MPS224 Scientific Computing

Dr. Sam Dolan

School of Mathematical and Physical Sciences,
University of Sheffield

Spring 2026

4. Solving differential equations

G18 Hicks Building
s.dolan@sheffield.ac.uk

Today's lecture

- Scientific computing packages:
  - `numpy`
  - `matplotlib`
  - `pandas`
  - `scipy`
- **Differential equations:**

  keywords: phase portraits; equilibria; limit cycles ...
- **Non-linear ODEs**:
  1. Logistic equation (1D)
  2. Predator-prey equation (2D autonomous conservative)
  3. van der Pol equation (2nd-order)

SciPy

SciPy is a collection of mathematical algorithms and functions built on the Numpy extension of Python.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import scipy as sp
```

- Tutorial:
  `http://docs.scipy.org/doc/scipy-dev/reference/tutorial/index.html`

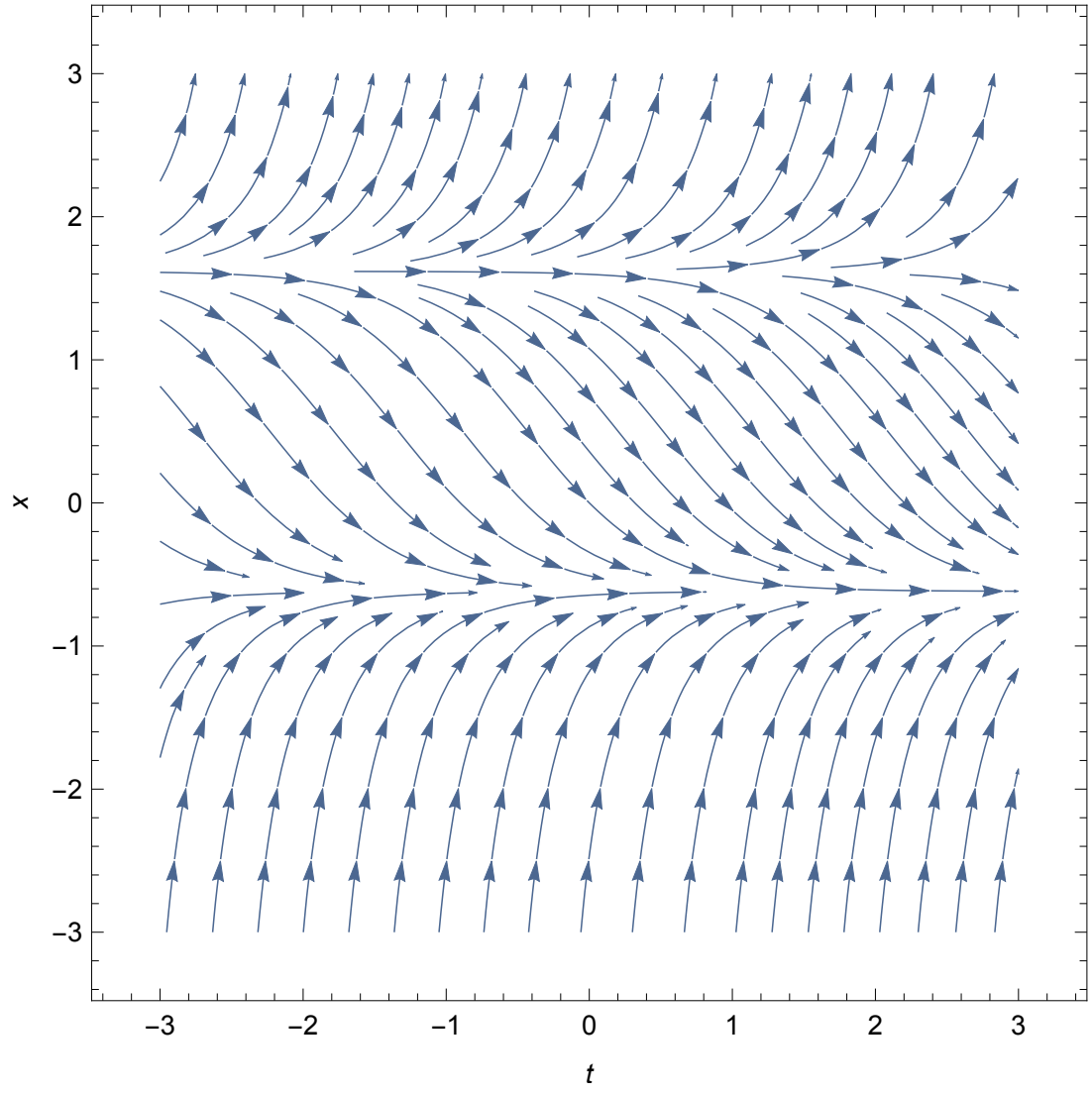
SciPy

- Various useful modules in the `scipy` package:
  - `sp.special` : special functions (Bessel, Legendre, Hypergeometric, etc).
  - `sp.integrate` : for integrating functions and sets of ODEs
  - `sp.optimize` : curve fitting, minimization, etc.
  - `sp.interpolate` : interpolation, splines, etc.
  - `sp.fftpack` : Fourier transforms.
  - `sp.linalg` : Linear algebra.
- We will solve differential equations with `scipy.integrate.solve_ivp()`
- IVP = **I**nitial **V**alue **P**roblem .

Ordinary differential equations

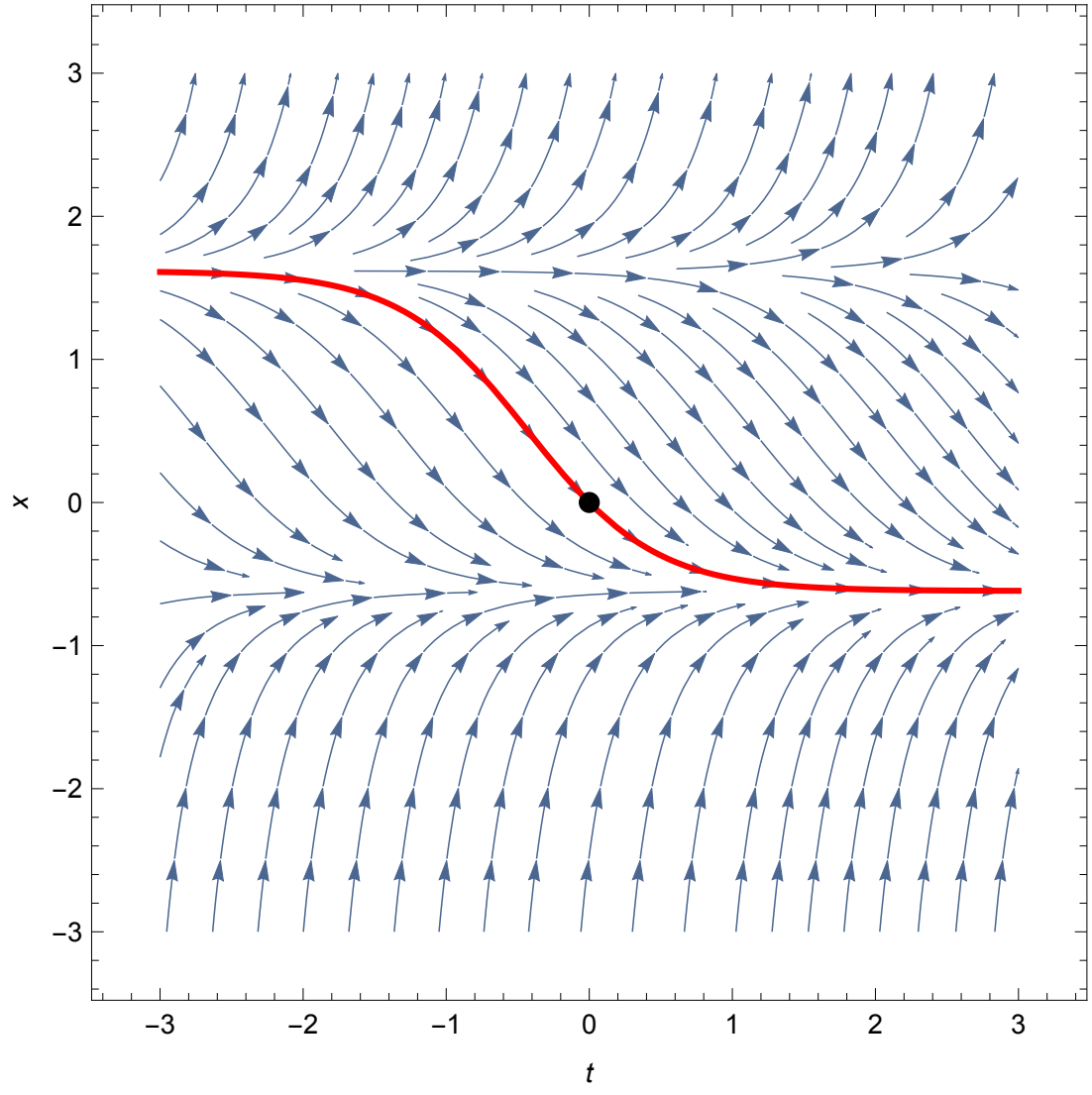
- Here is an example of an **ordinary differential equation** (ODE):
$$\frac{dx}{dt} = x^2 - x - 1.$$
- x is the dependent variable, and t is the **ind**ependent variable.
- a specific solution $x(t)$ is an **integral curve** of the ODE.
- to find an integral curve, we specify an **initial condition**, e.g.
$$x(t = 0) = 1.$$

Ordinary differential equations



- Here is the gradient field $\frac{dx}{dt}$ at each point in the flow.

Ordinary differential equations



- Here is an **integral curve** with initial condition $x(0) = 0$.

Numerical methods

- Suppose we have a first-order ODE describing an initial value problem:

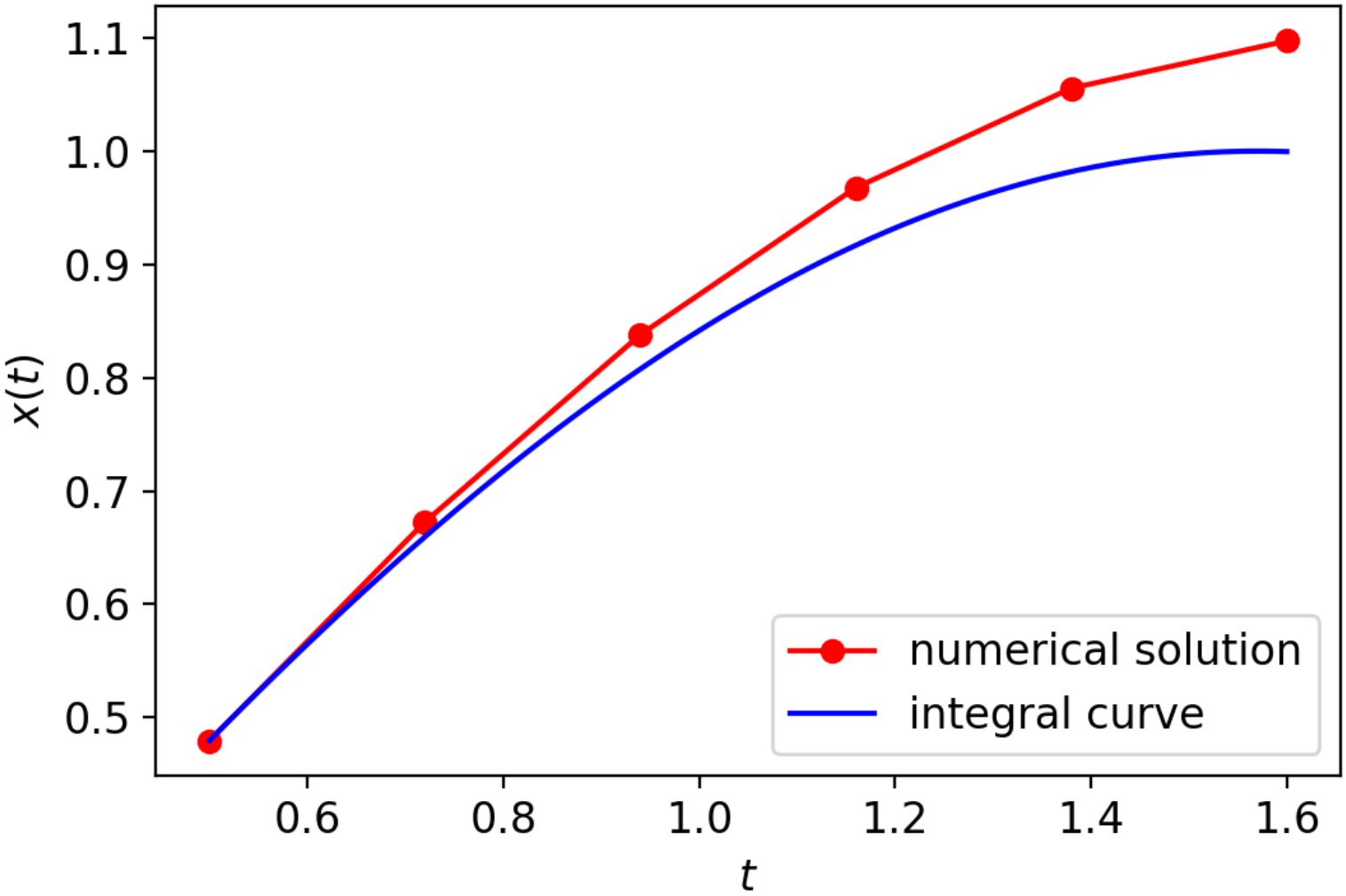
$$\frac{dx}{dt} = f(x,t), \qquad x(t_0) = x_0$$

- In principle, **how** do we find an integral curve numerically?

How do we ...

... find a sequence of numerical estimates $[x_0, x_1, x_2, \ldots x_n]$ at times $[t_0, t_1, t_2 \ldots t_n]$, where $t_0 < t_1 < t_2 < \ldots$, that are reasonable estimates of the exact sequence $[x(t_0), x(t_1), x(t_2), \ldots, x(t_n)]$ where $x(t)$ is the **exact** solution of the ODE, which may be unknown.

- We take small steps ...


1.1
1.0
0.9
0.8
0.7
0.6
0.5
x(t)
0.6
0.8
1.0
1.2
1.4
1.6
t
numerical solution
integral curve

Ordinary differential equations (ODEs)

- ODEs have just **one** independent variable (t).
- There may be several dependent variables $x_i = \{x_1(t), x_2(t), \ldots\}$,
- and a set of functions F_j relating x_i and its derivatives,

$$F_j(x_i, \dot{x}_i, \ddot{x}_i, \ldots; t) = 0,$$

where $\dot{x}_i = \frac{dx_i}{dt}$, $\ddot{x}_i = \frac{d^2x_i}{dt^2}$, etc.

Keywords

- **Order** refers to highest derivative: kth order $\Leftrightarrow \frac{d^k x}{dt^k}$
- **Dimension** refers number of dependent variables $\mathbf{x} = [x_1 \ldots x_d]$, and the number of independent equations.
- **Autonomous** $\Leftrightarrow$ F_j have no explicit dependence on t
- **Linear** if F_j has only linear dependence on x_i, $\dot{x}_i$, $\ldots$ and their combinations. Otherwise it is **non-linear**.
- **Linear** $\Rightarrow$ superposition principle $\Rightarrow$ 'Easy'.

Ordinary differential equations

$$\frac{dx}{dt} = x^2 - x - 1$$

This example is . . .

- . . . **first-order**, as dx/dt is the highest derivative.
- . . . **one-dimensional**, as x is the only dependent variable.
- . . . **autonomous**, as the rate of change dx/dt does not depend on the independent variable t.
- . . . **non-linear**, because of the non-linear term x^2 on the right-hand side.

1D autonomous equation

- Consider the 1st-order autonomous case:

$$\boxed{\frac{dx}{dt} = f(x)}$$

- A solution is typically found by **separation of variables**.
- Divide by $f(x)$ and integrate

$$\int \frac{dx}{f(x)} = t + c$$

- Some cases can be solved exactly, e.g,

$$f(x) = x \quad \Rightarrow \ln(x) = t + c \quad \Rightarrow x(t) = Ae^t.$$

- What if integral can't be found analytically?
- Integrate numerically and invert to find $x(t)$? **No.**
- Numerically solve the differential equation with `solve_ivp()`.

1D autonomous equation: example

- The **Logistic Equation** is a first-order 1D autonomous non-linear ODE:

$$\boxed{\frac{dx}{dt} = x(1-x), \quad x(0) = x_0.}$$

Example

1. Show that the logistic equation has the exact solution:

$$x(t) = \frac{1}{1 + Ae^{-t}}$$

   where $A = 1/x_0 - 1$.

2. Find a **numerical solution** for the initial value $x_0 = \frac{1}{2}$ on the domain $t \in [0, 10]$ using `solve_ivp()`.

import statement

```
>>> from scipy.integrate import solve_ivp
>>> ?solve_ivp   # Read the help file.
```

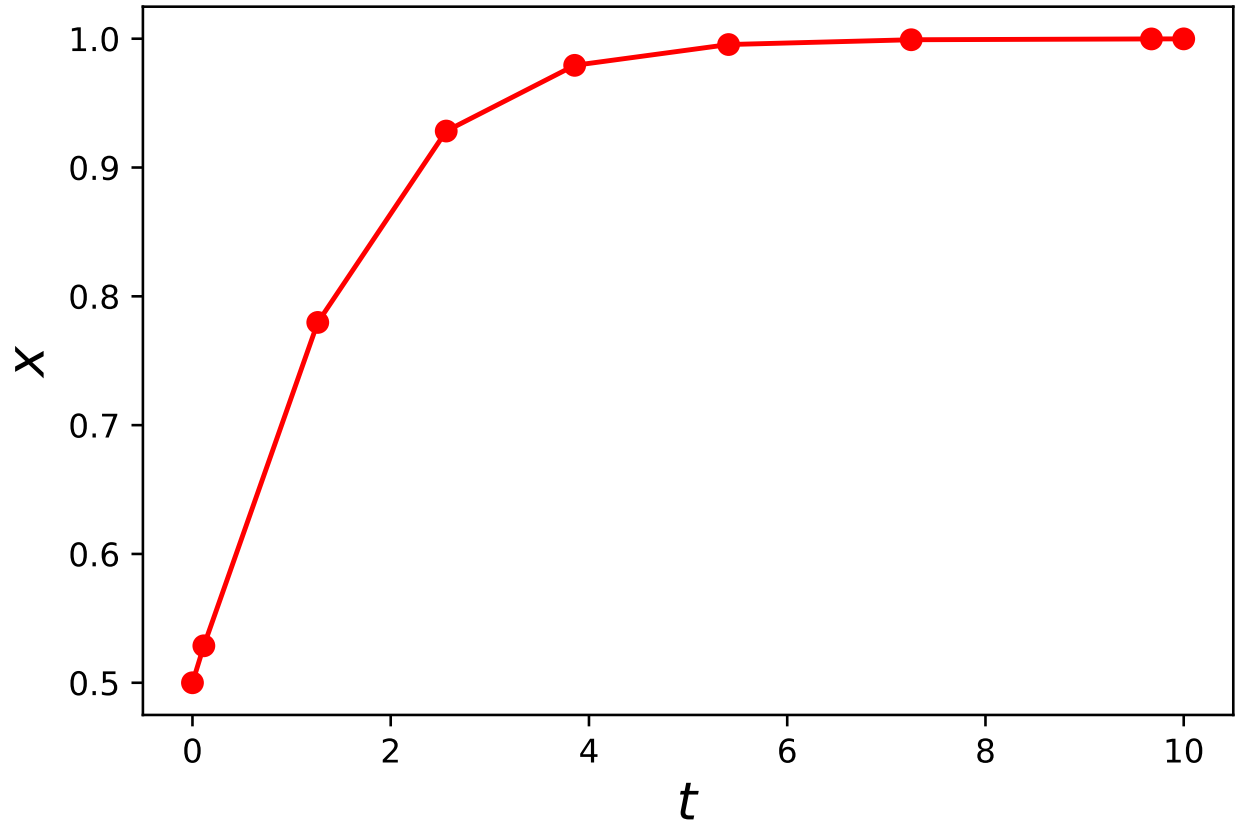
1D autonomous equation: example (version 1)

$$\frac{dx}{dt} = x(1-x), \quad x(0) = x_0$$

```
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
def logistic(t, x):
    """Returns the gradient dx/dt for the logistic equation"""
    return x*(1 - x)

x0 = 0.5   # an initial condition, x(0) = x0
sol = solve_ivp(logistic, (0, 10), [x0])
plt.xlabel('$t$', fontsize=16); plt.ylabel('$x$', fontsize=16)
plt.plot(sol.t, sol.y[0], 'r-o')
```

1D autonomous equation: result



- **Problem:** too few points in the domain.
- **Q.** How do I set the values in the domain (t)?
- **A.** Use the `t_eval` optional parameter.

1D autonomous equation: example (version 2)

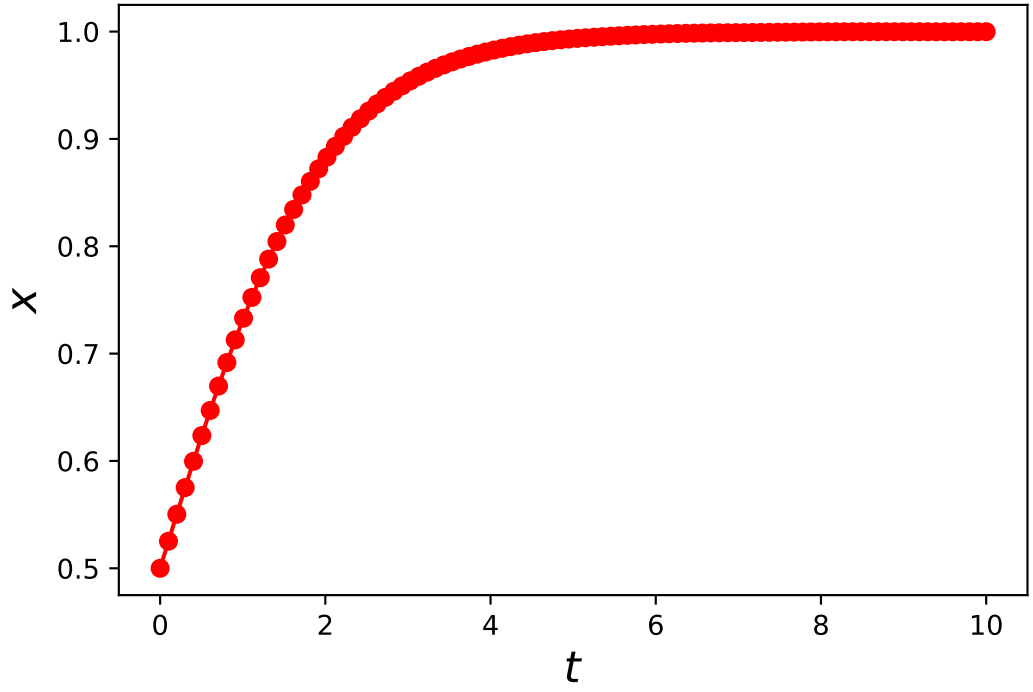
$$\frac{dx}{dt} = x(1-x), \quad x(0) = x_0$$

```
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
def logistic(t, x):
    """Returns the gradient dx/dt for the logistic equation"""
    return x*(1 - x)

ts = np.linspace(0.0, 10.0, 100) # values of independent variable
x0 = 0.5   # an initial condition, x(0) = x0
sol = solve_ivp(logistic, (ts[0], ts[-1]), [x0], t_eval=ts)
plt.xlabel('$t$', fontsize=16); plt.ylabel('$x$', fontsize=16)
plt.plot(sol.t, sol.y[0], 'r-o')
```

1D autonomous equation: example

$$\frac{dx}{dt} = x(1-x), \quad x(0) = x_0$$



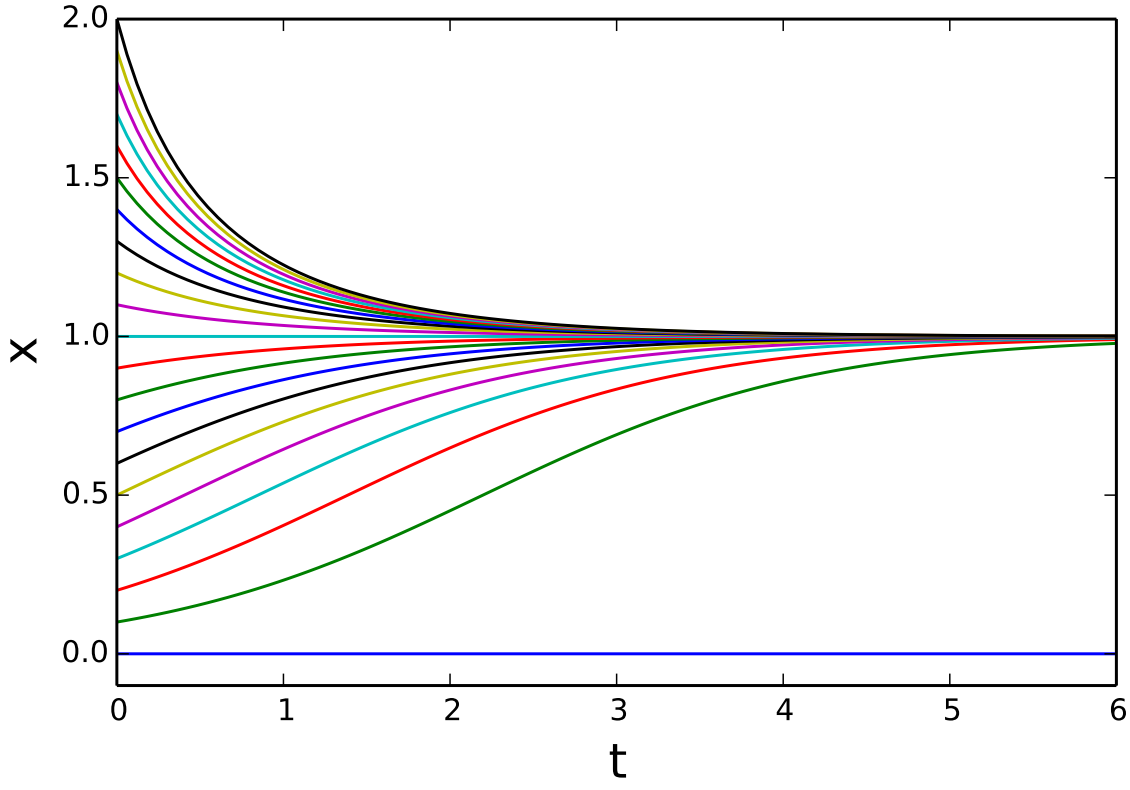
- Now let's plot curves for several initial conditions . . .

1D autonomous equation: example

$$\frac{dx}{dt} = x(1-x), \quad x(0) = x_0$$

```
# Plot curves for several initial conditions
ics = np.linspace(0.0, 2.0, 21)  # a list of initial conditions
for x0 in ics:
    sol = solve_ivp(logistic, (ts[0], ts[-1]), [x0], t_eval=ts)
    plt.plot(sol.t, sol.y[0])
```

1D autonomous equation: example



- Two equilibrium positions: $x = 0$ and $x = 1$.
- $x = 0$ is an **unstable** equilibrium.
- $x = 1$ is a **stable** equilibrium.

2D autonomous equations

- Now consider a first order system with two dependent variables, x and y,

$$\frac{dx}{dt} = f(x, y; t),$$
$$\frac{dy}{dt} = g(x, y; t).$$

- System is **autonomous** iff f and g do not depend on t.
- **Example:** Modelling the populations of rabbits and foxes.

2D autonomous equations: example

Predator-prey equations

Also known as *Lotka-Volterra equations*, the predator-prey equations are a pair of coupled first-order non-linear ordinary differential equations.

They represent a simplified model of the change in populations of two species which interact via predation. For example, foxes (predators) and rabbits (prey). Let x and y represent rabbit and fox populations, respectively. Then

$$\frac{dx}{dt} = ax - bxy$$
$$\frac{dy}{dt} = -cy + dxy$$

Here a, b, c and d are parameters, which are assumed to be positive.

- Define a vector of dependent variables: $\mathbf{Z}(t) = [x(t), y(t)]$.
- In Python this vector can be a list or an array.

Predator-prey equations

$$\frac{dx}{dt} = ax - bxy$$
$$\frac{dy}{dt} = -cy + dxy$$

```
def dZdt(t, Z, a=1, b=1, c=1, d=1): # a,b,c,d optional arguments.
    x, y = Z[0], Z[1]
    dxdt, dydt = x*(a - b*y), -y*(c - d*x)
    return [dxdt, dydt]


ts = np.linspace(0, 12, 100)
Z0 = [1.5, 1.0]   # initial conditions for x and y
sol = solve_ivp(dZdt, (ts[0], ts[-1]), Z0, t_eval=ts, args=(1,1,1,1))
  # use optional argument 'args' to pass parameters to dZ_dt
prey = sol.y[0] # first column
predators = sol.y[1]  # second column
```

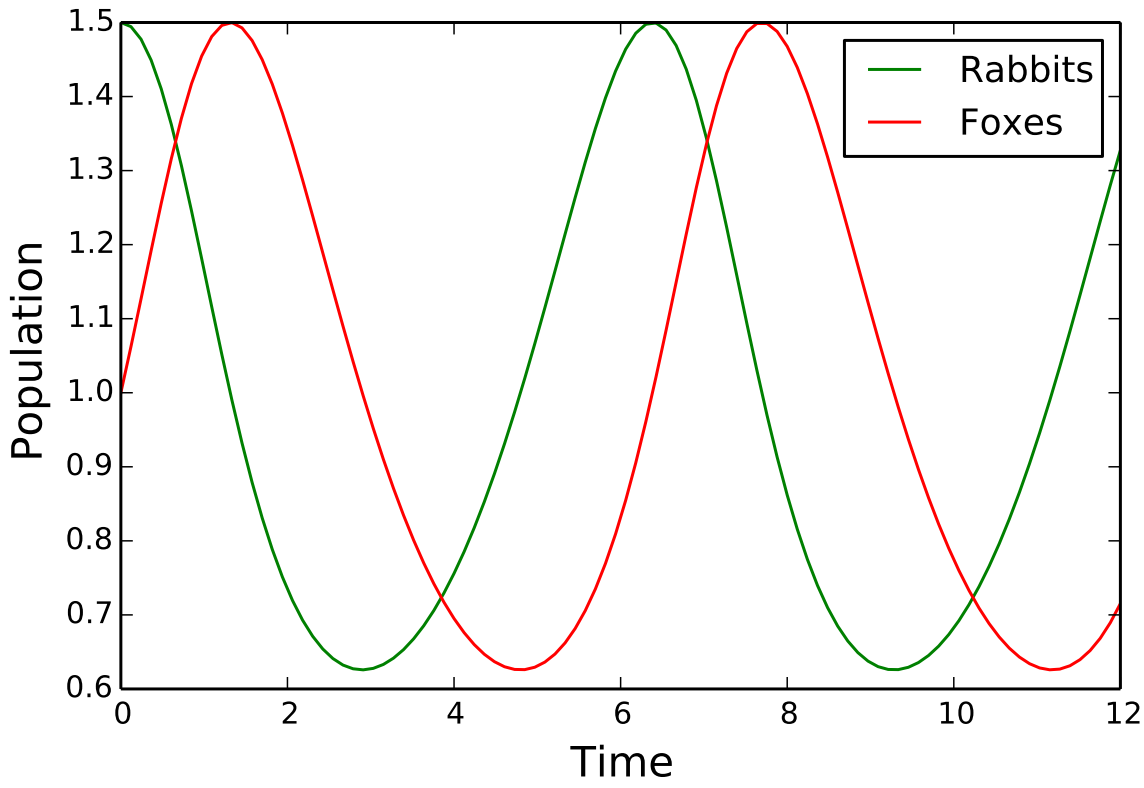
Predator-prey equations

$$\frac{dx}{dt} = ax - bxy$$
$$\frac{dy}{dt} = -cy + dxy$$

```
# Let's plot 'rabbit' and 'fox' populations as a function of time
plt.plot(ts, prey, "+", label="Rabbits")
plt.plot(ts, predators, "x", label="Foxes")
plt.xlabel("Time", fontsize=14)
plt.ylabel("Population", fontsize=14)
plt.legend();
```

Predator-prey equations

$$\frac{dx}{dt} = ax - bxy$$
$$\frac{dy}{dt} = -cy + dxy$$

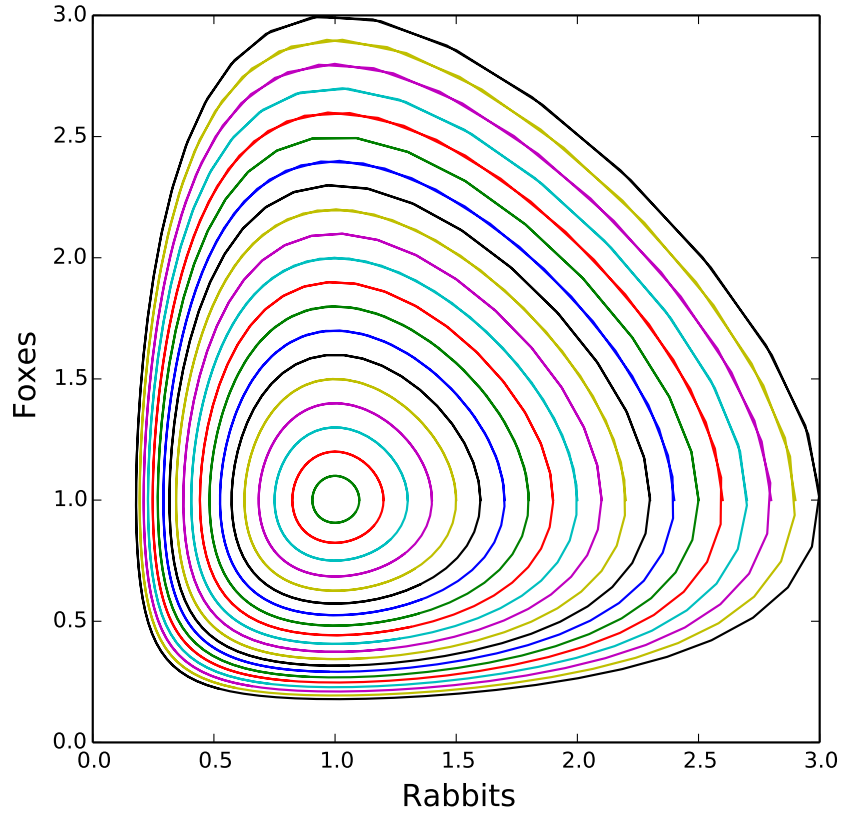


Predator-prey equations: Phase plot

- The ODEs are **autonomous**: no explicit dependence on t
- **Phase portrait**: Plot x vs y (instead of x, y vs t).
- One curve for each initial condition
- Curves will not cross (typically) for an autonomous system.

```
fig = plt.figure()
fig.set_size_inches(6,6)  # Square plot, 1:1 aspect ratio
ics = np.arange(1.0, 3.0, 0.1) # initial conditions
for r in ics:
    Z0 = [r, 1.0]
    sol = solve_ivp(dZ_dt, (ts[0], ts[1]), Z0, t_eval=ts)
    plt.plot(sol.y[0], sol.y[1], "-")
plt.xlabel("Rabbits", fontsize=14)
plt.ylabel("Foxes", fontsize=14)
```

Predator-prey equations: Phase plot



- Curves do not cross
- Closed curves $\Leftrightarrow$ **Periodic** solutions
- Equilibrium where $\dot{x} = \dot{y} = 0$, at $x = y = 1$.

The Van der Pol oscillator

The (undriven) Van der Pol oscillator is a non-conservative oscillator with non-linear damping, satisfying

$$\ddot{x} - a(1 - x^2)\dot{x} + x = 0$$

- This is a **second-order** ODE with one parameter, a.
- $|x| > 1$: system loses energy
- $|x| < 1$: system absorbs energy

- Originally, used as a model for an electric circuit with a vacuum tube.
- Used to model biological processes such as heart beat, circadian rhythms, biochemical oscillators, and pacemaker neurons.

Van der Pol oscillator

$$\ddot{x} - a(1 - x^2)\dot{x} + x = 0$$

First-order reduction:

Any second-order equation can be written as two coupled first-order equations, by introducing a new variable.

- Let $y = \dot{x}$. Then

$$\begin{aligned} \dot{x} &= y \\ \dot{y} &= a(1 - x^2)y - x \end{aligned}$$

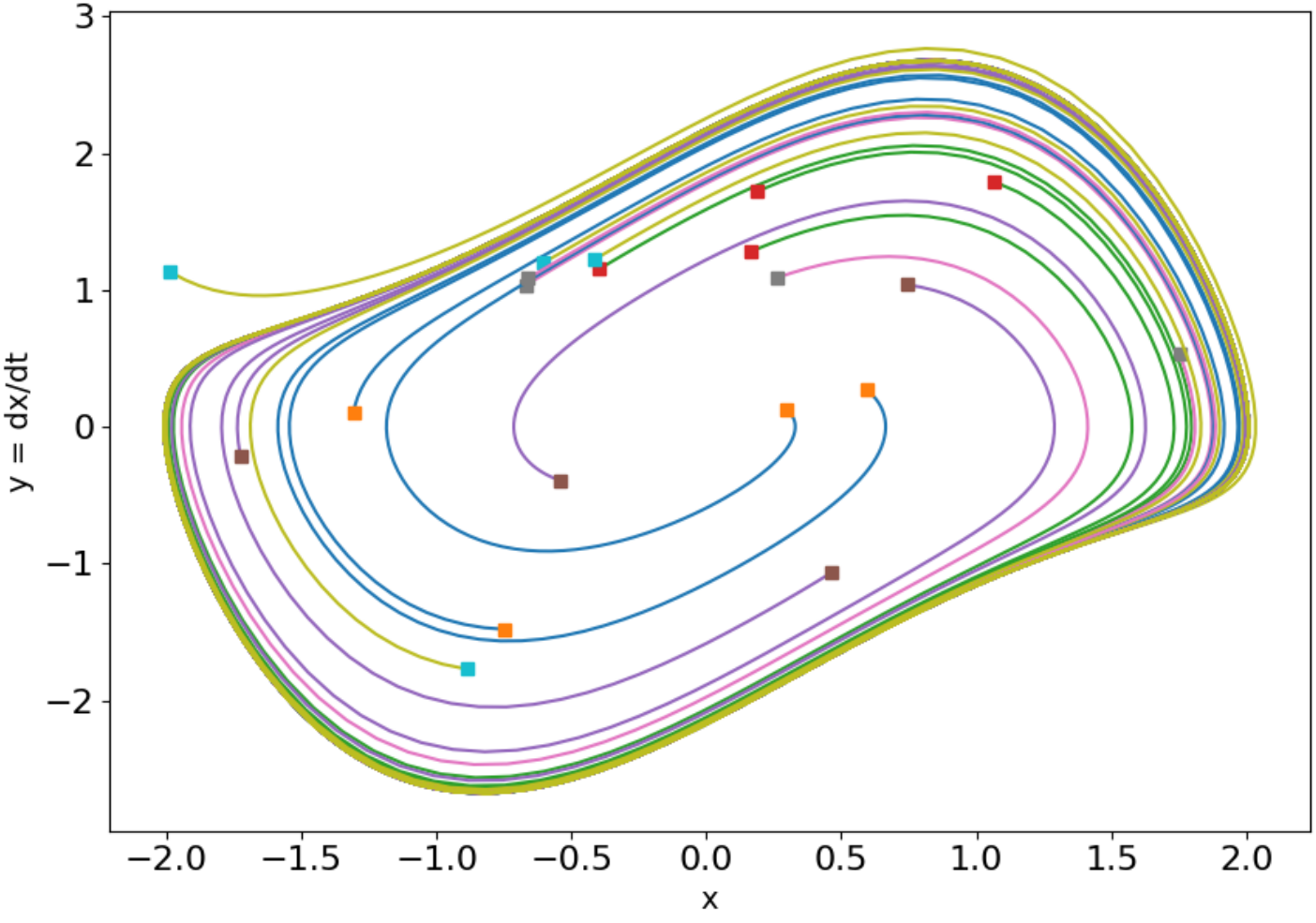
- Not unique: we could make another choice, such as $z = \dot{x} + x$.

$$\begin{aligned} \dot{x} &= y \\ \dot{y} &= a(1-x^2)y - x \end{aligned}$$

```
def dZdt(t, Z, a = 1.0):
    x, y = Z[0], Z[1]
    return [y, a*(1-x**2)*y - x]

def random_ic(scalefac=2.0):  # stochastic initial condition
    return scalefac*(2.0*np.random.rand(2) - 1.0)

ts = np.linspace(0.0, 40.0, 400)
ics = [random_ic() for i in range(20)]
for ic in ics:
    sol = solve_ivp(dZdt, (ts[0],ts[-1]), ic, t_eval=ts)
    plt.plot(sol.y[0], sol.y[1])
    plt.plot([sol.y[0][0]],[sol.y[1][0]], 's') # plot the first point
```



- All these curves tend towards a **limit cycle**

Van der Pol oscillator: Limit cycles

- Investigate how the limit cycle varies with the parameter *a*:
- Pass *a* as an optional argument to function `dZ_dt` using the `args` optional argument of `solve_ivp()`.

```
avals = np.arange(0.2, 2.0, 0.2)  # parameters
minpt = int(len(ts) / 2)  # look at the late-time behaviour
for a in avals:
    sol = solve_ivp(dZ_dt, (ts[0], ts[-1]), random_ic(),
                    t_eval=ts, args=(a,))
    # pass optional arguments to dZ_dt
    plt.plot(sol.y[0][minpt:], sol.y[1][minpt:])
```

Van der Pol oscillator: Limit cycles