SIMULATION


Sébastien Boisgérault

CONTROL ENGINEERING WITH PYTHON

- 📖 Course Materials
- © License CC BY 4.0
- 🏦 ITN, Mines Paris - PSL University

SYMBOLS

🐍	Code	🔍	Worked Example
📈	Graph	🧩	Exercise
🏷️	Definition	💻	Numerical Method
💎	Theorem	🧮	Analytical Method
📝	Remark	🧠	Theory
ℹ️	Information	🗝️	Hint
⚠️	Warning	🔓	Solution

🐍 IMPORTS

```
from numpy import *
from numpy.linalg import *
from matplotlib.pyplot import *
from scipy.integrate import solve_ivp
```

STREAM PLOT HELPER

```python
def Q(f, xs, ys):
    X, Y = meshgrid(xs, ys)
    fx = vectorize(lambda x, y: f([x, y])[0])
    fy = vectorize(lambda x, y: f([x, y])[1])
    return X, Y, fx(X, Y), fy(X, Y)
```

🏷️ SIMULATION

Numerical approximation solution $x(t)$ to the IVP

$$\dot{x} = f(x),\ x(t_0) = x_0$$

on some finite **time span** $[t_0, t_f]$.

🏷️ EULER SCHEME

Pick a (small) fixed **time step** $\Delta t > 0$.

Then use repeatedly the approximation:

$$
\begin{aligned}
x(t+\Delta t) &\simeq x(t) + \Delta t \times \dot{x}(t) \\
&= x(t) + \Delta t \times f(x(t)) \\
x(t+2\Delta t) &\simeq x(t+\Delta t) + \Delta t \times \dot{x}(t+\Delta t) \\
&= x(t+\Delta t) + \Delta t \times f(x(t+\Delta t)) \\
x(t+3\Delta t) &\simeq \cdots
\end{aligned}
$$

to compute a sequence of states $x_k \simeq x(t+k\Delta t)$.

🐍 EULER SCHEME

```
def basic_solve_ivp(f, t_span, y0, dt=1e-3):
    t0, t1 = t_span
    ts, xs = [t0], [y0]
    while ts[-1] < t1:
        t, x = ts[-1], xs[-1]
        t_next, x_next = t + dt, x + dt * f(x)
        ts.append(t_next); xs.append(x_next)
    return (array(ts), array(xs).T)
```

📖 USAGE - ARGUMENTS

- `f`, vector field (n-dim $\rightarrow$ n-dim),
- `t_span`, time span `(t0, t1)`,
- `y0`, initial state (n-dim),
- `dt`, time step.

📖 USAGE - RETURNS

- `t`, 1-dim array

  `t = [t0, t0 + dt, ...]`.

- `x`, 2-dim array, shape `(n, len(t))`

  `x[i][k]`: value of `x_i(t_k)`.

🔍 ROTATION

$$\left|\begin{array}{l}\dot{x}_1 = -x_2 \\ \dot{x}_2 = +x_1\end{array}\right. \quad \text{with} \quad \left|\begin{array}{l}x_1(0) = 1 \\ x_2(0) = 0\end{array}\right.$$

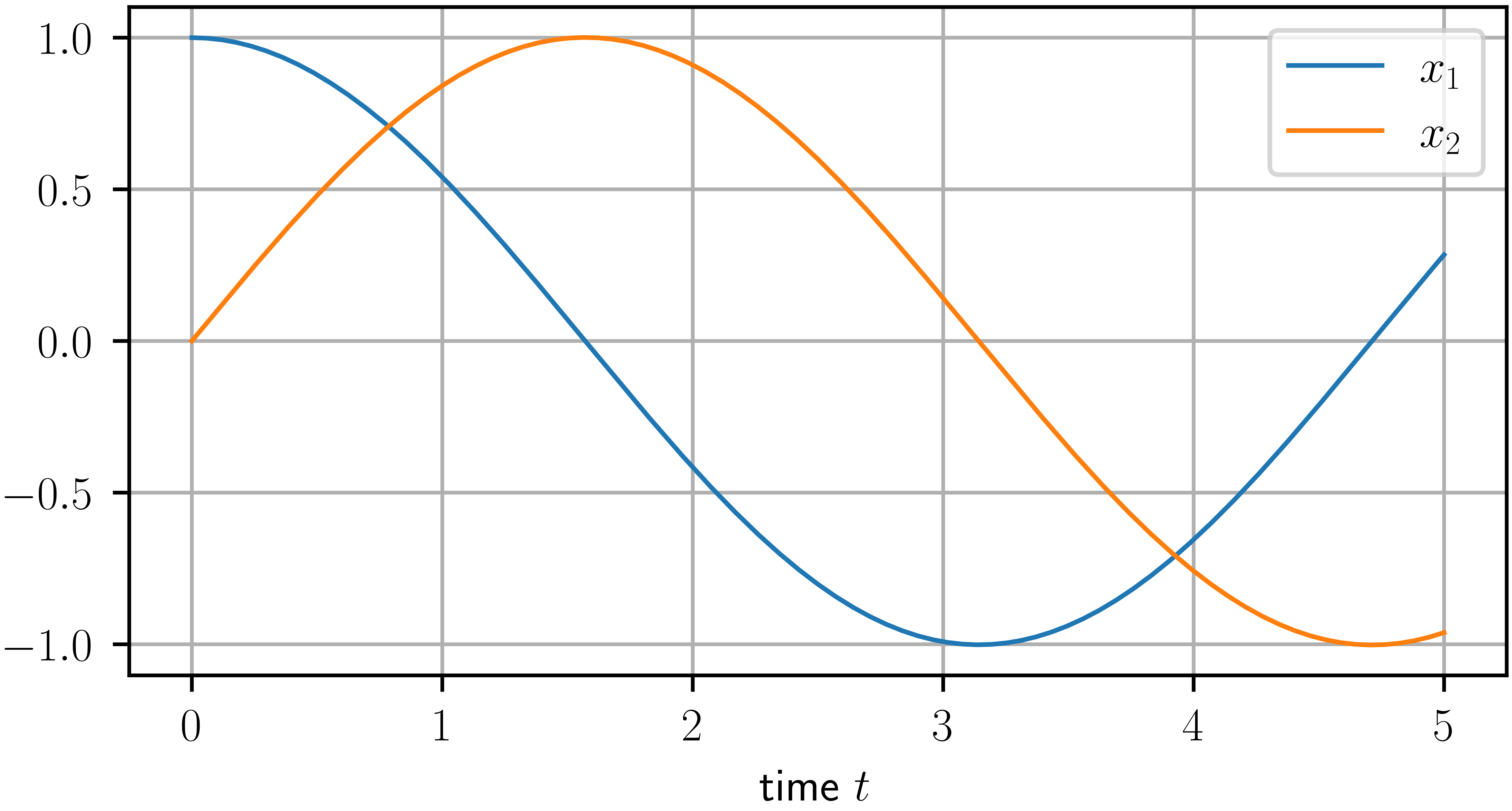
🐍 💻

```
def f(x):
    x1, x2 = x
    return array([-x2, x1])
t0, t1 = 0.0, 5.0
y0 = array([1.0, 0.0])

t, x = basic_solve_ivp(f, (t0, t1), y0)
```

📈 TRAJECTORIES

```
figure()
plot(t, x[0], label="$x_1$")
plot(t, x[1], label="$x_2$")
grid(True)
xlabel("time $t$")
legend()
```

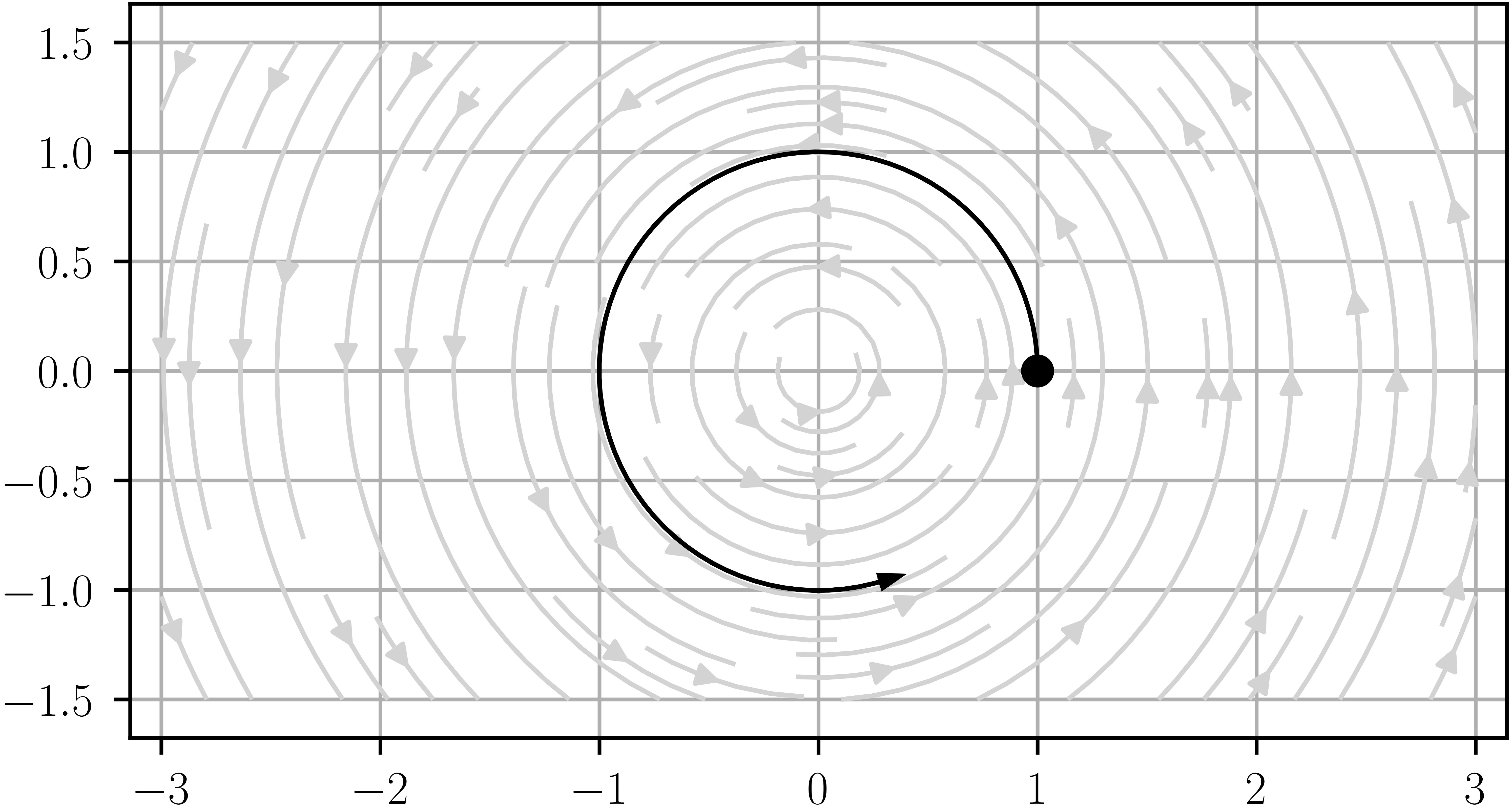

1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
time t
x_1
x_2

📈 TRAJECTORY (STATE SPACE)

```
def plot_trajectory_in_state_space(x):
    x1, x2 = x[0], x[1]
    plot(x1, x2, "k");
    plot(x1[0], x2[0], "ko")
    dx1, dx2 = x1[-1] - x1[-2], x2[-1] - x2[-2]
    arrow(x1[-1], x2[-1], dx1, dx2,
          width=0.02, color="k", zorder=10)
```

📈 STREAM PLOT + TRAJECTORY

```
figure()
xs = linspace(-3.0, 3.0, 50)
ys = linspace(-1.5, 1.5, 50)
streamplot(*Q(f, xs, ys), color="lightgrey")
plot_trajectory_in_state_space(x)
axis("equal"); grid(True)
```


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−3
−2
−1
0
1
2
3

⚠️ DON’T DO THIS AT HOME!

Now that you understand the basics

- ☠️ **Do NOT use this basic solver (anymore)!**
- ☠️ **Do NOT roll your own ODE solver !**

Instead

- ❤️ **Use a feature-rich and robust solver.**

(Solvers are surprisingly hard to get right.)

📖 SCIPY INTEGRATE

Use (for example):

```
from scipy.integrate import solve_ivp
```

📖 Documentation: `solve_ivp`

Features: time-dependent vector field, error control, dense outputs, multiple integration schemes, etc.

🔍 ROTATION

Compute the solution $x(t)$ for $t \in [0, 2\pi]$ of the IVP:

$$\left|\begin{aligned}\dot{x}_1 &= -x_2 \\ \dot{x}_2 &= +x_1\end{aligned}\right. \quad \text{with} \quad \left|\begin{aligned}x_1(0) &= 1 \\ x_2(0) &= 0\end{aligned}\right.$$

ROTATION

```python
def fun(t, y):
    x1, x2 = y
    return array([-x2, x1])
t_span = [0.0, 2*pi]
y0 = [1.0, 0.0]
result = solve_ivp(fun=fun, t_span=t_span, y0=y0)
```

⚠️ NON-AUTONOMOUS SYSTEMS

The solver is designed for time-dependent systems:

$$\dot{x} = f(t, x)$$

The `t` argument in the definition of `fun` is mandatory, even if the returned value doesn't depend on it (when the system is effectively time-invariant).

🐍 RESULT "BUNCH"

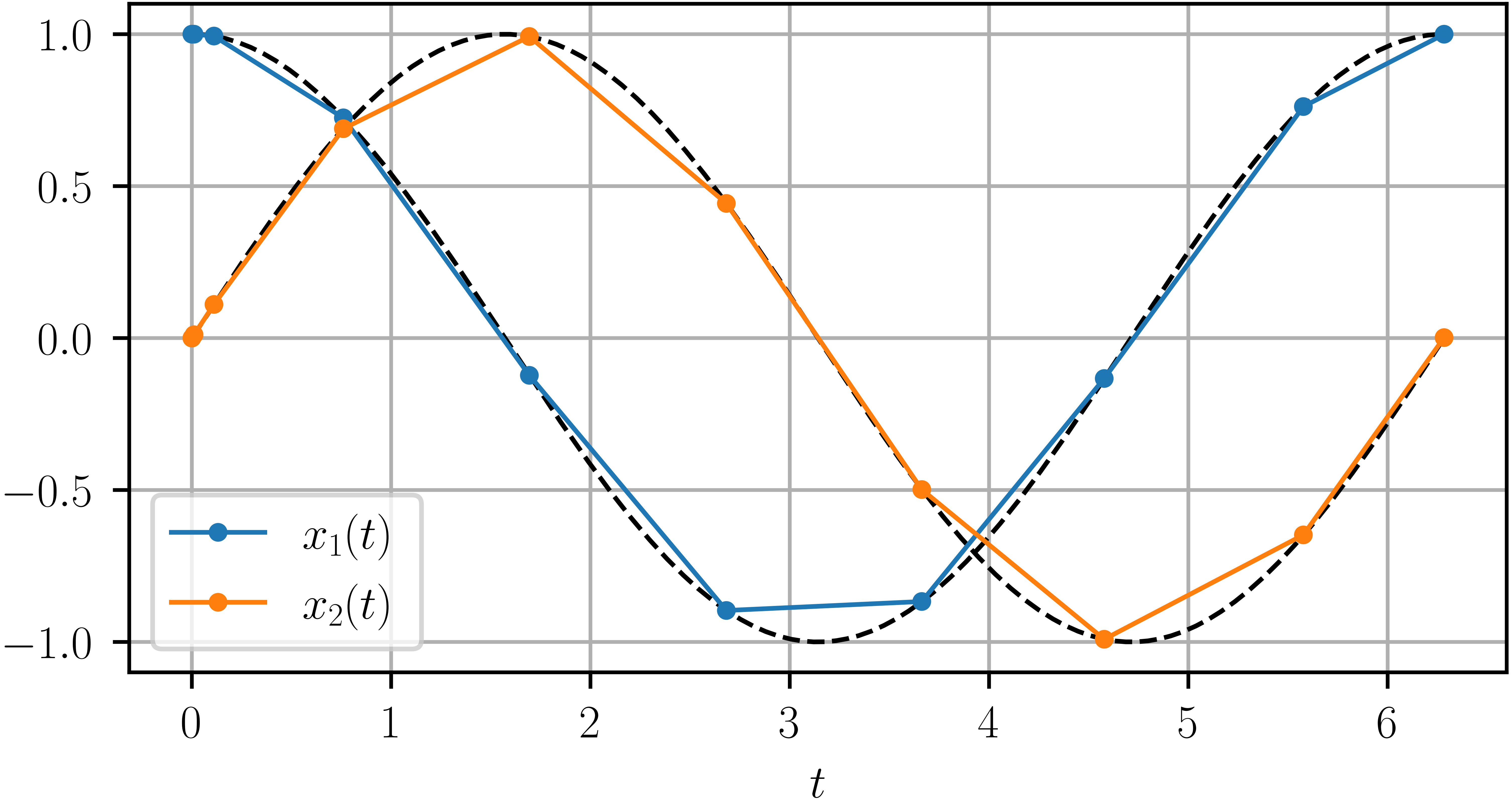
The `result` is a dictionary-like object with attributes:

- `t` : array, time points, shape `(n_points,)`,
- `y` : array, values of the solution at `t`, shape `(n, n_points)`,
- ...

(See 📖 `solve_ivp` documentation)

```python
rt = result["t"]
x1 = result["y"][0]
x2 = result["y"][1]
```

```
figure()
t = linspace(0, 2*pi, 1000)
plot(t, cos(t), "k--")
plot(t, sin(t), "k--")
plot(rt, x1, ".-", label="$x_1(t)$")
plot(rt, x2, ".-", label="$x_2(t)$")
xlabel("$t$"); grid(); legend()
```



1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6
t
$x_1(t)$
$x_2(t)$

🏷️ VARIABLE STEP SIZE

The step size is:

- **variable**: $t_{n+1} - t_n$ may not be constant,
- **automatically selected** by the solver,

The solver shall meet the user specification, but should select the largest step size to do so to minimize the number of computations.

Optionally, you can specify a `max_step` (default: $+\infty$).

🏷️ ERROR CONTROL

We generally want to control the (local) error $e(t)$: the difference between the numerical solution and the exact one.

- `atol` is the **absolute tolerance** (default: 10^{-6}),
- `rtol` is the **relative tolerance** (default: 10^{-3}).

The solver ensures (approximately) that at each step:

$$|e(t)| \leq \text{atol} + \text{rtol} \times |x(t)|$$

🔍 🐍 SOLVER OPTIONS

Example:

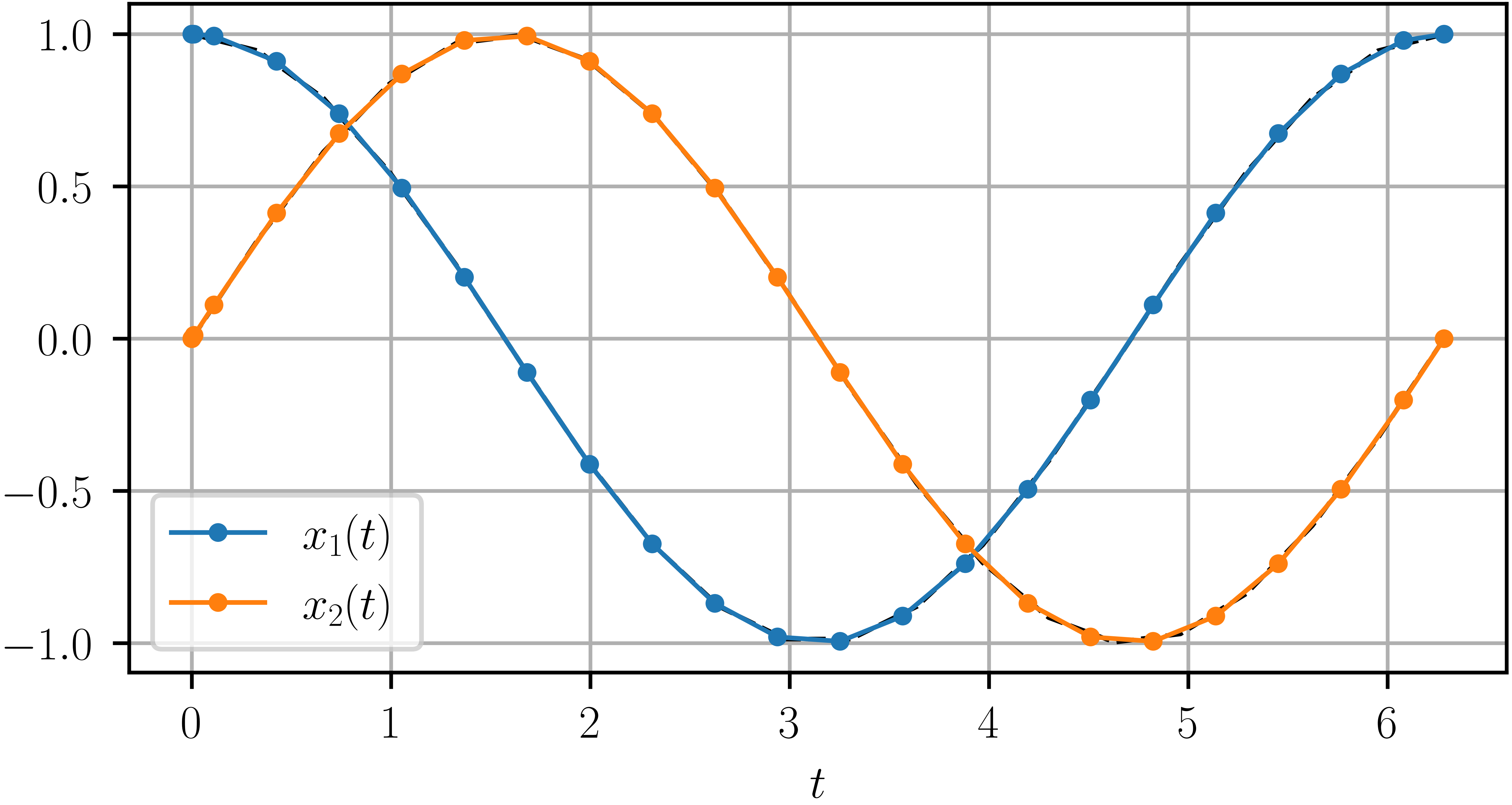
```
options = {
    # at least 20 data points
    "max_step": 2*pi/20,
    # standard absolute tolerance
    "atol"     : 1e-6,
    # very large relative tolerance
    "rtol"     : 1e9
}
```

🐍 SIMULATION

```
result = solve_ivp(
    fun=fun, t_span=t_span, y0=y0,
    **options
)
rt = result["t"]
x1 = result["y"][0]
x2 = result["y"][1]
```

📈 GRAPH

```
figure()
t = linspace(0, 2*pi, 20)
plot(t, cos(t), "k--")
plot(t, sin(t), "k--")
plot(rt, x1, ".-", label="$x_1(t)$")
plot(rt, x2, ".-", label="$x_2(t)$")
xlabel("$t$"); grid(); legend()
```



1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6
t
$x_1(t)$
$x_2(t)$

🏷️ DENSE OUTPUTS

Using a small `max_step` is usually the wrong way to "get more data points" since this will trigger many (potentially expensive) evaluations of `fun`.

Instead, use dense outputs: the solver may return the discrete data `result["t"]` and `result["y"]` **and** an approximate solution `result["sol"]` **as a function of t** with little extra computations.

🔍 🐍 SOLVER OPTIONS

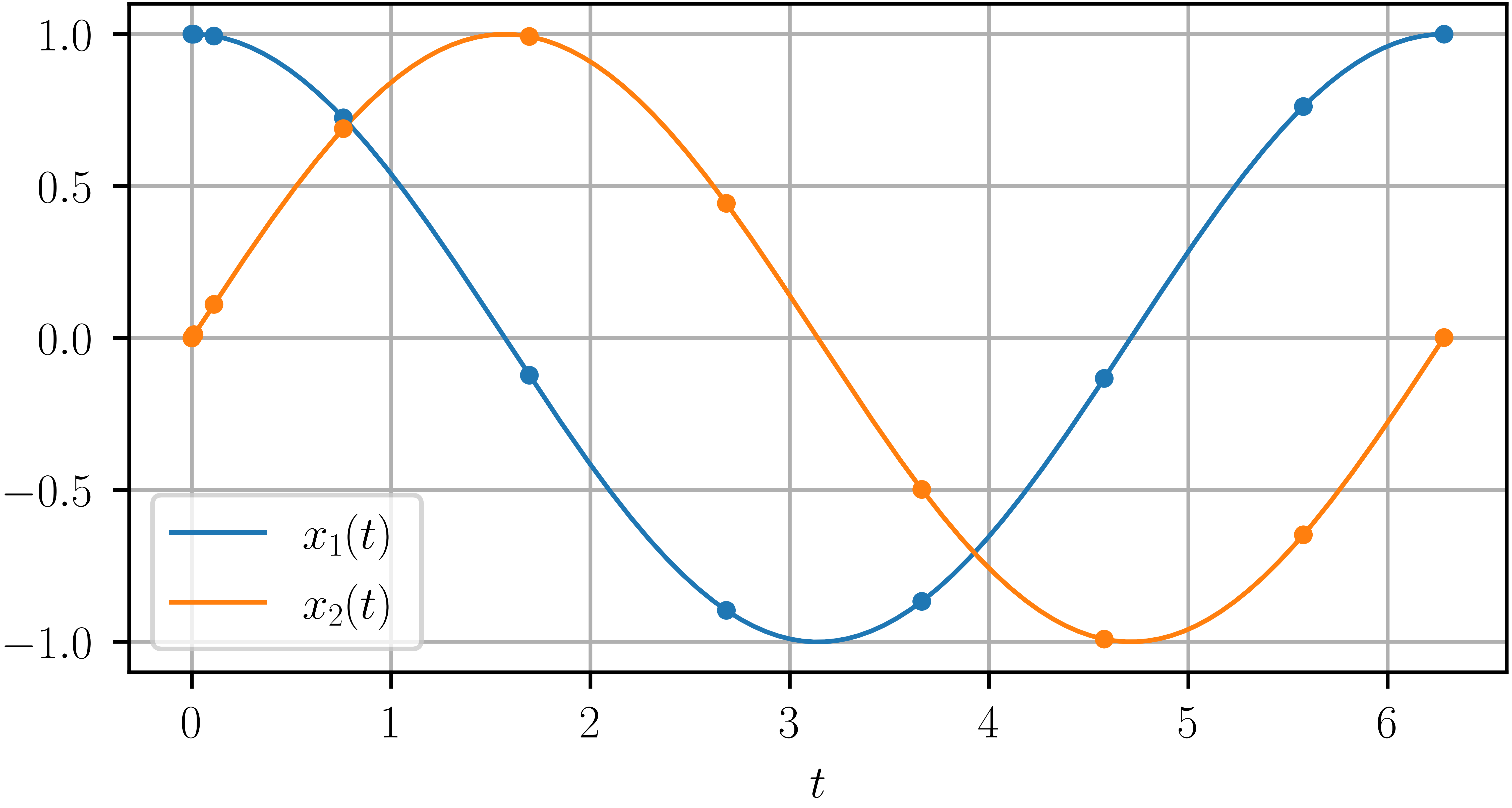
```
options = {
    "dense_output": True
}
```

🐍 SIMULATION

```
result = solve_ivp(
    fun=fun, t_span=t_span, y0=y0,
    **options
)
rt = result["t"]
x1 = result["y"][0]
x2 = result["y"][1]
sol = result["sol"]
```

📈 GRAPH

```
figure()
t = linspace(0, 2*pi, 1000)
plot(t, sol(t)[0], "-", label="$x_1(t)$")
plot(t, sol(t)[1], "-", label="$x_2(t)$")
plot(rt, x1, ".", color="C0")
plot(rt, x2, ".", color="C1")
xlabel("$t$"); grid(); legend()
```


1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6
t
x1(t)
x2(t)