DINJO

Release 0.1.1

Juan Esteban Aristizabal-Zuluaga and FEnFiSDi

Jun 08, 2021

GETTING STARTED

DINJO Is Not Just An Optimizer is a Python framework designed for the optimization of initial value problems' parameters.

Lets say you have some 'experimental' data of a state variable S corresponding to the initial value problem

$$d\mathbf{S}/dt = \mathbf{f}(t, S; \mathbf{p})$$
$$\mathbf{S}(t_0) = \mathbf{f}_0,$$

where $\mathbf{p}$ is a list of parameters, $\mathbf{f}_0$ and t_0 are constants.

If you want to know the optimal value of $\mathbf{p}$ so that the solution of the initial value problem fits your experimental data, you can use DINJO to get an approximate value of the optimal $\mathbf{p}$.

CHAPTER

ONE

GETTING STARTED

1.1 Examples

1.1.1 Harmonic Oscillator

Lets take the unit mass harmonic oscillator example to show you how to use our library.

The initial value problem can be stated in terms of the Hamilton equations as

$$\frac{dq}{dt} = p$$
$$\frac{dp}{dt} = -\omega^2 q,$$

and an initial condition $q(t_0) = q_0$, $p(t_0) = p_0$, where q represents the position of the particle and p the momentum.

The problem we will solve: find the value of ω that best fits to the IVP, given an experimental –noisy– solution.

Note that in this case we could find the optimal value of ω by directly fitting the noisy data to the parametrized solution of the harmonic oscillator because it is a well known solution. However, in most other problems, the exact parametrized solution is not known and thus the problem could not be solved that way. In that sense, this is a pedagogical example.

The step by step process is given as follows:

1. Define your IVP using the class *dinjo.model.ModelIVP*.

```
from dinjo.model import ModelIVP

# Define the IVP
class ModelOscillator(ModelIVP):
    def build_model(self, t, y, w):
        """Harmonic Oscillator differential equations
        """
        q, p = y
        # Hamilton's equations
        dydt = [
            p,
            - (w ** 2) * q
        ]
        return dydt
```

2. Note that the method `ModelOscillator.build_model` implicitely uses the state variables `p` and `q` and the parameter `w`. So, the next step is defining our State Variables and Parameters as dinjo objects:

```
from dinjo.model import StateVariable, Parameter
import numpy as np


# Define State Variables
q = StateVariable(
    name='position', representation='q', initial_value=1.0
)
p = StateVariable(
    name='momentum', representation='p', initial_value=0.0
)

# Define Paramters
omega = Parameter(
    name='frequency', representation='w',
    initial_value=2 * np.pi, bounds=[4, 8]
)
```

3. The State Variables and parameters must be encapsulated into lists in the same order implicetely defined in `ModelOscillator.build_model`: the Parameters must be

in the same order as the method's signature and the State Variables must be in the same order in which they are unpacked from the `y` parameter. In this case:

```
state_vars = [q, p]
params = [omega]
```

4. Now, we instantiate the `ModelOscillator` class which will contain all the information of the oscillator IVP. The initial values are implicitely assumed at time `t0 = t_span[0]`. So, the initial values are `p(t_span[0]) = p.initial_value` and `q(t_span[0]) = q.initial_value`. In this case $q(0) = 1$ and $p(0) = 0$.

```
t_span = [0, 1]
t_steps = 50

# Instantiate the IVP class with appropiate State Variables and Parameters
oscillator_model = ModelOscillator(
    state_variables=state_vars,
    parameters=params,
    t_span=t_span,
    t_steps=t_steps
)
```

5. At this point you can play with the IVP itself. For example, you can integrate the equations using the method `run_model()`. The resulting object is the same as the return value of scipy.integrate.solve_ivp

```
# Run the model
oscillator_solution = oscillator_model.run_model()
```

6. Now we will build our *dinjo.optimizer.Optimizer* instance. Ideally you may have some experimental (reference) data to use here. But as we do not, lets just generate some noisy data from the previous exact solution and use it to mock the experimental data. At this point we need to tell the optimizer what state variable corresponds to the observation data and also the times associated to the reference values.

```
from dinjo.optimizer import Optimizer

fake_data_noise_factor = 0.3

# Build fake observation data from the solution
oscillator_fake_position_data = (
    oscillator_solution.y[0]
    + (2 * np.random.random(t_steps) - 1) * fake_data_noise_factor
)
```

(continues on next page)

(continued from previous page)

```
# Instantiate Optimizer using your data
oscillator_optimizer = Optimizer(
    model=oscillator_model,
    reference_state_variable=q,
    reference_values=oscillator_fake_position_data,
    reference_t_values=oscillator_solution.t
)
```

7. Finally we can find the value of ω that best fits to the solution of the IVP by using the *dinjo.optimizer.Optimizer.optimize()* method.

```
minimization_algorithm = 'differential_evolution'

# Optimize parameters
oscillator_parameters_optimization = \
    oscillator_optimizer.optimize(algorithm=minimization_algorithm)

# The attribute oscillator_parameters_optimization.x contains the
# optimal parameters
print(f'Optimal value of $\omega$ = {oscillator_parameters_optimization.x[0]}')
```

8. That's it, but just for fun, lets plot the optimal solution

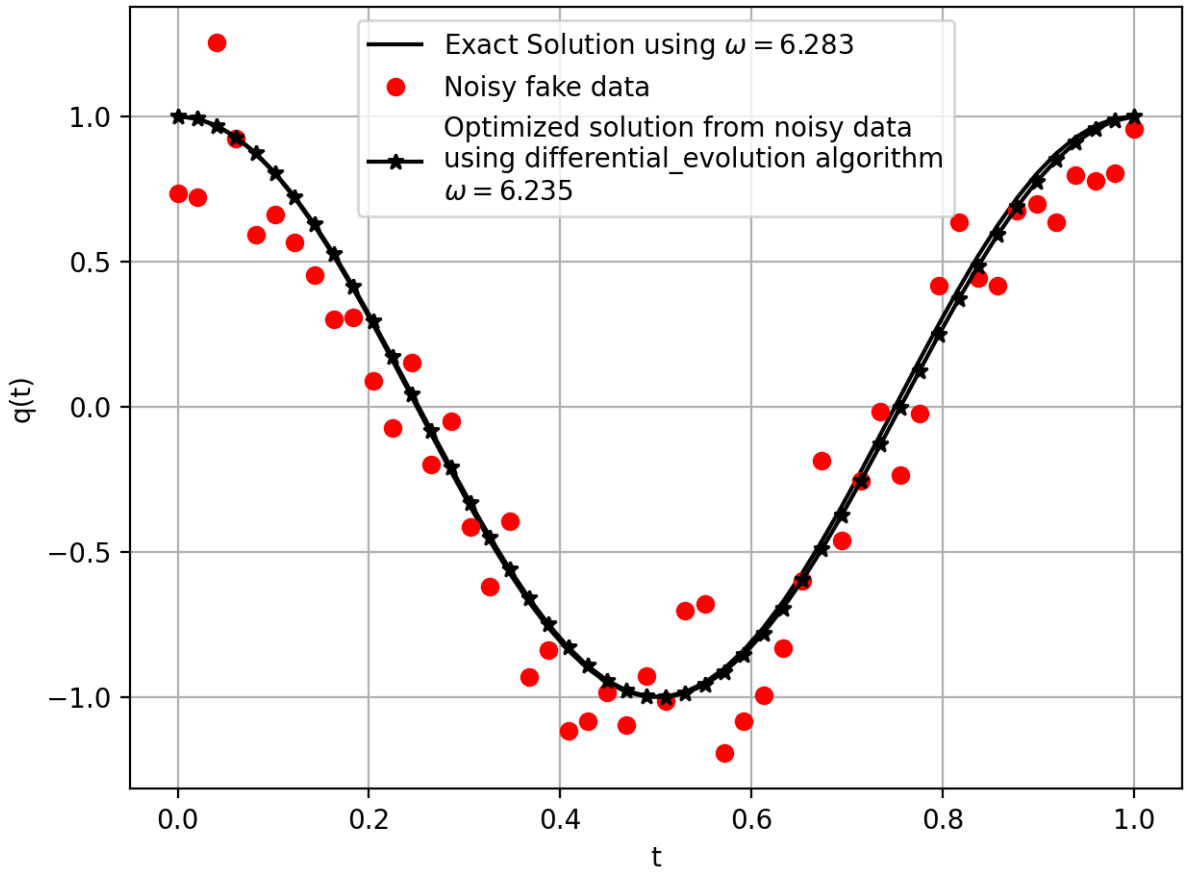
```
import matplotlib.pyplot as plt

oscillator_optimal_solution = oscillator_model.run_model(
    parameters=oscillator_parameters_optimization.x
)

plt.figure()
plt.plot(
    oscillator_solution.t, oscillator_solution.y[0],
    'k-',
    label='Exact Solution using '
          f'$\omega={omega.initial_value:.3f}$',
)
```

(continues on next page)

(continued from previous page)

```
plt.plot(
    oscillator_solution.t, oscillator_fake_position_data,
    'ro', label='Noisy fake data'
)
plt.plot(
    oscillator_optimal_solution.t, oscillator_optimal_solution.y[0],
    'k-*',
    label='Optimized solution from noisy data\n'
          f'using {minimization_algorithm} algorithm\n'
          f'$\omega={oscillator_parameters_optimization.x[0]:.3f}$',
)
plt.xlabel('t')
plt.ylabel('q(t)')
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
plt.close()
```

The plot you get should look similar to the following



The example is complete, should look like the following and should run as it is given that you have previously installed `dinjo`, `numpy` and `matplotlib`

```
from dinjo.model import ModelIVP, StateVariable, Parameter
from dinjo.optimizer import Optimizer

import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
# Define the IVP
class ModelOscillator(ModelIVP):
    def build_model(self, t, y, w):
        """Harmonic Oscillator differential equations
        """
        q, p = y

        # Hamilton's equations
        dydt = [
            p,
            - (w ** 2) * q
        ]

        return dydt


# Define State Variables
q = StateVariable(
    name='position', representation='q', initial_value=1.0
)
p = StateVariable(
    name='momentum', representation='p', initial_value=0.0
)

# Define Paramters
omega = Parameter(
    name='frequency', representation='w',
    initial_value=2 * np.pi, bounds=[4, 8]
)

state_vars = [q, p]
params = [omega]
```

(continues on next page)

(continued from previous page)

```
t_span = [0, 1]
t_steps = 50

# Instantiate the IVP class with appropiate State Variables and Parameters
oscillator_model = ModelOscillator(
    state_variables=state_vars,
    parameters=params,
    t_span=t_span,
    t_steps=t_steps
)

# Run the model
oscillator_solution = oscillator_model.run_model()

fake_data_noise_factor = 0.3

# Build fake observation data from the solution
oscillator_fake_position_data = (
    oscillator_solution.y[0]
    + (2 * np.random.random(t_steps) - 1) * fake_data_noise_factor
)

# Instantiate Optimizer using your data
oscillator_optimizer = Optimizer(
    model=oscillator_model,
    reference_state_variable=q,
    reference_values=oscillator_fake_position_data,
    reference_t_values=oscillator_solution.t
)

minimization_algorithm = 'differential_evolution'

# Optimize parameters
oscillator_parameters_optimization = \
    oscillator_optimizer.optimize(algorithm=minimization_algorithm)
```

(continues on next page)

(continued from previous page)

```
# The attribute oscillator_parameters_optimization.x contains the
# optimal parameters
print(f'Optimal value of $\omega$ = {oscillator_parameters_optimization.x[0]}')

# Plot solution
oscillator_optimal_solution = oscillator_model.run_model(
    parameters=oscillator_parameters_optimization.x
)

plt.figure()
plt.plot(
    oscillator_solution.t, oscillator_solution.y[0],
    'k-',
    label='Exact Solution using '
          f'$\omega={omega.initial_value:.3f}$',
)
plt.plot(
    oscillator_solution.t, oscillator_fake_position_data,
    'ro', label='Noisy fake data'
)
plt.plot(
    oscillator_optimal_solution.t, oscillator_optimal_solution.y[0],
    'k-*',
    label='Optimized solution from noisy data\n'
          f'using {minimization_algorithm} algorithm\n'
          f'$\omega={oscillator_parameters_optimization.x[0]:.3f}$',
)
plt.xlabel('t')
plt.ylabel('q(t)')
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
plt.close()
```

- *Examples*

Install DINJO using PyPI:

```
pip install dinjo
```

Or directly from the latest dev versioN, using source code:

```
git clone https://github.com/fenfisdi/dinjo
cd dinjo
python setup.py install
```

Start using DINJO!

CHAPTER

TWO

THE SOURCE CODE

2.1 dinjo package

2.1.1 Package contents

2.1.2 Subpackages

2.1.2.1 dinjo.predefined subpackage

2.1.2.1.1 Package contents

2.1.2.1.2 Subpackages

2.1.2.1.2.1 dinjo.predefined.epidemiology subpackage

For your convenience, the classes *ModelSEIR*, *ModelSEIRV*, *ModelSimpleSEIRV*, *ModelSIR* can be imported directly from the subpackage dinjo.predefined.epidemiology:

```
from dinjo.predefined.epidemiology import (
   ModelSEIR,
   ModelSEIRV,
   ModelSimpleSEIRV,
```

(continues on next page)

(continued from previous page)

```
    ModelSIR
)
```

2.1.2.1.2.2 Submodules

2.1.2.1.2.3 dinjo.predefined.epidemiology._seir_model

class dinjo.predefined.epidemiology._seir_model.**ModelSEIR**(*state_variables: List[*dinjo.model.StateVariable*]*, *parameters: List[*dinjo.model.Parameter*]*, *t_span: Optional[List[float]] = None*, *t_steps: int = 50*, *t_eval: Optional[List[float]] = None*)

Bases: *dinjo.model.ModelIVP*

build_model(*t*, *y*, *Lmbd*, *mu*, *omega*, *gamma*, *inv_alpha*, *chi*, *beta_E*, *beta_I*) → List[float]

Returns the vector field dy/dt evaluated at a given point in phase space

2.1.2.1.2.4 dinjo.predefined.epidemiology._seirv_model

class dinjo.predefined.epidemiology._seirv_model.**ModelSEIRV**(*state_variables: List[*dinjo.model.StateVariable*]*, *parameters: List[*dinjo.model.Parameter*]*, *t_span: Optional[List[float]] = None*, *t_steps: int = 50*, *t_eval: Optional[List[float]] = None*)

Bases: *dinjo.model.ModelIVP*

build_model(*t*, *y*, *Lmbd*, *mu*, *inv_alpha*, *omega*, *gamma*, *xi_E*, *xi_I*, *sigma*, *beta_E*, *beta_I*, *beta_V*, *c_E*, *c_I*, *c_V*) → List[float]

Returns the vector field dy/dt evaluated at a given point in phase space

2.1.2.1.2.5 dinjo.predefined.epidemiology._seirv_fixed

class dinjo.predefined.epidemiology._seirv_fixed.**ModelSimpleSEIRV**(*state_variables: List[*dinjo.model.StateVariable*]*, *parameters: List[*dinjo.model.Parameter*]*, *t_span: Optional[List[float]] = None*, *t_steps: int = 50*, *t_eval: Optional[List[float]] = None*)

Bases: *dinjo.model.ModelIVP*

build_model(*t*, *y*, *Lmbd*, *mu*, *omega*, *gamma*, *inv_alpha*, *xi_E*, *xi_I*, *sigma*, *beta_E*, *beta_I*, *beta_V*) → List[float]

Returns the vector field dy/dt evaluated at a given point in phase space

2.1.2.1.2.6 dinjo.predefined.epidemiology._sir_model

class dinjo.predefined.epidemiology._sir_model.**ModelSIR**(*state_variables: List[*dinjo.model.StateVariable*]*, *parameters: List[*dinjo.model.Parameter*]*, *t_span: Optional[List[float]] = None*, *t_steps: int = 50*, *t_eval: Optional[List[float]] = None*)

Bases: *dinjo.model.ModelIVP*

build_model(*t*, *y*, *Lmbd*, *mu*, *omega*, *gamma*, *chi*, *eta*, *Pi*, *tau*) → List[float]

Returns the vector field dy/dt evaluated at a given point in phase space

2.1.3 Submodules

2.1.3.1 dinjo.model module

class dinjo.model.**ModelIVP**(*state_variables: List[*dinjo.model.StateVariable*]*, *parameters: List[*dinjo.model.Parameter*]*, *t_span: Optional[List[float]] = None*, *t_steps: int = 50*, *t_eval: Optional[List[float]] = None*)

Bases: `object`

Defines and integrates an initial value problem.

state_variables

Type list[*StateVariable*]

parameters

Type list[*Parameter*]

t_span

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

Type 2-tuple of floats

t_steps

The solver will get the solution for `t_steps` equally separated times from `t0` to `tf`.

Type int

t_span

List containing the time values in which the user wants to evaluate the solution. All the values must be within the interval defined by `t_span`.

Type list[float]

__init__(*state_variables: List[dinjo.model.StateVariable]*, *parameters: List[dinjo.model.Parameter]*, *t_span: Optional[List[float]] = None*, *t_steps: int = 50*, *t_eval: Optional[List[float]] = None*) → None
Initialize self. See help(type(self)) for accurate signature.

build_model(*t*, *y*, **args*)
Defines the differntial equations of the model.

Override this method so that it contains the differential equations of your IVP. The signature of the method must be `build_model(self, t, y, *args)` where `t` is the time, `y` is the state vector, and `args` are other parameters of the system.

Parameters

- **t** (*float*) – time at which the differential equation must be evaluated.
- **y** (*list[float]*) – state vector at which the differential must be evaluated.
- ***args** (*any*) – other parameters of the differential equation

Returns

- *The method must return the time derivative of the*
- *state vector evaluated at a given time.*

Note: The parameters must be defined in the same order in which the parameters are stored in *ModelIVP.parameters*.

Note: The state variable vector must be defined in the same order in which the state variables are stored in *ModelIVP.state_variables*.

Example

For example if you want to simulate a harmonic oscillator of frequency ω and mass m this method must be implemented as follows:

```
def build_model(self, t, y, w, m):
    q, p = y

    # Hamilton's equations
    dydt = [
        p,                      # dq/dt
```

(continues on next page)

(continued from previous page)

```
        - (w ** 2) * q          # dp/dt
    ]

    return dydt
```

property parameters_init_vals
Get the values of the model's parameters initial values in theorder they are currently stored in `self.parameters`.

run_model(*parameters: Optional[List[float]] = None*, *method: str = 'RK45'*)
Integrate model using `scipy.integrate.solve_ivp`

Parameters

- **parameters** (*list[float]*) – List of the values of the paramters of the initial value problem.
- **method** (*srt*) – Integration method. Must be one of the methods accepted by `scipy.integrate.solve_ivp`

Returns

- *Bunch object with the following fields defined (same return type*
- *as scipy.integrate.solve_ivp)*
- **t** (*ndarray, shape (n_points,)*) – Time points.
- **y** (*ndarray, shape (n, n_points)*) – Values of the solution at t.
- **sol** (*OdeSolution or None*) – Found solution as OdeSolution instance; None if dense_output was set to False.
- **t_events** (*list of ndarray or None*) – Contains for each event type a list of arrays at which an event of that type event was detected. None if events was None.
- **y_events** (*list of ndarray or None*) – For each value of t_events, the corresponding value of the solution. None if events was None.
- **nfev** (*int*) – Number of evaluations of the right-hand side.
- **njev** (*int*) – Number of evaluations of the Jacobian.
- **nlu** (*int*) – Number of LU decompositions.
- **status** (*int*) – Reason for algorithm termination:

property state_variables_init_vals: List[float]
Get the values of the model's state variables initial values in the order they are currently stored in `self.state_variables`.

class dinjo.model.**Parameter**(*name: str*, *representation: str*, *initial_value: float = 0*, *bounds: Optional[List[float]] = None*, **args*, ***kwargs*)
Bases: *dinjo.model.Variable*

Represents a parameter of the differential equations definining an initial value problem.

In addition to the attributes defined in *dinjo.model.Variable*

bounds
list containing the minimum and maximum values that the parameter can take `(min, max)`.

Type 2-tuple of floats.

__init__(*name: str*, *representation: str*, *initial_value: float = 0*, *bounds: Optional[List[float]] = None*, **args*, ***kwargs*) → None
Initialize self. See help(type(self)) for accurate signature.

property bounds

class dinjo.model.**StateVariable**(*name: str*, *representation: str*, *initial_value: float = 0*, **args*, ***kwargs*)
Bases: *dinjo.model.Variable*

Represents a State Variable of an initial value problem.

class dinjo.model.**Variable**(*name: str*, *representation: str*, *initial_value: float = 0*, **args*, ***kwargs*)
Bases: `object`

Represents a variable

name
name of the variable.

Type str

representation
string representing the state variable (could be the same as name.)

Type str

initial_value
reference value of the variable being represented.

Type float

__init__(*name: str*, *representation: str*, *initial_value: float = 0*, **args*, ***kwargs*) → None
Initialize self. See help(type(self)) for accurate signature.

2.1.3.2 dinjo.optimizer module

class dinjo.optimizer.**Optimizer**(*model:* dinjo.model.ModelIVP, *reference_state_variable:* dinjo.model.StateVariable, *reference_values: List[float]*, *reference_t_values: List[float]*, *integration_method: str = 'RK45'*)

Bases: object

Optimizes the initial value problem's parameters, as defined in the class ModelIVP.

model
the initial value problem to be optimized.

Type ModelIVP

reference_state_variable :class:`StateVariable`
the state variable to be fitted to the solution of the IVP.

reference_values
the 'experimental data' of the reference state variable to be used as the fitting variable.

Type list[float]

reference_t_values
the corresponding times at which the reference_values are given.

Type list[float]

integration_method
must be one of the methods accepted by scipy.integrate.solve_ivp.

Type str

__init__(*model:* dinjo.model.ModelIVP, *reference_state_variable:* dinjo.model.StateVariable, *reference_values: List[float]*, *reference_t_values: List[float]*, *integration_method: str = 'RK45'*) → None
Initialize self. See help(type(self)) for accurate signature.

cost_function(*parameters: List[float]*, *cost_method: str = 'root_mean_square'*)
Function to be minimized by the optimizer. Initially this will be the root mean square of the difference between the observations and the numerical solution.

Parameters

- **parameters** (*list[float]*) – parameters of the model to be minimized. The order of the parameters must be the same as they appear in self.model.parameters.
- **cost_method** (*str*) – Must be one of ['root_mean_square',].

optimize(*cost_method: str = 'root_mean_square'*, *algorithm: str = 'differential_evolution'*, *algorithm_kwargs: Dict[str, Any] = {}*) → scipy.optimize.optimize.OptimizeResult

Global minimization of cost function.

Parameters

- **cost_function_method** (*str*) – Must be one of the permitted values for cost_method parameter in `Optimize.cost_function()`.
- **algorithm** (*str*) – scipy.optimize algorithm used for the optimization. Must be one of `'differential_evolution'`, `'shgo'`, `'dual_annealing'`.
- **algorithm_kwargs** (*dict[str, any]*) – parameters passed to the optimization algorithm. They are different depending on the algorithm.

Returns object containing the minimization result as returned by the appropiate scipy algorithm (e.g. if the chosen al).

Return type minimization

property reference_t_values

dinjo Package source code

dinjo.model Define your own initial value problems (IVPs) and solve them using this module.

dinjo.optimizer Define your IVP optimization problem and solve it using this module.

dinjo.predefined Some predefined models

dinjo.predefined.epidemiology._seir_model SEIR initial value problem.

dinjo.predefined.epidemiology._seirv_model SEIRV initial value problem.

dinjo.predefined.epidemiology._seirv_fixed SEIR initial value problem.

dinjo.predefined.epidemiology._sir_model SIR initial value problem.

dinjo.predefined.physics._harmonic_oscillator Unit mass Harmonic Oscillator initial value problem. See *Examples*.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

B

C

D

I

M

N

O

P

R

S

T

V