`Scipy.integrate.solve_ivp`

(an updated version of `odeint`)

scipy.integrate.solve_ivp

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

Solve an initial value problem for a system of ODEs.

This function numerically integrates a system of ordinary differential equations given an initial value:

```
dy / dt = f(t, y)
y(t0) = y0
```

Here t is a 1-D independent variable (time), y(t) is an N-D vector-valued function (state), and an N-D vector-valued function f(t, y) determines the differential equations. The goal is to find y(t) approximately satisfying the differential equations, given an initial value y(t0)=y0.

Some of the solvers support integration in the complex domain, but note that for stiff ODE solvers, the right-hand side must be complex-differentiable (satisfy Cauchy-Riemann equations [11]). To solve a problem in the complex domain, pass y0 with a complex data type. Another option always available is to rewrite your problem for real and imaginary parts separately.

Parameters: fun : *callable*

Right-hand side of the system. The calling signature is `fun(t, y)`. Here *t* is a scalar, and there are two options for the ndarray *y*: It can either have shape (n,); then *fun* must return array_like with shape (n,). Alternatively, it can have shape (n, k); then *fun* must return an array_like with shape (n, k), i.e., each column corresponds to a single column in *y*. The choice between the two options is determined by *vectorized* argument (see below). The vectorized implementation allows a faster approximation of the Jacobian by finite differences (required for stiff solvers).

t_span : *2-tuple of floats*

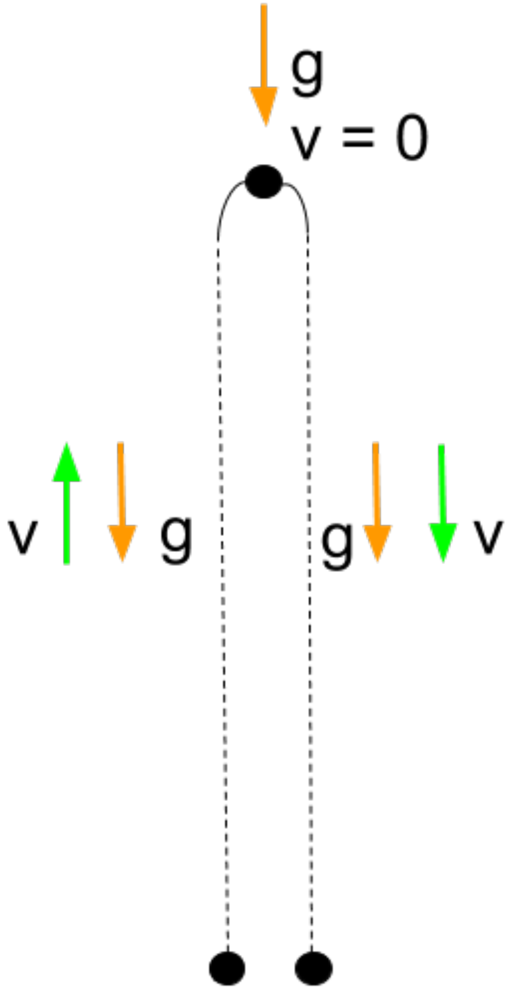
Interval of integration (t0, tf). The solver starts with t=t0 and integrates until it reaches t=tf.

y0 : *array_like, shape (n,)*

Initial state. For problems in the complex domain, pass *y0* with a complex data type (even if the initial value is purely real).

method : *string or OdeSolver, optional*

$$\frac{d^2h}{dt^2} = -g$$



$y = y_0 + v_0\Delta t + ½\, a(\Delta t)^2$

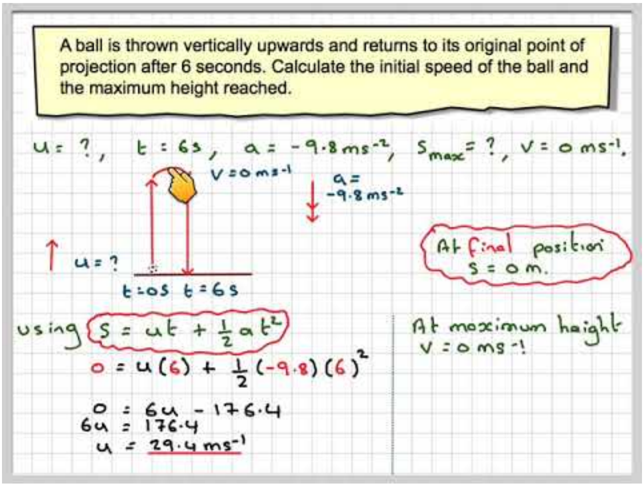
$y_3 = y_1 + v_1\Delta t + ½\, a(\Delta t)^2$

$0 = 1.2\text{ m} + (6.4\text{ m/s})\,\Delta t + ½\,(-9.8\text{ m/s}^2)(\,\Delta t)^2$

$0 = 1.2\text{ m} + (6.4\text{ m/s})\,\Delta t - (4.9\text{ m/s}^{2)}(\,\Delta t)^2$

$\Delta t = -0.16\text{ s or } +1.5\text{ s}$

$\Delta t = 1.5\text{ s}$



$$\frac{d^2h}{dt^2} = -g \quad \Rightarrow \quad \frac{d\left(\frac{dh}{dt}\right)}{dt} = -g \quad \Rightarrow \quad \frac{dh}{dt} = v, \qquad \frac{dv}{dt} = -g$$

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

Solve an initial value problem for a system of ODEs.

This function numerically integrates a system of ordinary differential equations given an initial value:

```
dy / dt = f(t, y)
y(t0) = y0
```

```
def upward_cannon(t, y):
```

$$\frac{dh}{dt} = v$$

$$\frac{dv}{dt} = -g$$

```python
def upward_cannon(t, y):

    h = y[0]
    v = y[1]

    dhdt = v
    dvdt = -9.807

    dydt = np.array([dhdt, dvdt])
    return dydt
```

```python
def upward_cannon(t, y):
    return np.array([y[1], -9.807])
```

From
`solve_ivp`

$$t\,,\quad y = \begin{bmatrix} p \\ \vdots \\ s \end{bmatrix}$$

```
def our_ODE_function(t,y):
    .
    .
    .
    return dydt
```

Back to
`solve_ivp`

$$\frac{dy}{dt} = \begin{bmatrix} \frac{dp}{dt} \\ \vdots \\ \frac{ds}{dt} \end{bmatrix}$$

SciPy.org

SciPy.org | Docs | SciPy v1.6.0 Reference Guide | Integration and ODEs (scipy.integrate)

scipy.integrate.solve_ivp

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

Solve an initial value problem for a system of ODEs.

This function numerically integrates a system of ordinary differential equations given an initial value:

```
dy / dt = f(t, y)
y(t0) = y0
```

Here t is a 1-D independent variable (time), y(t) is an N-D vector-valued function (state), and an N-D vector-valued function f(t, y) determines the differential equations. The goal is to find y(t) approximately satisfying the differential equations, given an initial value y(t0)=y0.

Some of the solvers support integration in the complex domain, but note that for stiff ODE solvers, the right-hand side must be complex-differentiable (satisfy Cauchy-Riemann equations [11]). To solve a problem in the complex domain, pass y0 with a complex data type. Another option always available is to rewrite your problem for real and imaginary parts separately.

Parameters: **fun** : *callable*

Right-hand side of the system. The calling signature is `fun(t, y)`. Here *t* is a scalar, and there are two options for the ndarray *y*: It can either have shape (n,); then *fun* must return array_like with shape (n,). Alternatively, it can have shape (n, k); then *fun* must return an array_like with shape (n, k), i.e., each column corresponds to a single column in *y*. The choice between the two options is determined by *vectorized* argument (see below). The vectorized implementation allows a faster approximation of the Jacobian by finite differences (required for stiff solvers).

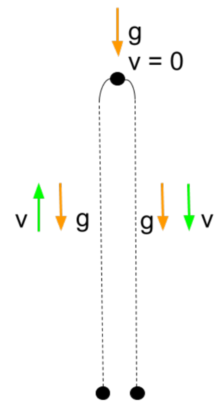
t_span : *2-tuple of floats*

Interval of integration (t0, tf). The solver starts with t=t0 and integrates until it reaches t=tf.

y0 : *array_like, shape (n,)*

Initial state. For problems in the complex domain, pass *y0* with a complex data type (even if the initial value is purely real).

method : *string or OdeSolver, optional*

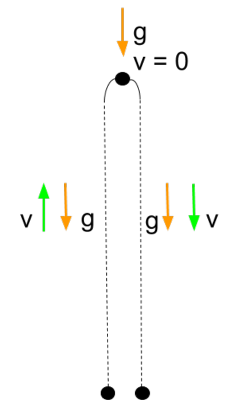



```python
def upward_cannon(t, y):
    return np.array([y[1], -9.807])


y_init = [30, 50]

sim_time = np.linspace(0, 30, 1000)


result = solve_ivp(upward_cannon,
                   (sim_time[0],sim_time[-1]),
                   y_init,
                   t_eval = sim_time
                  )
```

Returns: Bunch object with the following fields defined:

t : *ndarray, shape (n_points,)*
Time points.

y : *ndarray, shape (n, n_points)*
Values of the solution at *t*.



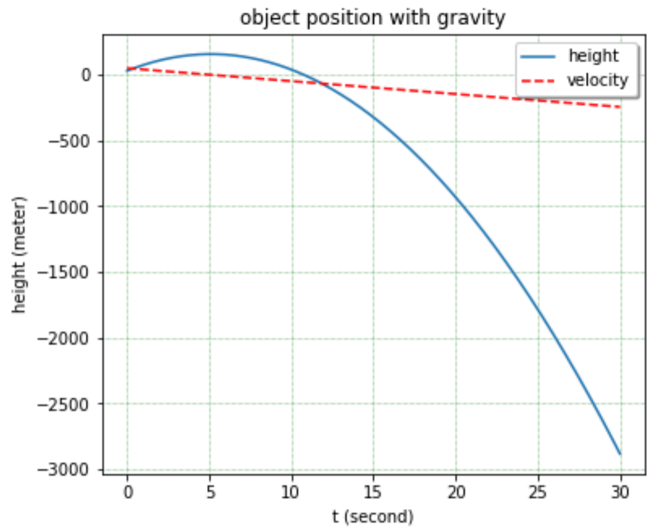

```
print(result)
result.success      True
result.y.shape      (2, 1000)
```

```
def upward_cannon(t, y):
    return np.array([y[1], -9.807])
```

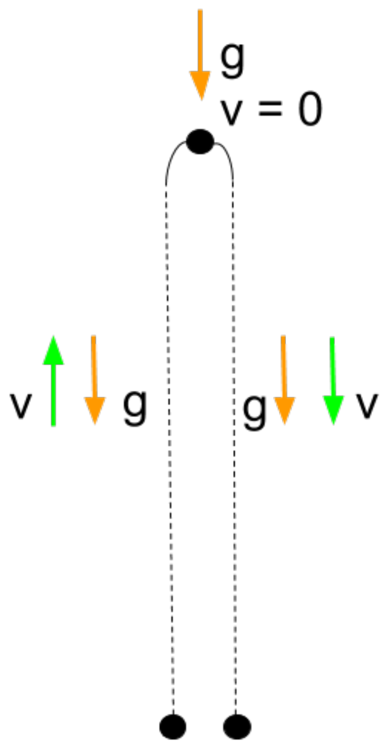
```
result.t       # time
result.y[0]    # h
result.y[1]    # v
```

```
y_init = [30, 50]

sim_time = np.linspace(0, 30, 1000)

result = solve_ivp(upward_cannon,
                   (sim_time[0],sim_time[-1]),
                   y_init,
                   t_eval = sim_time
                  )
```

```
import matplotlib.pyplot as plt

plt.figure(1, figsize=(6,5))
plt.plot(result.t, result.y[0])
plt.plot(result.t, result.y[1],'r--')
plt.legend(['height', 'velocity'], shadow=True)
plt.xlabel('t (second)')
plt.ylabel('height (meter)')
plt.title('object position with gravity')
plt.grid(color='g', linestyle=':', linewidth=0.5)
```



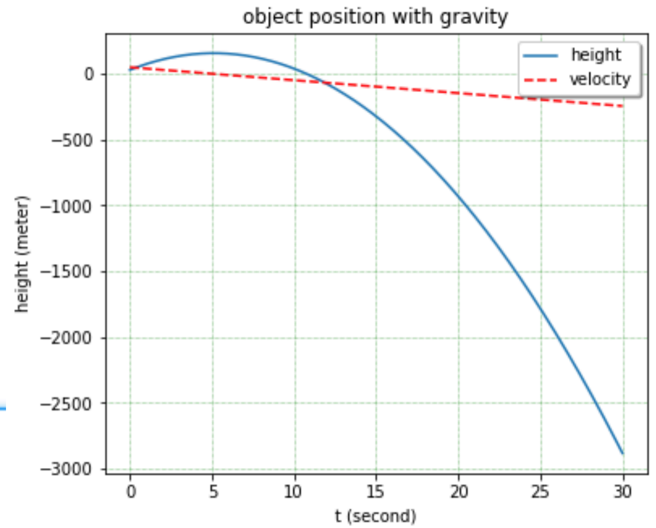
$$\frac{d^2h}{dt^2} = -g$$



$$\frac{dh}{dt} = v$$

$$\frac{dv}{dt} = -g$$

```
import numpy as np
from scipy.integrate import solve_ivp

def upward_cannon(t, y):
    return np.array([y[1], -9.807])

y_init = [30, 50]
sim_time = np.linspace(0, 30, 1000)

result = solve_ivp(upward_cannon,
                   (sim_time[0],sim_time[-1]),
                   y_init,
                   t_eval = sim_time
                  )

print('success?', result.success)

result.t        # time
result.y[0]     # h
result.y[1]     # v
```



What if our object has a booster?

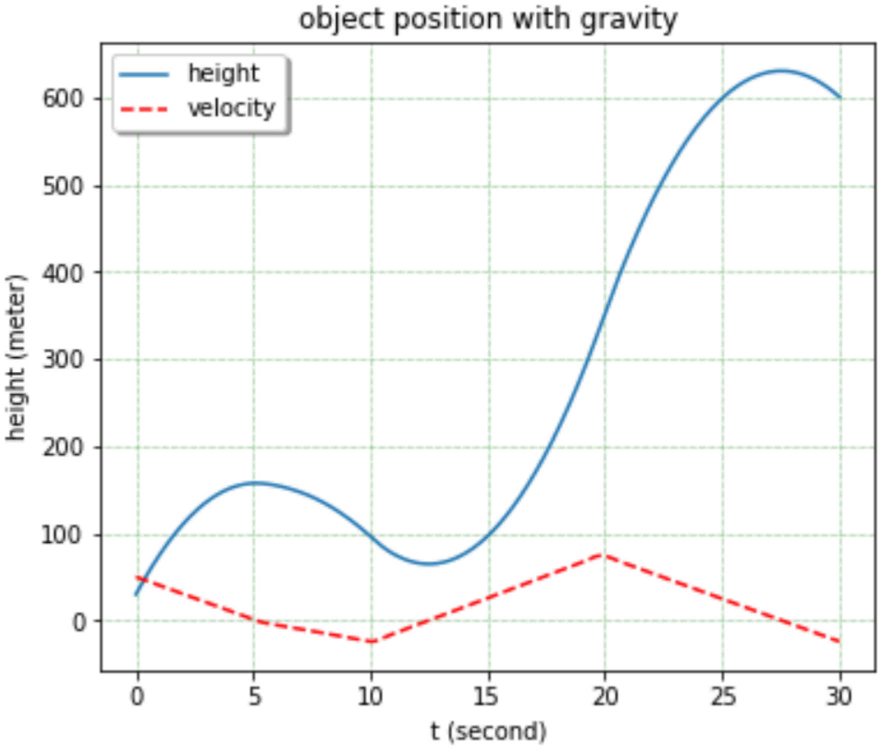
The upward acceleration by the booster is like this:

$$\frac{dh}{dt} = v$$

$$\frac{dv}{dt} = -g + a$$

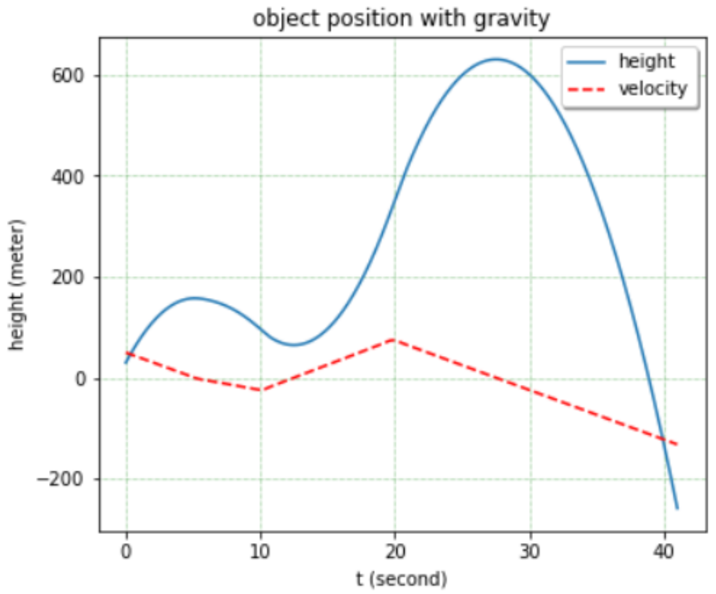
```
0<= t <5     → a = 0    (no boosting)
5<= t <10    → a = 5    (initial boosting)
10<= t <20   → a = 20   (full-power boosting)
20<= t       → a = 0    (out of fuel)
```

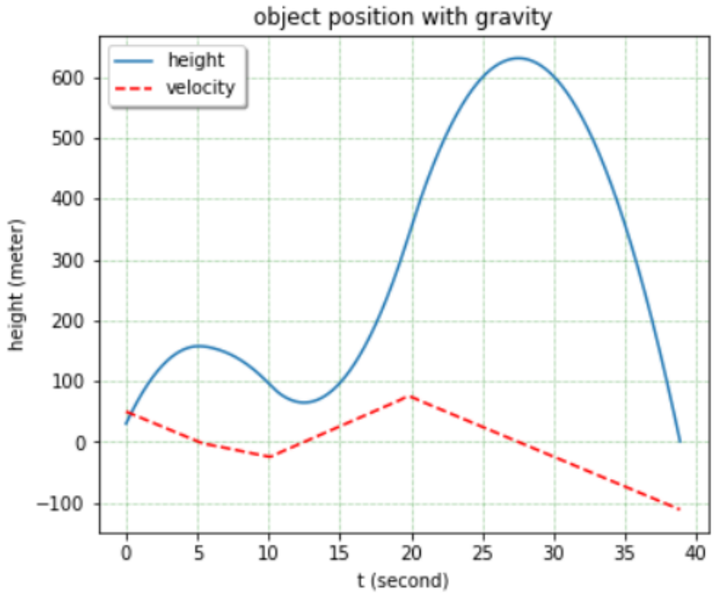
```python
def upward_cannon(t, y):
    h = y[0]
    v = y[1]

    if 5<=t<10:
        a = 5
    elif 10<=t<20:
        a = 20
    else:
        a = 0

    dhdt = v
    dvdt = -9.807 + a

    return np.array([dhdt, dvdt])
```



What if we extend the simulation time but want to stop when the object hit the ground?

```
def hit_ground(t, y):
    h = y[0]
    return h

hit_ground.terminal = True
hit_ground.direction = -1

result = solve_ivp(upward_cannon,
                   (sim_time[0],sim_time[-1]),
                   y_init,
                   t_eval = sim_time,
                   events = hit_ground
                  )
```





```python
import numpy as np
from scipy.integrate import solve_ivp

def upward_cannon(t, y):
    h = y[0]
    v = y[1]

    if 5<=t<10:
        a = 5
    elif 10<=t<20:
        a = 20
    else:
        a = 0

    dhdt = v
    dvdt = -9.807 + a

    return np.array([dhdt, dvdt])

def hit_ground(t, y):
    h = y[0]
    return h

hit_ground.terminal = True
hit_ground.direction = -1
```

```python
y_init = [30, 50]
sim_time = np.linspace(0, 50, 1000)

result = solve_ivp(upward_cannon,
                   (sim_time[0],sim_time[-1]),
                   y_init,
                   t_eval = sim_time,
                   events = hit_ground
                  )

print('success?', result.success)

result.t       # time
result.y[0]    # h
result.y[1]    # v
```

$$\frac{dh}{dt} = v$$

$$\frac{dv}{dt} = -g + a$$