하게 나온 것을 볼 수 있다. 이것에 대한 오차값은 위에서 구한 방식과 동일하게 구하면 되므로 생략한다.

2.11 Problem Recognition

이번에 풀어볼 문제는 Particle in a Box이다. 먼저 이 문제는