diffeqzoo

Nicholas Krämer

Dec 28, 2023

GETTING STARTED

So, what was the initial condition of the restricted three-body problem again?

`diffeqzoo` delivers all differential equation test problems in one place. It works with numpy and jax.

CHAPTER

ONE

INSTALLATION

Get the most recent stable version from PyPi:

```
pip install diffeqzoo
```

Or directly from GitHub:

```
pip install git+https://github.com/pnkraemer/diffeqzoo.git
```

These commands assume that NumPy or JAX are installed separately by the user. Read more about installing this package here.

CHAPTER

TWO

FEATURES INCLUDE

- Oscillating systems (Lotka-Volterra, Fitzhugh-Nagumo, Van-der-Pol, ...)
- Chaotic systems (Lorenz63, Lorenz96, Roessler, ...)
- Epidemiological models (SIR, SEIR, SIRD, ...)
- N-Body problems and celestial mechanics (Rigid-body, restricted-three-body, Pleiades, Henon-Heiles, ...)
- Chemical reactions (HIRES, ROBER, ...)
- Boundary value problems

2.1 As well as

- Flexibly NumPy and JAX-backends. Other than one of those two, there are 0 (zero!) dependencies.
- Mathematical descriptions **and BibTex entries** of the ODE problems
- Compatibility with all NumPy/JAX-based ODE solvers: SciPy, JAX, Diffrax, ProbNum, Tornadox, etc..

and many more goodies.

- **DOCUMENTATION:** documentation
- **ISSUE TRACKER:** issue tracker

CHAPTER

THREE

QUICK EXAMPLE

```
>>> from diffeqzoo import ivps, backend
>>> backend.select("numpy")
>>>
>>> # Create test problems like this
>>> f, u0, t_span, f_args = ivps.lotka_volterra()
>>> x = f(u0, *f_args)
>>> print(x)
[-10.  10.]
>>>
>>> # The numpy backend determines the type of input/output
>>> print(type(x))
<class 'numpy.ndarray'>
>>>
>>> # All sorts of ODEs are available, e.g., Rigid-Body:
>>> f, u0, t_span, f_args = ivps.rigid_body()
>>> print(f(u0, *f_args))
[-0.     1.125 -0.   ]
>>>
>>> ## make it jax
>>> backend.change_to("jax")
>>> f, u0, t_span, f_args = ivps.rigid_body()
>>> x = f(u0, *f_args)
>>> print(x)
[-0.     1.125 -0.   ]
>>> print(type(x))
<class 'jaxlib.xla_extension.ArrayImpl'>
```

CHAPTER

FOUR

SIMILAR PROJECTS

- F. Mazzia et al. published a test set for IVP solvers for Matlab and Fortran. There is a similar test set for BVP solvers. Neither one offers Python code, and both also run benchmarks, which `diffeqzoo` does not care about at all.
- E. Hairer et al. published their stiff ODE test set, but there is no Python code
- NonlinearBenchmark hosts datasets of nonlinear dynamical system observations. They are quite specialised problems, and don't contain the textbook problems like Lotka-Volterra, van der Pol, etc..
- DifferentialEquations.jl provides example ODE problems in Julia.
- ProbNum's problem zoo offers a similar set of problems to `diffeqzoo` (no surprise – the set of authors intersects) but tied to ProbNum's ODE solver interface. `diffeqzoo` is less of an API, switches more flexibly between numpy and jax (at the time of developing), and contains more problems.
- W. Gilpin published a benchmark for forecasting and data-driven modeling, which comes with a large number of (mostly chaotic) dynamical systems.
- J. Meier lists a number of ODE attractors on his website.
- GeometricProblems.jl curates a similar list of example problems with interesting geometric structure, in Julia (link)

Anything missing in this list? Please open an issue or make a pull request.

CHAPTER

FIVE

CONTENT

5.1 Installation

Get the most recent stable version from PyPi:

```
pip install diffeqzoo
```

Or directly from GitHub:

```
pip install git+https://github.com/pnkraemer/diffeqzoo.git
```

This assumes that you have NumPy or JAX installed. It is best you do this yourself (especially for JAX), but you can also install them with `diffeqzoo` via *either* of the following:

```
pip install diffeqzoo[jax]
pip install diffeqzoo[numpy]
pip install diffeqzoo[jax,numpy]
```

which installs the CPU version of JAX. For the GPU version, install JAX yourself.

5.1.1 Local installation

In a fork, you can install the project locally: go to the project's root and run:

```
pip install .
```

or install in editable mode:

```
pip install -e .
```

5.1.2 Optional dependencies

`diffeqzoo` comes with optional dependencies. For example, NumPy or JAX (see above), but also `dev`, `doc`, `test`, `lint`, `example` groups, which are useful for the continuous integration. See the explanation of the CI for more info.

5.2 Quick example

To get started, import the `ivps` and `bvps` from `diffeqzoo`. You must also import the backend, because `diffeqzoo` needs to now whether to build the ODEs in numpy or in jax.

```
>>> from diffeqzoo import ivps, bvps, backend
>>> backend.select("numpy")
```

Here, we chose the numpy backend. We could have also used `backend.select("jax")`. Now with this backend, numpy-implementations are available via `backend.numpy`. (Under the hood, this imports and exposes either `jax.numpy` or `numpy`).

Use this backend to create ODE problems.

```
>>> f_lv, u0_lv, _, f_args_lv = ivps.lotka_volterra()
>>> x_lv = f_lv(u0_lv, *f_args_lv)
>>> print(x_lv)
[-10.  10.]
```

```
>>> f_rb, u0_rb, _, f_args_rb = ivps.rigid_body()
>>> x_rb = f_rb(u0_rb, *f_args_rb)
>>> print(x_rb)
[-0.     1.125 -0.   ]
```

```
>>> f_sir, u0_sir, _, f_args_sir = ivps.sir()
>>> x_sir = f_rb(u0_sir, *f_args_sir)
>>> print(backend.numpy.round(x_sir, 1))
[3.00000e-01 9.98000e+01 9.96004e+05]
```

While all IVP problem creators have a *similar* API, the ODE functions are not necessarily unified.

```
>>> import inspect
>>> f1, *_ = ivps.three_body_restricted()
>>>
>>> f2, *_ = ivps.sird()
>>>
>>> print(inspect.signature(f1))
(Y, dY, /, standardised_moon_mass)
>>>
>>> print(inspect.signature(f2))
(u, /, beta, gamma, eta, population_count)
```

Here, one IVP is a first-order problem with multiple parameters, the other one is a second-order problem with a single parameter. Second-order ODEs are taken seriously in the `diffeqzoo`, because the second-order form can be solved faster.

5.3 Behind the scenes

The two main assumptions behind the design of the `diffeqzoo` are:

1. A `diffeqzoo` dependency will be the least crucial dependency of any project, and the first one to be dropped; because:
2. Almost all users will use only a single function from `diffeqzoo` at a time. If necessary, this function can be copied/pasted from somewhere into each project.

What does this imply? It implies that the `diffeqzoo`'s API should be instantaneous to learn, easy to copy/paste out of (embrace the drop-ability), and extremely easy to maintain.

This is achieved by a pure-function API with minimal dependencies (either numpy or jax), and no custom data structures: everything is a plain callable, tuple, array, list, or dict. The source should be understandable by anyone that has spent time with Sections 1-5 in this tutorial. (If not, let us know!)

There are almost no nested function-calls, and the only non-trivial implementation is that of the flexible backend (which is, itself, much more minimal than comparable implementations). If copying ODE examples from this project into your own project is the best thing for you, please do so! We tried to make copy/pasting ODE code as easy as possible.

5.4 Solve IVPs with SciPy

SciPy is everyone's first stop for scientific computing in Python.

The initial value problems provided by `diffeqzoo` can be plugged into scipy's ordinary differential equation (ODE) solvers.

SciPy has two IVP solvers: `odeint` (wraps FORTRAN's `odepack`) and `solve_ivp` (native Python). They require slightly different inputs: for example, `odeint` expects vector fields `f(y, t)` and `solve_ivp` expects vector fields `f(t, y)`.

`diffeqzoo` can be used for both.

```
import inspect  # to inspect function signatures

import matplotlib.pyplot as plt
import scipy.integrate

from diffeqzoo import backend, ivps

backend.select("numpy")
```

5.4.1 Using solve_ivp

Let's start with `solve_ivp`, because that is the recommendation given in the docs of `scipy.integrate.odeint` (link).

```
print(inspect.signature(scipy.integrate.solve_ivp))
```

Here is how we solve ODEs from `diffeqzoo` with `solve_ivp`. Most ODE test problems are autonomous, which means that the vector fields do not depend on time. We can wrap them into a non-autonomous format and plug them into scipy.

```
f, y0, t_span, args = ivps.lotka_volterra()
print(inspect.signature(f), args)


def fun(t, y, *args):
    return f(y, *args)


scipy.integrate.solve_ivp(fun=fun, t_span=t_span, y0=y0, args=args)
```

Let's plot the solution.

```
t_eval = backend.numpy.linspace(*t_span, num=200)
sol = scipy.integrate.solve_ivp(fun=fun, t_span=t_span, t_eval=t_eval, y0=y0, args=args)
y_eval = sol.y.T
```

```
plt.plot(t_eval, y_eval)
plt.show()
```

5.4.2 Using odeint

The usage of `odeint` is very similar to that of `solve_ivp`. We simply need to rename a few arguments and wrap the vector field slightly differently.

```
print(inspect.signature(scipy.integrate.odeint))
```

Let's compute the ODE solution with `odeint` and plot the solution.

```
f, y0, t_span, args = ivps.pleiades_first_order()
print(inspect.signature(f), args)


def func(y, t, *args):
    return f(y, *args)


t = backend.numpy.linspace(*t_span, num=300)
y = scipy.integrate.odeint(func=func, y0=y0, t=t, args=args)
colors = ["C" + str(i) for i in range(7)]
for x1, x2, color in zip(y[:, 0:7].T, y[:, 7:14].T, colors):
    plt.plot(x1, x2, color=color)
    plt.plot(x1[0], x2[0], marker=".", color=color)
    plt.plot(x1[-1], x2[-1], marker="*", markersize=10, color=color)
plt.show()
```

5.5 Solve IVPs with Diffrax

Diffrax provides numerical differential equation solvers in JAX. Its advantages over, e.g., JAX' solvers include a larger set of available solvers. We can plug the `diffeqzoo`'s problems into diffrax as follows.

```
import inspect

import diffrax
import jax
import matplotlib.pyplot as plt

from diffeqzoo import backend, ivps

backend.select("jax")
```

```
print(inspect.signature(diffrax.diffeqsolve))
```

Most ODEs are autonomous (i.e., they do not depend on the time variable), and the `diffeqzoo` implements them as such. Just like most other ODE solvers in Python, Diffrax expects non-autonomous vector fields. It further requires wrapping vector fields into `diffrax.ODETerm` objects, which can be achieved easily.

Let's plot the solution of an example initial value problem.

```
f, y0, (t0, t1), args = ivps.seir()


@jax.jit
def vf(t, y, p):
    return f(y, *p)


term = diffrax.ODETerm(vf)
solver = diffrax.Dopri5()

ts = backend.numpy.linspace(t0, t1, num=200)
saveat = diffrax.SaveAt(ts=ts)

sol = diffrax.diffeqsolve(
    term,
    solver,
    t0=t0,
    t1=t1,
    dt0=0.1,
    y0=y0,
    args=args,
    saveat=saveat,
)

plt.plot(sol.ts, sol.ys)
plt.show()
```

5.6 Solve IVPs with JAX

JAX provides not only a linear algebra backend, automatic differentiation, and other useful function transformations, but also an initial value problem solver: `jax.experimental.ode.odeint()`. Its API mirrors the API of `scipy.integrate.odeint` (which we cover in a different tutorial).

With the JAX backend, we can plug `diffeqzoo`'s initial value problems into this API as follows.

```
import inspect

import jax
import jax.experimental.ode
import matplotlib.pyplot as plt

from diffeqzoo import backend, ivps

backend.select("jax")
```

```
print(inspect.signature(jax.experimental.ode.odeint))
```

Most ODEs are autonomous (which means that the vector field does not depend on the time variable), but just like most other ODE solvers, JAX' `odeint` expects a time-dependent vector field. We can wrap the output of the `diffeqzoo` into the desired format easily. Let's compute the solution of an example problem and plot the solution.

```
f, y0, tspan, f_args = ivps.fitzhugh_nagumo()


@jax.jit
def fun(y, _, *args):
    return f(y, *args)


t = backend.numpy.linspace(*tspan, num=200)
y = jax.experimental.ode.odeint(fun, y0, t, *f_args)

plt.plot(t, y)
plt.show()
```

```
(f, y0, tspan, f_args), info = ivps.heat_1d_dirichlet(num_gridpoints=100)
grid = info["grid"]


@jax.jit
def fun(y, _, *args):
    return f(y, *args)


t = backend.numpy.linspace(*tspan, num=200)
y = jax.experimental.ode.odeint(fun, y0, t, *f_args)

for i, ys in enumerate(y):
    # Reduce the opacity over time
    alpha = 3.0 * float(backend.numpy.mean(ys))
```

(continues on next page)

(continued from previous page)

```
    plt.plot(grid, ys, alpha=alpha, color="C0")
plt.show()
```

5.7 Solve BVPs with SciPy

There are not many boundary value problem solvers in Python, but SciPy offers one.

The boundary value problems in the `diffeqzoo` can be plugged into SciPy's solver.

```
import inspect

import matplotlib.pyplot as plt
import scipy.integrate

from diffeqzoo import backend, bvps

backend.select("numpy")
```

There are many kinds of boundary conditions. One common distinction is between a two-point boundary condition $g_0(y(0)) = g_1(y(1)) = 0$ and a (general) boundary condition $g(y(0), y(1)) = 0$. This tutorial covers both.

5.7.1 General boundary conditions

We start with the more general case, because it is what SciPy's solver expects.

```
print(inspect.signature(scipy.integrate.solve_bvp))
```

Let's compute the solution of an example boundary value problem and plot the solution.

```
f, bc, tspan, f_args = bvps.measles()

print(inspect.signature(f), inspect.signature(bc))


def fun(t, y):
    return f(t, y, *f_args)


x = backend.numpy.linspace(*tspan, 50)
y = backend.numpy.ones((3, x.shape[0]))
sol = scipy.integrate.solve_bvp(fun=fun, bc=bc, x=x, y=y)


# Plotting grids
xs = backend.numpy.linspace(*tspan)
ys = sol.sol(xs).T


plt.plot(xs, ys)
plt.show()
```

5.7.2 Two-point boundary conditions

Some boundary value problems have a boundary condition of the form $g_0(y(0)) = g_1(y(1)) = 0$, and the separability of the boundary conditions can often be exploited for faster solvers.

But if a solver expects non-separable conditions (like SciPy's solver), we can wrap them easily. Note how many boundary value problems are second-order differential equations, and solvers like SciPy's often expect a first-order form.

```
f, (g0, g1), tspan, f_args = bvps.bratu()

# split & concatenate to second-order ODEs as first-order ODEs


def fun(t, y):
    y, dy = backend.numpy.split(y, 2)
    ddy = f(y, *f_args)
    return backend.numpy.concatenate((dy, ddy))


def bc(y0, y1):
    y0, _ = backend.numpy.split(y0, 2)
    y1, _ = backend.numpy.split(y1, 2)
    return backend.numpy.concatenate((g0(y0), g1(y1)))


# Initial guess
x = backend.numpy.linspace(*tspan, 15)
y = backend.numpy.zeros((2, x.shape[0]))
y[0] += 3

sol = scipy.integrate.solve_bvp(fun=fun, bc=bc, x=x, y=y)

# Plotting grid
xs = backend.numpy.linspace(*tspan)
ys = sol.sol(xs).T

plt.plot(xs, ys)
plt.show()
```

5.8 diffeqzoo Package

Differential equation problem zoo.

5.8.1 Variables

backend	Backend implementation of NumPy-like functions via either numpy or jax.

backend

diffeqzoo.**backend = <diffeqzoo.backend object>**

Backend implementation of NumPy-like functions via either numpy or jax.

```
>>> from diffeqzoo import backend  # one-stop-shop for numpy, scipy, etc.
```

```
>>> # backend.numpy
>>> backend.select("jax")
>>> backend.numpy.asarray(2.)
DeviceArray(2., dtype=float32, weak_type=True)
```

```
>>> # Change backend; backend.select("numpy") is not allowed anymore!
>>> backend.change_to("numpy")
>>> backend.numpy.asarray(2.)
array(2.)
```

To learn about its properties and methods, run `help(diffeqzoo.backend)`.

Note: If you want to change the backend, use `backend.change_to(choice)()` instead of `backend.select()`. The reason for this distinction is that we want to reduce the number of accidental backend-changes. Ideally, you select the backend once, and only once.

---

diffeqzoo.bvps Module

Boundary value problem examples.

This module provides a number of example implementations of boundary value problems (BVPs). BVPs are a combination of a (commonly second-order) ordinary differential equation

$$\ddot{u}(t) = f(u(t), \dot{u}(t), t, \theta)$$

and a set of boundary conditions: often, it is either a two-point boundary condition, $g_0(u(0)) = g_1(u(1)) = 0$, or a general boundary condition, $g(u(0), u(1)) = 0$. The boundary conditions g_0, g_1, or g, and the vector field f are known, the parameters θ might be known, and u is unknown.

The functions in this module construct implementations of this kind of problem. They (loosely) follow the input/output rule

```
f, g, (t0, tmax), param = constructor()
f, (g0, g1), (t0, tmax), param = constructor_two_point()
```

where the constructor is, e.g., `pendulum()` or `measles()`. This API specification is only loose, because every problem is different. For example, first-order problems implement a differential equation

$$\dot{u}(t) = f(u(t), t, \theta).$$

We try to stick as closely as possible to the above signature, but if problem-specific issues arise, we allow ourselves to deviate from this specification. When in doubt, consult the documentation of the respect constructor function. Each function's documentation also explains whether the problem is a two-point boundary value problem, and whether the differential equation is first-, second-, or higher order.

Functions

bratu(*[, time_span, parameters])	Construct Bratu's problem.
bratu_with_unused_derivative_argument(*[, ...])	Construct Bratu's problem with a signature $(u, \dot{u})$ / and an unused second argument.
measles(*[, time_span, mu, lmbda, eta, beta0])	Construct the Measles problem.
pendulum(*[, time_span, parameters])	Construct the pendulum problem.
pendulum_with_unused_derivative_argument(*)	Construct the pendulum problem with a signature $(u, \dot{u})$ / and an unused second argument.

bratu

diffeqzoo.bvps.**bratu**(*, *time_span=(0.0, 1.0)*, *parameters=(1.0,)*)

Construct Bratu's problem.

Bratu's problem consists of a second-order differential equation and two-point boundary conditions. It is a common example problem to showcase BVP solvers and due to Bratu (1913).

```
@article{bratu1913equations,
    title={Sur les {\'e}quations int{\'e}grales non lin{\'e}aires},
    author={Bratu, G},
    journal={Bulletin de la Soci{\'e}t{\'e} Math{\'e}matique de France},
    volume={41},
    pages={346--350},
    year={1913}
}
```

bratu_with_unused_derivative_argument

diffeqzoo.bvps.**bratu_with_unused_derivative_argument**(*, *time_span=(0.0, 1.0)*, *parameters=(1.0,)*)

Construct Bratu's problem with a signature $(u, \dot{u})$ / and an unused second argument.

See *bratu()* for a more detailed problem description.

measles

diffeqzoo.bvps.**measles**(*, *time_span=(0.0, 1.0)*, *mu=0.02*, *lmbda=0.0279*, *eta=0.01*, *beta0=1575*)

Construct the Measles problem.

The Measles problem is a first-order differential equation with general boundary conditions (specifically: periodic boundary conditions). It describes the dynamics of a seasonal disease, and is a standard BVP testproblem.

Ascher et al. (1995) point to Schwartz (1983) as the original source.

```
@article{schwartz1983estimating,
    title={Estimating regions of existence of unstable periodic orbits using␣
↪computer-based techniques},
    author={Schwartz, Ira Bruce},
    journal={SIAM Journal on Numerical Analysis},
    volume={20},
    number={1},
    pages={106--120},
    year={1983},
    publisher={SIAM}
}
```

```
@book{ascher1995numerical,
    title={Numerical Solution of Boundary Value Problems for Ordinary Differential␣
↪Equations},
    author={Ascher, Uri M and Mattheij, Robert MM and Russell, Robert D},
    year={1995},
    publisher={SIAM}
}
```

pendulum

diffeqzoo.bvps.**pendulum**(*, *time_span=(0.0, 1.5707963267948966)*, *parameters=(9.81,)*)

Construct the pendulum problem.

The pendulum problem consists of a second-order differential equation and two-point boundary conditions. It is a common example BVP to showcase a boundary value problem solver.

Note: Help wanted!

If you know which paper/book to cite when the pendulum problem is used in a paper, please consider making a contribution.

---

pendulum_with_unused_derivative_argument

diffeqzoo.bvps.**pendulum_with_unused_derivative_argument**(*, *time_span=(0.0, 1.5707963267948966)*, *parameters=(9.81,)*)

Construct the pendulum problem with a signature $(u, \dot{u})$ / and an unused second argument.

See *pendulum()* for a more detailed problem description.

diffeqzoo.ivps Package

Initial value problem examples.

This module provides a number of example implementations of initial value problems (IVPs). IVPs are a combination of an ordinary differential equation

$$\dot{u}(t) = f(u(t), t, \theta)$$

and an initial condition $u(0) = u_0$. The initial values u_0 and the vector field f are known, the parameters θ might be known, and u is unknown.

The functions in this module construct implementations of this kind of problem. They (loosely) follow the input/output rule

```
f, u0, (t0, tmax), param = constructor()
```

where the constructor is, e.g., `lotka_volterra()` or `sir()`. This API specification is only loose, because every problem is different. For example, second-order problems implement a second-order differential equation

$$\ddot{u}(t) = f(u(t), \dot{u}(t), t, \theta)$$

subject to the initial conditions $u(0) = u_0$ and $\dot{u}(0) = u_1$. For these problems (e.g., `three_body_restricted()` or `van_der_pol()`), there are two initial values:

```
f, (u0, u1), (t0, tmax), param = constructor()
```

We try to stick as closely as possible to the above signature, but if problem-specific issues arise, we allow ourselves to deviate from this specification. When in doubt, consult the documentation of the respect constructor function.

Functions

affine_dependent(*[, initial_values, ...])	Construct an IVP with an affine vector field.
affine_independent(*[, initial_values, ...])	Construct an IVP with an affine vector field, where each dimension is treated independently.
fitzhugh_nagumo(*[, initial_values, ...])	Construct the FitzHugh-Nagumo model.
goodwin(*[, initial_values, time_span, r, ...])	Construct the Goodwin Oscillator dynamics.
heat_1d_dirichlet(*[, initial_values, ...])	Discretised heat equation in 1d with Dirichlet boundary.
henon_heiles(*[, initial_values, time_span, p])	Construct the Henon-Heiles problem.
henon_heiles_first_order(**kwargs)	Construct the Henon-Heiles problem as a first-order differential equation.
henon_heiles_with_unused_derivative_argument	Construct the Henon-Heiles problem as $\ddot{u}(t) = f(u(t), \dot{u}(t))$ (with an unused second argument).
hires(*[, initial_values, time_span])	Construct the High Irradiance Response (HIRES) problem.
logistic(*[, initial_value, time_span, ...])	Construct the logistic ODE model.
lorenz63(*[, initial_values, time_span, ...])	Construct the Lorenz63 model.
lorenz96(*[, initial_values, time_span, ...])	Construct the Lorenz96 model.
lotka_volterra(*[, initial_values, ...])	Construct the Lotka--Volterra / predator-prey model.
neural_ode_mlp(*[, initial_values, ...])	Construct an IVP with a neural ODE vector field.
nonlinear_chemical_reaction(*[, ...])	Construct the Nonlinear Chemical Reaction dynamics.
oregonator(*[, initial_values, time_span, ...])	Construct the scaled Oregonator Mass-Action dynamics in a well-stirred, homogeneous system.
pleiades(*[, initial_values, time_span])	Construct the Pleiades problem in its original, second-order form.
pleiades_first_order(**kwargs)	Construct the Pleiades problem as a first-order differential equation.
pleiades_with_unused_derivative_argument(...)	Construct the Pleiades problem as $\ddot{u}(t) = f(u(t), \dot{u}(t))$ (with an unused second argument).
rigid_body(*[, time_span, initial_values, ...])	Construct the rigid body dynamics without external forces.
rober(*[, initial_values, time_span, k1, k2, k3])	Construct the ROBER problem due to Robertson (1966).
roessler(*[, initial_values, time_span, ...])	Construct the Roessler model.
seir(*[, initial_values, time_span, alpha, ...])	Construct the SEIR model.
sir(*[, initial_values, time_span, beta, gamma])	Construct the SIR model without vital dynamics.
sird(*[, initial_values, time_span, beta, ...])	Construct the SIRD model.
three_body_restricted(*[, initial_values, ...])	Construct the restricted three-body problem as a second-order differential equation.
three_body_restricted_first_order(**kwargs)	Construct the restricted three-body problem as a first-order differential equation.
van_der_pol(*[, stiffness_constant, ...])	Construct the Van-der-Pol system as a second-order differential equation.
van_der_pol_first_order(**kwargs)	Construct the Van-der-Pol system as a first-order differential equation.

affine_dependent

diffeqzoo.ivps.**affine_dependent**(*, *initial_values=(1.0, 1.0)*, *time_span=(0.0, 1.0)*, *A=((1, 0), (0, 1))*, *b=(0, 0)*)

Construct an IVP with an affine vector field.

In Python code, this means `f(y, A, b) = A @ y + b`.

By default, this is a 2d-problem. Change the initial value to make this a multidimensional problem.

affine_independent

diffeqzoo.ivps.**affine_independent**(*, *initial_values=1.0*, *time_span=(0.0, 1.0)*, *a=1.0*, *b=0.0*)

Construct an IVP with an affine vector field, where each dimension is treated independently.

In Python code, this means `f(y, a, b)=a * y + b`.

By default, this is a scalar problem. Change the initial value to make this a multidimensional problem.

fitzhugh_nagumo

diffeqzoo.ivps.**fitzhugh_nagumo**(*, *initial_values=(-1.0, 1.0)*, *time_span=(0.0, 20.0)*, *parameters=(0.2, 0.2, 3.0)*)

Construct the FitzHugh-Nagumo model.

The FitzHugh-Nagumo model is a simple example of an excitable system (for example: a neuron). This simplified, 2d-version of the Hodgkin-Huxley model (which describes the spike generation in squid giant axons) was suggested by FitzHugh (1961) and Nagumo et al. (1962)

The following bibtex(s) point to the original papers about the FitzHugh-Nagumo models. (Source: Google Scholar).

```
@article{fitzhugh1961impulses,
    title={Impulses and physiological states in
    theoretical models of nerve membrane},
    author={FitzHugh, Richard},
    journal={Biophysical Journal},
    volume={1},
    number={6},
    pages={445--466},
    year={1961},
    publisher={Elsevier}
}
```

```
@article{nagumo1962active,
    title={An active pulse transmission line simulating nerve axon},
    author={Nagumo, Jinichi and Arimoto, Suguru and Yoshizawa, Shuji},
    journal={Proceedings of the IRE},
    volume={50},
    number={10},
    pages={2061--2070},
    year={1962},
```

(continues on next page)

(continued from previous page)

```
    publisher={IEEE}
}
```

goodwin

diffeqzoo.ivps.**goodwin**(*, *initial_values=(0.0, 0.0)*, *time_span=(0.0, 25.0)*, *r=10*, *a1=1.0*, *a2=3.0*, *alpha=0.5*, *k=(1.0,)*)

Construct the Goodwin Oscillator dynamics.

Describes a mechanism for periodic protein expression described by Goodwin (1965). The first dimension describes the mRNA concentration, the last dimension the protein inhibiting mRNA production, and the remaining dimensions correspond to intermediate protein species. r > 8 leads to oscillatory behavior. It is a n-dimensional ODE initial value problem. The length of the *k* needs to be *len(initial_values)-1*.

Common problem for parameter inference, where the posterior has a multimodal distribution.

```
@article{goodwin1965oscillatory,
    title = {Oscillatory behavior in enzymatic control processes},
    journal = {Advances in Enzyme Regulation},
    volume = {3},
    pages = {425-437},
    year = {1965},
    author = {Brian C. Goodwin},
}
```

heat_1d_dirichlet

diffeqzoo.ivps.**heat_1d_dirichlet**(*, *initial_values=None*, *time_span=(0.0, 1.0)*, *bounding_box=(0.0, 1.0)*, *num_gridpoints=10*, *coefficient=1.0*)

Discretised heat equation in 1d with Dirichlet boundary.

The discretisation uses second-order central differences. The vector field is evaluated with a convolution with zero-padding.

henon_heiles

diffeqzoo.ivps.**henon_heiles**(*, *initial_values=((0.5, 0.0), (0.0, 0.1))*, *time_span=(0.0, 100.0)*, *p=1.0*)

Construct the Henon-Heiles problem.

The Henon-Heiles problem relates to the non-linear motion of a star around a galactic center with the motion restricted to a plane. It is a 2-dimensional, second-order differential equation and commonly solved as a 4-dimensional, first-order equation. In in its original, second-order form, it is

$$\ddot{u}(t) = f(u(t)),$$

with nonlinear dynamics $f : \mathbb{R}^2 \to \mathbb{R}^2$.

The Henon-Heiles problem is not stiff. It is a popular benchmark problem because of its well-known Hamiltonian, which makes it a good test for symplectic integrators.

The Henon-Heiles problem is due to Henon and Heiles (1964).

```
@article{henon1964applicability,
    title={The applicability of the third integral of motion: some numerical␣
↪experiments},
    author={H{\'e}non, Michel and Heiles, Carl},
    journal={The astronomical journal},
    volume={69},
    pages={73},
    year={1964}
}
```

See also:

diffeqzoo.ivps.henon_heiles, *diffeqzoo.ivps.henon_heiles_with_unused_derivative_argument*, *diffeqzoo.ivps.henon_heiles_first_order*

henon_heiles_first_order

diffeqzoo.ivps.**henon_heiles_first_order**(***kwargs*)

Construct the Henon-Heiles problem as a first-order differential equation.

> **Warning:** This problem has been generated by wrapping the function *henon_heiles_with_unused_derivative_argument()* through the function *diffeqzoo.transform.second_to_first_order_auto()*.
>
> The problem is not originally of first order. If you have access to solvers for second-order problems, it might be more efficient to solve the original problem.

See *henon_heiles()* for a more detailed problem description.

See also:

diffeqzoo.ivps.henon_heiles, *diffeqzoo.ivps.henon_heiles_with_unused_derivative_argument*, *diffeqzoo.ivps.henon_heiles_first_order*

henon_heiles_with_unused_derivative_argument

diffeqzoo.ivps.**henon_heiles_with_unused_derivative_argument**(***kwargs*)

Construct the Henon-Heiles problem as $\ddot{u}(t) = f(u(t), \dot{u}(t))$ (with an unused second argument).

See *henon_heiles()* for a more detailed problem description.

See also:

diffeqzoo.ivps.henon_heiles, *diffeqzoo.ivps.henon_heiles_with_unused_derivative_argument*, *diffeqzoo.ivps.henon_heiles_first_order*

hires

diffeqzoo.ivps.**hires**(*, *initial_values=(1.0, 0.0, 0.0, 0, 0, 0, 0, 0.0057)*, *time_span=(0.0, 321.8122)*)

Construct the High Irradiance Response (HIRES) problem.

The "High Irradiance Response" ODE (HIRES) from plant physiology describes how light is involved in morphogenesis. It was proposed by Schäfer (1975) and named "HIRES" by Hairer and Wanner (1996).

It is a system of 8 nonlinear differential equations,

$$\dot{u}(t) = f(u(t))$$

and a common testproblem for ODE solvers that can handle stiff problems.

The following bibtex(s) point to the original paper about the HIRES model and the book by Hairer and Wanner. (Source: Google Scholar).

```
@article{schafer1975new,
    title={A new approach to explain the "high irradiance responses"
    of photomorphogenesis on the basis of phytochrome},
    author={Sch{\"a}fer, E},
    journal={Journal of Mathematical Biology},
    volume={2},
    number={1},
    pages={41--56},
    year={1975},
    publisher={Springer}
}
```

```
@book{hairer1996solving,
    title={Solving Ordinary Differential Equations II,
    Stiff and Differential-Algebraic Problems},
    author={Hairer, Ernst and Wanner, Gerhard},
    year={1996},
    publisher={Springer}
}
```

logistic

diffeqzoo.ivps.**logistic**(*, *initial_value=0.1*, *time_span=(0.0, 2.5)*, *parameters=(1.0, 1.0)*)

Construct the logistic ODE model.

The logistic ODE is a differential equation model whose solution exhibits exponential growth early in the time interval, and approaches a constant value over time.

It is a differential equation version of the sigmoid and the logistic function. The logistic ODE has a closed-form solution.

Note: Help wanted!

If you know which paper/book to cite when the logistic ODE is used in a paper, please consider making a contribution.

---

lorenz63

diffeqzoo.ivps.**lorenz63**(*, *initial_values=(0.0, 1.0, 1.05)*, *time_span=(0.0, 20.0)*, *parameters=(10.0, 28.0, 2.6666666666666665)*)

Construct the Lorenz63 model.

The Lorenz63 model, initially used for atmospheric convection, is a common example of an initial value problem that has a chaotic solution.

It was proposed by Lorenz (1963).

```
@article{lorenz1963deterministic,
    title={Deterministic nonperiodic flow},
    author={Lorenz, Edward N},
    journal={Journal of atmospheric sciences},
    volume={20},
    number={2},
    pages={130--141},
    year={1963}
}
```

lorenz96

diffeqzoo.ivps.**lorenz96**(*, *initial_values=None*, *time_span=(0.0, 30.0)*, *num_variables=10*, *forcing=8.0*, *perturb=0.01*)

Construct the Lorenz96 model.

The Lorenz96 is a chaotic initial value problem, due to Lorenz (1996), and commonly used as a testproblem in data assimilation.

```
@inproceedings{lorenz1996predictability,
    title={Predictability: A problem partly solved},
    author={Lorenz, Edward N},
    booktitle={Proceedings of the Seminar on Predictability},
    volume={1},
    number={1},
    year={1996}
}
```

lotka_volterra

diffeqzoo.ivps.**lotka_volterra**(*, *initial_values=(20.0, 20.0)*, *time_span=(0.0, 20.0)*, *parameters=(0.5, 0.05, 0.5, 0.05)*)

Construct the Lotka–Volterra / predator-prey model.

The Lotka–Volterra equations describe the dynamics of biological systems in which two species, predators and prey, interact.

The original version is due to Lotka (1910). Its application to predator-Prey dynamics is due to Lotka (1925). The same model was discovered by Volterra (1926).

```
@article{lotka1910contribution,
    title={Contribution to the theory of periodic reactions},
    author={Lotka, Alfred J},
    journal={The Journal of Physical Chemistry},
    volume={14},
    number={3},
    pages={271--274},
    year={1910},
    publisher={ACS Publications}
}
```

```
@book{lotka1925elements,
    title={Elements of physical biology},
    author={Lotka, Alfred James},
    year={1925},
    publisher={Williams \& Wilkins}
}
```

```
@book{volterra1926variazioni,
    title={Variazioni e fluttuazioni del numero d'individui in specie animali␣
↪conviventi},
    author={Volterra, Vito},
    year={1926},
    publisher={Societ{\`a} anonima tipografica" Leonardo da Vinci"}
}
```

neural_ode_mlp

diffeqzoo.ivps.**neural_ode_mlp**(*, *initial_values=(0.0,)*, *time_span=(0.0, 1.0)*, *scale=1.0*, *layer_sizes=(2, 20, 1)*, *seed=1234*)

Construct an IVP with a neural ODE vector field.

The vector field is given by a neural network.

The neural network is a multi-layer perceptron with *tanh* activation functions.

We implement the dynamics used in the "implicit-layers" tutorial: `http://implicit-layers-tutorial.org/neural_odes/`.

Note: The neural network is not trained in this function. This function only constructs the IVP.

nonlinear_chemical_reaction

diffeqzoo.ivps.**nonlinear_chemical_reaction**(*, *initial_values=(1.0, 0.0, 0.0)*, *time_span=(0.0, 1.0)*, *k1=1*, *k2=1*)

Construct the Nonlinear Chemical Reaction dynamics.

We use the version described by Liu et al. (2012).

```
@article{liu2012analytic,
    title={Analytic solution for a nonlinear chemistry system of ordinary␣
↪differential equations},
    author={Liu, Li-Cai and Tian, Bo and Xue, Yu-Shan and Wang, Ming and Liu, Wen-
↪Jun},
    journal={Nonlinear Dynamics},
    volume={68},
    number={1},
    pages={17--21},
    year={2012},
    publisher={Springer}
}
```

Note: If you know a better source of this (class of) problem(s), please consider making a pull request.

oregonator

diffeqzoo.ivps.**oregonator**(*, *initial_values=(1.0, 2.0, 3.0)*, *time_span=(0.0, 100000.0)*, *s=77.27*, *q=8.375e-06*, *w=0.161*)

Construct the scaled Oregonator Mass-Action dynamics in a well-stirred, homogeneous system.

What is often referred to as the "Oregonator" problem is a simplified model of the chemical dynamics of the oscillatory Belousov-Zhabotinsky reaction and due to Fields and Noyes (1974). It is a three-dimensional, stiff initial value problem,

$$\dot{u}(t) = f(u(t))$$

and a common test problem for numerical solvers for stiff differential equations.

```
@article{field1974oscillations,
    title={Oscillations in chemical systems. IV. Limit cycle behavior in a model of␣
↪a real chemical reaction},
    author={Field, Richard J and Noyes, Richard M},
    journal={The Journal of Chemical Physics},
    volume={60},
    number={5},
    pages={1877--1884},
    year={1974},
    publisher={American Institute of Physics}
}
```

pleiades

diffeqzoo.ivps.**pleiades**(*, *initial_values=((3.0, 3.0, -1.0, -3.0, 2.0, -2.0, 2.0, 3.0, -3.0, 2.0, 0.0, 0.0, -4.0, 4.0), (0.0, 0.0, 0.0, 0.0, 0.0, 1.75, -1.5, 0.0, 0.0, 0.0, -1.25, 1.0, 0.0, 0.0))*, *time_span=(0.0, 3.0)*)

Construct the Pleiades problem in its original, second-order form.

The Pleiades problem from celestial mechanics describes the gravitational interaction(s) of seven stars (the "Pleiades", or "Seven Sisters") in a plane. It is a 14-dimensional, second-order differential equation and commonly solved as a 28-dimensional, first-order equation. In in its original, second-order form, it is

$$\ddot{u}(t) = f(u(t)),$$

with nonlinear dynamics $f : \mathbb{R}^{14} \to \mathbb{R}^{14}$.

The Pleiades problem is not stiff. It is a popular benchmark problem because it is not very difficult to solve numerically, but (a) it requires high accuracy in each ODE solver step, and (b) its 14 (or 28) dimensions start to expose those numerical solvers that do not scale well to high dimensions.

A common citation for the Pleiades problem is p. 245 in the book by Hairer et al. (1993):

```
@book{hairer1993solving,
    title={Solving Ordinary Differential equations I, Nonstiff Problems},
    author={Hairer, Ernst and N{\o}rsett, Syvert P and Wanner, Gerhard},
    year={1993},
    publisher={Springer}
    edition={2}
}
```

Note: If you know a more suitable original reference, please make some noise!

See also:

diffeqzoo.ivps.pleiades, *diffeqzoo.ivps.pleiades_with_unused_derivative_argument*, *diffeqzoo.ivps.pleiades_first_order*

pleiades_first_order

diffeqzoo.ivps.**pleiades_first_order**(**kwargs*)

Construct the Pleiades problem as a first-order differential equation.

Warning: This problem has been generated by wrapping the function *pleiades_with_unused_derivative_argument()* through the function *diffeqzoo.transform.second_to_first_order_auto()*.

The problem is not originally of first order. If you have access to solvers for second-order problems, it might be more efficient to solve the original problem.

See *pleiades()* for a more detailed problem description.

See also:

diffeqzoo.ivps.pleiades, *diffeqzoo.ivps.pleiades_with_unused_derivative_argument*, *diffeqzoo.ivps.pleiades_first_order*

pleiades_with_unused_derivative_argument

diffeqzoo.ivps.**pleiades_with_unused_derivative_argument**(***kwargs*)

Construct the Pleiades problem as $\ddot{u}(t) = f(u(t), \dot{u}(t))$ (with an unused second argument).

See *pleiades()* for a more detailed problem description.

See also:

diffeqzoo.ivps.pleiades, *diffeqzoo.ivps.pleiades_with_unused_derivative_argument*, *diffeqzoo.ivps.pleiades_first_order*

rigid_body

diffeqzoo.ivps.**rigid_body**(**, time_span=(0.0, 20.0), initial_values=(1.0, 0.0, 0.9), parameters=(-2.0, 1.25, -0.5)*)

Construct the rigid body dynamics without external forces.

The rigid body dynamics from classical mechanics, or "Euler's rotation equations", describe the rotation of a rigid body in three-dimensional, principal, orthogonal coordinates.

A common citation for the rigid-body problem is p. 244 in the book by Hairer et al. (1993):

```
@book{hairer1993solving,
    title={Solving Ordinary Differential equations I, Nonstiff Problems},
    author={Hairer, Ernst and N{\o}rsett, Syvert P and Wanner, Gerhard},
    year={1993},
    publisher={Springer}
    edition={2}
}
```

Note: If you know a more suitable original reference, please make some noise!

rober

diffeqzoo.ivps.**rober**(**, initial_values=(1.0, 0.0, 0.0), time_span=(0.0, 100000.0), k1=0.04, k2=30000000.0, k3=10000.0*)

Construct the ROBER problem due to Robertson (1966).

The ROBER problem describes the kinetics of an auto-catalytic reaction, and was proposed by Robertson (1966). It was named "ROBER" by Hairer and Wanner (1996).

It is a three-dimensional, stiff initial value problem,

$$\dot{u}(t) = f(u(t))$$

and a common test problem for numerical solvers for stiff differential equations.

The following bibtex(s) point to the original paper about the ROBER model and the book by Hairer and Wanner. (Source: Google Scholar).

```
@article{robertson1966solution,
    title={The solution of a set of reaction rate equations},
    author={Robertson, HH},
    journal={Numerical Analysis: An Introduction},
    publisher={Academic Press},
    year={1966},
    pages={178-182},
}
```

```
@book{wanner1996solving,
    title={Solving Ordinary Differential Equations II,
    Stiff and Differential-Algebraic Problems},
    author={Hairer, Ernst and Wanner, Gerhard},
    year={1996},
    publisher={Springer}
}
```

roessler

diffeqzoo.ivps.**roessler**(*, *initial_values=(1.0, 0.0, 0.0)*, *time_span=(0.0, 100.0)*, *parameters=(0.1, 0.1, 14.0)*)

Construct the Roessler model.

The Roessler model is a three-dimensional chaotic system, that was proposed by Roessler (1990).

```
@article{rossler1976equation,
    title={An equation for continuous chaos},
    author={R{"o}ssler, Otto E},
    journal={Physics Letters A},
    volume={57},
    number={5},
    pages={397--398},
    year={1976},
    publisher={Elsevier}
}
```

seir

diffeqzoo.ivps.**seir**(*, *initial_values=(998.0, 1.0, 1.0, 1.0)*, *time_span=(0.0, 200.0)*, *alpha=0.3*, *beta=0.3*, *gamma=0.1*)

Construct the SEIR model.

The SEIR model is a variant of the SIR model, but additionally includes a compartment of the population that has been exposed to the virus (but is not infected yet). See Hethcode (2000).

```
@article{hethcote2000mathematics,
    title={The Mathematics of Infectious Diseases},
    author={Hethcote, Herbert W},
    journal={SIAM Review},
    volume={42},
```

(continues on next page)

(continued from previous page)

```
    number={4},
    pages={599--653},
    year={2000},
    publisher={SIAM}
}
```

Note: If you know a more suitable original reference, please make some noise!

See also:

`ivps.sir`, `ivps.sird`

sir

`diffeqzoo.ivps.sir`(*, *initial_values=(998.0, 1.0, 1.0)*, *time_span=(0.0, 200.0)*, *beta=0.3*, *gamma=0.1*)

Construct the SIR model without vital dynamics.

The SIR model describes the spread of a virus in a population. More specifically, it describes how populations move from being susceptible, to being infected, to being removed from the population.

It was first proposed by Kermack and McKendrick (1927).

```
@article{kermack1927contribution,
    title={A contribution to the mathematical theory of epidemics},
    author={Kermack, William Ogilvy and McKendrick, Anderson G},
    journal={Proceedings of the Royal Society of London. Series A},
    volume={115},
    number={772},
    pages={700--721},
    year={1927},
    publisher={The Royal Society London}
}
```

See also:

`ivps.seir`, `ivps.sird`

sird

`diffeqzoo.ivps.sird`(*, *initial_values=(998.0, 1.0, 1.0, 0.0)*, *time_span=(0.0, 200.0)*, *beta=0.3*, *gamma=0.1*, *eta=0.005*)

Construct the SIRD model.

The SIRD model is a variant of the SIR model that distinguishes the recovered compartment from the deceased compartment in the population. See Hethcode (2000).

```
@article{hethcote2000mathematics,
    title={The Mathematics of Infectious Diseases},
    author={Hethcote, Herbert W},
    journal={SIAM Review},
    volume={42},
```

(continues on next page)

(continued from previous page)

```
    number={4},
    pages={599--653},
    year={2000},
    publisher={SIAM}
}
```

Note: If you know a more suitable original reference, please make some noise!

See also:

`ivps.sir`, `ivps.seir`

three_body_restricted

`diffeqzoo.ivps.`**`three_body_restricted`**(*, *initial_values=((0.994, 0), (0, -2.0015851063790824))*, *standardised_moon_mass=0.012277471*, *time_span=(0.0, 17.065216560157964)*)

Construct the restricted three-body problem as a second-order differential equation.

The restricted three-body problem describes how a body of negligible mass moves under the influence of two massive bodies. It can be described in terms of two-body motion.

It is commonly pointed to p. 129 of Hairer et al. (1993) as the first reference.

```
@book{hairer1993solving,
    title={Solving Ordinary Differential equations I, Nonstiff Problems},
    author={Hairer, Ernst and N{\o}rsett, Syvert P and Wanner, Gerhard},
    year={1993},
    publisher={Springer}
    edition={2}
}
```

three_body_restricted_first_order

`diffeqzoo.ivps.`**`three_body_restricted_first_order`**(***kwargs*)

Construct the restricted three-body problem as a first-order differential equation.

Warning: This problem has been generated by wrapping the function *three_body_restricted()* through the function *diffeqzoo.transform.second_to_first_order_auto()*.

The problem is not originally of first order. If you have access to solvers for second-order problems, it might be more efficient to solve the original problem.

a second-order differential equation.

The restricted three-body problem describes how a body of negligible mass moves under the influence of two massive bodies. It can be described in terms of two-body motion.

It is commonly pointed to p. 129 of Hairer et al. (1993) as the first reference.

```
@book{hairer1993solving,
    title={Solving Ordinary Differential equations I, Nonstiff Problems},
    author={Hairer, Ernst and N{\o}rsett, Syvert P and Wanner, Gerhard},
    year={1993},
    publisher={Springer}
    edition={2}
}
```

van_der_pol

diffeqzoo.ivps.**van_der_pol**(*, *stiffness_constant=1.0*, *initial_values=(2.0, 0.0)*, *time_span=(0.0, 6.3)*)

Construct the Van-der-Pol system as a second-order differential equation.

The Van-der-Pol system is a non-conservative oscillator subject to non-linear damping. It is a popular benchmark problem, because it involves a parameter μ (the "stiffness constant") which governs the stiffness of the problem. For μ 1, the problen is not stiff. For large values (e.g. μ 10^6) the problem is stiff. It was first published by Van der Pol (1920).

```
@article{van1920theory,
    title={Theory of the amplitude of free and forced triode vibrations},
    author={Van der Pol, Balthasar},
    journal={Radio Review},
    volume={1},
    pages={701--710},
    year={1920}
}
```

van_der_pol_first_order

diffeqzoo.ivps.**van_der_pol_first_order**(**kwargs*)

Construct the Van-der-Pol system as a first-order differential equation.

> **Warning:** This problem has been generated by wrapping the function *van_der_pol()* through the function *diffeqzoo.transform.second_to_first_order_auto()*.
>
> The problem is not originally of first order. If you have access to solvers for second-order problems, it might be more efficient to solve the original problem.

The Van-der-Pol system is a non-conservative oscillator subject to non-linear damping. It is a popular benchmark problem, because it involves a parameter μ (the "stiffness constant") which governs the stiffness of the problem. For μ 1, the problen is not stiff. For large values (e.g. μ 10^6) the problem is stiff. It was first published by Van der Pol (1920).

```
@article{van1920theory,
    title={Theory of the amplitude of free and forced triode vibrations},
    author={Van der Pol, Balthasar},
    journal={Radio Review},
    volume={1},
    pages={701--710},
```

(continues on next page)

(continued from previous page)

```
    year={1920}
}
```

diffeqzoo.transform Module

Transform ODE models into equivalent versions.

Functions

long_description(description, /)	Add a long description to the docstring of a function.
replace_short_summary(docstring, /, *, ...)	Replace the short summary in a docstring with a new short summary.
second_to_first_order_auto(ivp_fn, /[, ...])	Transform a second-order, autonomous differential equation into an equivalent first-order form.
second_to_first_order_vf_auto(fn, /[, ...])	Transform the vector-field of a second-order, autonomous differential equation into an equivalent first-order form.

long_description

diffeqzoo.transform.**long_description**(*description*, */*)

Add a long description to the docstring of a function.

Use this function as a decorator.

replace_short_summary

diffeqzoo.transform.**replace_short_summary**(*docstring*, */*, *, *short_summary*)

Replace the short summary in a docstring with a new short summary.

second_to_first_order_auto

diffeqzoo.transform.**second_to_first_order_auto**(*ivp_fn*, */*, *short_summary=None*)

Transform a second-order, autonomous differential equation into an equivalent first-order form.

second_to_first_order_vf_auto

diffeqzoo.transform.**second_to_first_order_vf_auto**(*fn*, */*, *short_summary=None*)

Transform the vector-field of a second-order, autonomous differential equation into an equivalent first-order form.

diffeqzoo.vector_fields Module

Vector fields for differential equations.

Functions

affine_dependent(u, /, A, b)	Affine ODE.
affine_independent(u, /, a, b)	Affine ODE.
bratu(u, /, k)	Bratu's problem.
bratu_with_unused_derivative_argument(u, _, /, k)	Bratu's problem with signature (u, u').
fitzhugh_nagumo(u, /, a, b, c)	FitzHugh--Nagumo model.
goodwin(u, /, r, a1, a2, alpha, k)	Goodwin oscillator.
heat_1d_dirichlet(y, weights, coeff)	Discretized heat equation with Dirichlet (i.e.
henon_heiles(u, /, p)	Henon-Heiles dynamics as a second-order differential equation.
henon_heiles_first_order(u, *args)	Henon-Heiles dynamics as a first-order differential equation.
henon_heiles_with_unused_derivative_argument(...)	Henon-Heiles dynamics as a second-order differential equation (with an unused second argument).
hires(u, /)	High irradiance response.
logistic(u, p0, p1, /)	Logistic ODE dynamics.
lorenz63(u, /, a, b, c)	Lorenz63 dynamics.
lorenz96(y, /, forcing)	Lorenz96 dynamics.
lotka_volterra(y, /, a, b, c, d)	Lotka--Volterra dynamics.
measles(t, u, /, mu, lmbda, eta, beta0)	Measles problem.
neural_ode_mlp(state, time, /, params)	Neural ODE based on a multi-layer perceptron.
nonlinear_chemical_reaction(u, /, k1, k2)	Nonlinear chemical reaction.
oregonator(u, /, s, q, w)	Oregonator problem.
pendulum(u, /, p)	Bratu's problem.
pendulum_with_unused_derivative_argument(u, ...)	Bratu's problem.
pleiades(u, /)	Evaluate the Pleiades vector field in its original, second-order form.
pleiades_first_order(u, *args)	The Pleiades problem as a first-order differential equation.
pleiades_with_unused_derivative_argument(u, _, /)	Evaluate the Pleiades vector field as $\ddot{u}(t) = f(u(t), \dot{u}(t))$ (with an unused second argument).
rigid_body(y, /, p1, p2, p3)	Rigid body dynamics without external forces.
rober(u, /, k1, k2, k3)	'Rober' ODE problem.
roessler(u, /, a, b, c)	Roessler attractor.
seir(u, /, alpha, beta, gamma, population_count)	SEIR model.
sir(u, /, beta, gamma, population_count)	SIR model.
sird(u, /, beta, gamma, eta, population_count)	SIRD model.
three_body_restricted(Y, dY, /, ...)	Restricted three-body dynamics as a second-order differential equation.
three_body_restricted_first_order(u, *args)	Restricted three-body dynamics as a first-order differential equation.
van_der_pol(u, du, /, stiffness_constant)	Van-der-Pol dynamics as a second-order differential equation.

continues on next page

Table 1 – continued from previous page

van_der_pol_first_order(u, *args)	Van-der-Pol dynamics as a first-order differential equation.

affine_dependent

diffeqzoo.vector_fields.**affine_dependent**(*u*, */*, *A*, *b*)

Affine ODE.

affine_independent

diffeqzoo.vector_fields.**affine_independent**(*u*, */*, *a*, *b*)

Affine ODE.

Each state is scaled and shifted independently.

bratu

diffeqzoo.vector_fields.**bratu**(*u*, */*, *k*)

Bratu's problem.

bratu_with_unused_derivative_argument

diffeqzoo.vector_fields.**bratu_with_unused_derivative_argument**(*u*, *_*, */*, *k*)

Bratu's problem with signature (u, u').

fitzhugh_nagumo

diffeqzoo.vector_fields.**fitzhugh_nagumo**(*u*, */*, *a*, *b*, *c*)

FitzHugh–Nagumo model.

goodwin

diffeqzoo.vector_fields.**goodwin**(*u*, */*, *r*, *a1*, *a2*, *alpha*, *k*)

Goodwin oscillator.

heat_1d_dirichlet

diffeqzoo.vector_fields.**heat_1d_dirichlet**(*y*, *weights*, *coeff*)

Discretized heat equation with Dirichlet (i.e. zero) boundary conditions.

henon_heiles

diffeqzoo.vector_fields.**henon_heiles**(*u*, */*, *p*)

Henon-Heiles dynamics as a second-order differential equation.

henon_heiles_first_order

diffeqzoo.vector_fields.**henon_heiles_first_order**(*u*, **args*)

Henon-Heiles dynamics as a first-order differential equation.

> **Warning:** This problem has been generated by wrapping the function *henon_heiles_with_unused_derivative_argument()* through the function *diffeqzoo.transform.second_to_first_order_vf_auto()*.
>
> The problem is not originally of first order. If you have access to solvers for second-order problems, it might be more efficient to solve the original problem.

henon_heiles_with_unused_derivative_argument

diffeqzoo.vector_fields.**henon_heiles_with_unused_derivative_argument**(*u*, *_*, */*, *p*)

Henon-Heiles dynamics as a second-order differential equation (with an unused second argument).

hires

diffeqzoo.vector_fields.**hires**(*u*, */*)

High irradiance response.

logistic

diffeqzoo.vector_fields.**logistic**(*u*, *p0*, *p1*, */*)

Logistic ODE dynamics.

lorenz63

diffeqzoo.vector_fields.**lorenz63**(*u*, */*, *a*, *b*, *c*)

Lorenz63 dynamics.

lorenz96

diffeqzoo.vector_fields.**lorenz96**(*y*, */*, *forcing*)

Lorenz96 dynamics.

lotka_volterra

diffeqzoo.vector_fields.**lotka_volterra**(*y*, */*, *a*, *b*, *c*, *d*)

Lotka–Volterra dynamics.

measles

diffeqzoo.vector_fields.**measles**(*t*, *u*, */*, *mu*, *lmbda*, *eta*, *beta0*)

Measles problem.

neural_ode_mlp

diffeqzoo.vector_fields.**neural_ode_mlp**(*state*, *time*, */*, *params*)

Neural ODE based on a multi-layer perceptron.

nonlinear_chemical_reaction

diffeqzoo.vector_fields.**nonlinear_chemical_reaction**(*u*, */*, *k1*, *k2*)

Nonlinear chemical reaction.

oregonator

diffeqzoo.vector_fields.**oregonator**(*u*, */*, *s*, *q*, *w*)

Oregonator problem.

pendulum

diffeqzoo.vector_fields.**pendulum**(*u*, */*, *p*)

Bratu's problem.

pendulum_with_unused_derivative_argument

diffeqzoo.vector_fields.**pendulum_with_unused_derivative_argument**(*u*, *_*, */*, *p*)

Bratu's problem.

pleiades

diffeqzoo.vector_fields.**pleiades**(*u*, */*)

Evaluate the Pleiades vector field in its original, second-order form.

pleiades_first_order

diffeqzoo.vector_fields.**pleiades_first_order**(*u*, **args*)

The Pleiades problem as a first-order differential equation.

> **Warning:** This problem has been generated by wrapping the function *pleiades_with_unused_derivative_argument()* through the function *diffeqzoo.transform.second_to_first_order_vf_auto()*.
>
> The problem is not originally of first order. If you have access to solvers for second-order problems, it might be more efficient to solve the original problem.

pleiades_with_unused_derivative_argument

diffeqzoo.vector_fields.**pleiades_with_unused_derivative_argument**(*u*, *_*, */*)

Evaluate the Pleiades vector field as $\ddot{u}(t) = f(u(t), \dot{u}(t))$ (with an unused second argument).

rigid_body

diffeqzoo.vector_fields.**rigid_body**(*y*, */*, *p1*, *p2*, *p3*)

Rigid body dynamics without external forces.

rober

diffeqzoo.vector_fields.**rober**(*u*, */*, *k1*, *k2*, *k3*)

'Rober' ODE problem.

roessler

diffeqzoo.vector_fields.**roessler**(*u*, */*, *a*, *b*, *c*)

Roessler attractor.

seir

diffeqzoo.vector_fields.**seir**(*u*, */*, *alpha*, *beta*, *gamma*, *population_count*)

SEIR model.

sir

diffeqzoo.vector_fields.**sir**(*u*, */*, *beta*, *gamma*, *population_count*)

SIR model.

sird

diffeqzoo.vector_fields.**sird**(*u*, */*, *beta*, *gamma*, *eta*, *population_count*)

SIRD model.

three_body_restricted

diffeqzoo.vector_fields.**three_body_restricted**(*Y*, *dY*, */*, *standardised_moon_mass*)

Restricted three-body dynamics as a second-order differential equation.

three_body_restricted_first_order

diffeqzoo.vector_fields.**three_body_restricted_first_order**(*u*, **args*)

Restricted three-body dynamics as a first-order differential equation.

> **Warning:** This problem has been generated by wrapping the function *three_body_restricted()* through the function *diffeqzoo.transform.second_to_first_order_vf_auto()*.
>
> The problem is not originally of first order. If you have access to solvers for second-order problems, it might be more efficient to solve the original problem.

van_der_pol

diffeqzoo.vector_fields.**van_der_pol**(*u*, *du*, */*, *stiffness_constant*)

Van-der-Pol dynamics as a second-order differential equation.

van_der_pol_first_order

diffeqzoo.vector_fields.**van_der_pol_first_order**(*u*, **args*)

Van-der-Pol dynamics as a first-order differential equation.

> **Warning:** This problem has been generated by wrapping the function *van_der_pol()* through the function *diffeqzoo.transform.second_to_first_order_vf_auto()*.
>
> The problem is not originally of first order. If you have access to solvers for second-order problems, it might be more efficient to solve the original problem.

5.9 Contribution

Open PRs.

5.10 Adding a problem implementation

First off: thank you for considering a contribution to `diffeqzoo`'s problem test set. Your example will be a valuable addition! Here is how to proceed:

1. Add a test-case to `tests/test_bvps.py` or `tests/test_ivps.py`. This is important, because we need to verify that implementations work. You can use the existing test-cases as a guideline.
2. Implement the ODE vector field in `diffeqzoo/_vector_fields.py`. Implementing the vector field separately from the IVP/BVP constructor has the advantage that some vector fields may be used for multiple problems. For example, the SIR vector field powers both `ivps.sir` and `bvps.measles`.
3. Implement the IVP/BVP constructore in `diffeqzoo/ivps.py` or `diffeqzoo/bvps.py`. Again, use the existing code as a reference.
4. Add a short docstring that describes the problem, why it is interesting, and who came up with it (if possible). It would be great if you could add a bibtex snippet that points to the original paper, just like in the existing problems. The better the docs, the more useful the function will be to end-users, but if something is hard to find, don't sweat it too much.
5. Check that everything passes the quality checks via `make format test lint`.
6. Make a pull request with your changes.

5.11 Adding an example Jupyter notebook

First off: thank you for considering a contribution to `diffeqzoo`'s examples. The more examples we have, the better! Here is how to add a notebook.

1. Create the .ipynb file in `docs/source/example_notebooks` and fill it with content
2. Sync the notebook to a markdown file via jupytext. Check out these instructions. This is useful to keep the git diff clean and legible.
3. Make sure the `example` dependencies in `setup.cfg` are sufficient. We try to keep the optional dependencies small, but are not as strict with the `example` dependencies as with the remaining ones.
4. Add the notebook to `docs/source/index.rst`. If you skip this step, the notebook will not be rendered in the docs.
5. Add the .ipynb file to the .gitignore and to the `clean` job in the makefile. Do this after syncing it to a markdown version.
6. Check that everything passes the quality checks via `make format example doc lint`.

7. Make a pull request with your changes.

5.12 Internal design choices

`diffeqzoo` is a database of ODE problems. As such, the following principles apply to the source:

It must be compatible with all numpy/jax-based ODE solvers.

It must be easy to copy/paste from, if desired.

It must not have opinions (we don't care whether your ODE variable is called `u`, `x`, or `y`).

It must take non-standard dynamics seriously: if an ODE is of second order, autonomous, or in mass-matrix form, it is implemented as such (we trust the user to translate it to an appropriate first-order version).

It should provide all information that might be relevant if the problem appears in a paper (citation, maths, meaning).

`diffeqzoo` must be extremely easy to maintain, even if it costs a tiny bit of user-friendliness.

5.13 Continuous integration

The continuous integration has multiple components,

- format: apply black, isort, etc.. Includes notebooks.
- lint: check flake8, and check whether black and isort are happy with the code. No formatting.
- test: Run the unittests with different backends, and run some doctests
- example: sync and execute the example notebooks. Bundles a bunch of bigger optional dependencies (scipy, diffrax, matplotlib, ...), so it is kept separate from the rest of the dependencies.
- doc: build the sphinx docs.

which appear in different parts of the specification

- Groups of optional dependencies: `pip install diffeqzoo[lint,test,example]`. Formatting dependencies are a subset of the linting dependencies.
- In the makefile: `make format; make lint; make test; make example; make doc`
- In the workflows

One exception are the doc-requirements, which are not part of the setup.cfg, but are isolated in a separate requirements file in the docs/ directory. (Reason: readthedocs config.)

There are also the `numpy` and the `jax` optional dependencies. Combine any of the execution-based workflows (test, example) with either one, e.g.,

```
pip install diffeqzoo[jax]
pip install diffeqzoo[example,numpy]
pip install diffeqzoo[test,jax,numpy]
```

5.13.1 The makefile

To apply *all* {linting, formatting, testing, ...} operations, use the makefile. In the root, run

```
make format
make lint
make test
```

To remove automatically generated files (caches, etc.), run `make clean`.

To check conformity with the pre-commit file, run `make pre-commit`. This command also updates the versions in the pre-commit.

PYTHON MODULE INDEX

d

INDEX

O

P

R

S

T

V