PHYS Simulation API

Release 1.0

Juan E. Aristizabal

Oct 22, 2020

GETTING STARTED

Using PHYS Simulation API you can request simulations of some physical or mathematical systems such as the Harmonic Oscillator or the Chen-Lee Attractor –and soon many others, stay tuned!

CHAPTER ONE

HISTORY

PHYS Simulation API started as a way of learning to develop an API. We believe that the best way of learning is by doing. This way, we chose a topic that drives us (yes, physics and mathematics!) and started learning by writting code! Of course, we had to read a lot and fail a lot as well, but we managed to develop our very first web application and we are proud of it, as impefect as it may be.

In the future we want to add as many features as we can, so far it enriches the client experience.

Note: This project was also presented as Harvard's CS50x final project. Check out the video.

CHAPTER

TWO

GETTING STARTED

2.1 Overview

In PHYS Simulation API you can request –via HTTP POST method– a simulation from a list of available simulations. Here we will show you the schemas you need to appropiately request your simulation. In *Examples* you will see how this works in the real life.

2.1.1 Request a simulation

The simulation requests are made in route `/api/simulate/{sim_system}`, where `sim_system` is one of the members of `SimSystem`.

The body of the request must abide by the following schema

2.1.1.1 `SimRequest`

<table>
<tr><td colspan="3">Schema needed to request simulations via POST in <code>/api/request/{sim_system}</code>.</td></tr>
<tr><td>type</td><td colspan="2">object</td></tr>
<tr><td colspan="3">properties</td></tr>
<tr><td rowspan="2">• system</td><td colspan="2"><code>SimSystem</code></td></tr>
<tr><td>type</td><td>string</td></tr>
<tr><td rowspan="3">• t_span</td><td colspan="2">T Span</td></tr>
<tr><td>type</td><td>array</td></tr>
<tr><td>default</td><td></td></tr>
</table>

continues on next page

Table 1 – continued from previous page

	items	type	*number*
• t_eval	*T Eval*		
	type	*array*	
	default		
	items	type	*number*
• t_steps	*T Steps*		
	type	*integer*	
	default	0	
• **ini_cndtn**	*Ini Cndtn*		
	type	*array*	
	default		
	items	type	*number*
• **params**	*Params*		
	type	*object*	
	additionalProperties	type	*number*
• method	`IntegrationMethods`		
	type	*string*	
• **username**	*Username*		
	type	*string*	
	default	Pepito Perez	
definitions			
• SimSystem	*SimSystem*		
	List of available systems for simulation.		
	type	*string*	
	enum	Harmonic-Oscillator, Chen-Lee-Attractor	
• IntegrationMethods	*IntegrationMethods*		
	List of available integration methods. For more information see scipy.integrate.solve_ivp.		
	type	*string*	
	enum	RK45, RK23	

The body of the HTML response will have the following schema

2.1.1.2 `SimIdResponse`

Schema for the response of a simulation request (requested via POST in route `/api/simulate/{sim_sys}`.)		
type	*object*	
properties		
• sim_id	*Sim Id*	
	type	*string*
• user_id	*User Id*	
	type	*integer*
• username	*Username*	
	type	*string*
• sim_sys	`SimSystem`	
	type	*string*
• sim_status_path	*Sim Status Path*	
	type	*string*
• sim_pickle_path	*Sim Pickle Path*	
	type	*string*
• message	*Message*	
	type	*string*

2.1.2 Request Simulation Status

The response of the request simulation (*SimIdResponse*) contains a simulation ID, `sim_id`. In order to know the simulation status you just need to make an HTTP request via GET with empty body in route `/api/request/status/{sim_id}`.

Note: The route will be availble just after the simulation is finished. If you do not receive a successful response and you are sure about the `sim_id` you provided in the API route, the simulation may still be in course.

The schema of the response will be the following

2.1.2.1 `SimStatus`

Schema of the Simulation Satus.					
type	*object*				
properties					
• **sim_id**	*Sim Id*				
	type	*string*			
• **user_id**	*User Id*				
	type	*integer*			
• **date**	*Date*				
	type	*string*			
	format	date-time			
• system	`SimSystem`				
	type	*string*			
• ini_cndtn	*Ini Cndtn*				
	type	*array*			
	items	type	*number*		
• params	*Params*				
	type	*object*			
	additionalProperties	type	*number*		
• method	`IntegrationMethods`				
	type	*string*			
• route_pickle	*Route Pickle*				
	type	*string*			
• route_results	*Route Results*				
	type	*string*			
• route_plots	*Route Plots*				
	type	*string*			
• plot_query_values	`PlotQueryValues`				
	type	*array*			
	items	anyOf	`PlotQueryValues_HO`		
			type	*object*	
			additionalProperties	type	*string*
			`PlotQueryValues_ChenLee`		
			type	*object*	

continues on next page

Table 2 – continued from previous page

			additionalProperties	type	*string*
• plot_query_receipe	*Plot Query Receipe*				
	type	*string*			
	default	'route_plots' + '?value=' + 'plot_query_value'			
• success	*Success*				
	type	*boolean*			
• message	*Message*				
	type	*string*			
definitions					
• SimSystem	*SimSystem*				
	List of available systems for simulation.				
	type	*string*			
	enum	Harmonic-Oscillator, Chen-Lee-Attractor			
• IntegrationMethods	*IntegrationMethods*				
	List of available integration methods. For more information see scipy.integrate.solve_ivp.				
	type	*string*			
	enum	RK45, RK23			
• PlotQueryValues_HO	*PlotQueryValues_HO*				
	List of tags of each different plot generated automatically by the backend when a Harmonic Oscillator simulation is requested.				
	type	*string*			
	enum	coord, phase			
• PlotQueryValues_ChenLee	*PlotQueryValues_ChenLee*				
	List of tags of each different plot generated automatically by the backend when a Chen-Lee simulation is requested.				
	type	*string*			
	enum	threeD, project			

2.1.3 Request Results

If `success=True` in *SimStatus*, you can

1. *Directly download your results*

    a. Download pickle file with all the simulation results as returned by scipy.integrate.solve_ivp. You can do this via GET with empty body in route `/api/results/{sim_id}/pickle`.

    b. Download your simulation's automatically generated plots. You can do this via GET with empty body in route `/api/results/{sim_id}/plot?value={plot_query_value}`, where `plot_query_value` is one of the items in `plot_query_values` in the simulation status, *SimStatus*.

2. *See your results online*. You can do this via GET in route `/results/{sim_system}/{sim_id}`. In the rendered HTML file, you have the option to download both the generated plots and the pickle file mentioned in the last item.

2.2 Install and run PHYS Simulation

1. Before installing PHYS Simulation make sure you install pipenv.
2. To download PHYS Simulation clone it from the github repository using the following command

```
$ git clone https://github.com/jearistiz/phys_simulation
```

3. Change to the directory where PHYS Simulation is installed

```
$ cd phys_simulation
```

4. Create a virtual environment and install all the requirements using `pipenv` (this can take several minutes)

```
$ pipenv install
```

5. Activate the virtual environment

```
$ pipenv shell
```

6. Finally, run the web application in your localhost http://0.0.0.0:5700/

```
$ pipenv run simulation_api
```

You can manage `HOST` and `PORT` variables in the server configuration file `~/phys_simulation/config.py`.

Note:

- The project runs on a Uvicorn server. The file in charge of setting up and starting the server is `~/phys_simulation/run.py`. Change the options to your prefferred ones.
- If you run the server locally you can try the frontend and API in the host and port you chose. The frontend is almost self-explanatory.
- If you want to try out the API, go to the *Examples* section. We also explain how to use the API in general terms in the *Overview* section.
- **The API has its own client documentation**, thanks to FastAPI's integration with Swagger (formerly OpenAPI). To read this docs, after starting the server go to the resource `localhos:5700/docs` or `localhos:5700/redoc` and refer to the documentation of the resources starting with `/api/`.
- If you want to stop the server, go to the terminal where you started it and use the shortcut `Ctrl + C`.
- If you want to remove the virtual environment after using the app move to `~/phys_simulation/` directory and execute

```
$ pipenv --rm
```

2.3 Examples

Here we will see an example of the work flow with PHYS Simulation API. To make HTTP requests we will use the `requests` Python library.

Let's see our API in action!

2.3.1 1. Request Simulation

Lets start with some imports and defining our simulation request body.

```
>>> import sys
>>> import os
>>> from time import sleep
>>>
>>> import requests
>>>
>>> ######################## Example: Simulation Request #########################
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # Prepare the simulation request
>>> sim_system = "Chen-Lee-Attractor"
>>> sim_request = {
>>>     "system": sim_system,
>>>     "username": "PHYSSimulation",
>>>     "t_span": [0, 200],
>>>     "t_steps": 1e5,
>>>     "ini_cndtn": [10, 10, 0],
>>>     "params": {
>>>         "a": 3,
>>>         "b": -5,
>>>         "c": -1,
>>>     }
>>> }
```

Note that `sim_request` follows the pydantic model `simulation_API.controller.schemas.SimRequest`. For the `ChenLeeAttractor`, the initial condition array is of length 3, and the parameters are `a`, `b` and `c`. In your case, the conditions may be different, so you must check what are the parameters and what are the initial conditions for the system you want to simulate.

Now we request the simulation via post to the appropiate route and print the result

```
>>> # PHYS Simulatio URI
>>> url = 'http://0.0.0.0:5700'
>>>
>>> # Simulation request route
>>> sim_requrest_route = f'/api/simulate/{sim_system}'
>>>
>>> # Request simulation via HTTP using `requests` module
>>> sim_request_response = requests.post(url + sim_requrest_route, json=sim_request)
>>>
>>> # Print response
>>> print(
>>>     "",
>>>     "Simulation Request Response",
>>>     "-----------------------------------------",
>>>     f"HTML status code: {sim_request_response.status_code}",
>>>     "Response:",
>>>     "    {",
>>>     sep='\n',
```

(continues on next page)

(continued from previous page)

```
>>> )
>>> for key, v in sim_request_response.json().items():
>>>     print(f"        '{key}': {v}")
>>> print("    }")

Simulation Request Response
----------------------------------------
HTML status code: 200
Response:
    {
        'sim_id': e5f6d0e0719b45fea4aa9f098e12e7c3,
        'user_id': 1,
        'username': PHYSSimulation,
        'sim_sys': Chen-Lee-Attractor,
        'sim_status_path': /api/simulate/status/e5f6d0e0719b45fea4aa9f098e12e7c3,
        'sim_pickle_path': /api/results/e5f6d0e0719b45fea4aa9f098e12e7c3/pickle,
        'message': (When -and if- available) request via GET your simulation'sstatus in route 'sim_status_path' or download your
↪results(pickle fomat) via GET in route 'sim_pickle_path',
    }
```

We have just finished the first step in the workflow. We now know our `sim_id` how to request the simulation results.

2.3.2 2. Request Simulation Status

We proceed now to request the simulation results via GET in route `/api/simulate/status/{sim_id}`.

```
>>> ######################### Example: Simulation Status ##########################
>>>
>>> # Wait until the simulation is done
>>> sleep(5)
>>>
>>> # Get simulation ID
>>> sim_id = sim_request_response.json()["sim_id"]
>>>
>>> # Simulation status route
>>> sim_status_route = f"/api/simulate/status/{sim_id}"
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # Request simulation status via HTTP using `requests` module
>>> sim_status_response = requests.get(url + sim_status_route)
>>>
>>> # Print response
>>> sim_status_response_json = sim_status_response.json()
>>> print(
>>>     "",
>>>     "Simulation status Response",
>>>     "----------------------------------------",
>>>     f"HTML status code: {sim_status_response.status_code}",
>>>     "Response:",
>>>     "    {",
>>>     sep='\n',
>>> )
>>> for key, v in sim_status_response_json.items():
>>>     print(f"        '{key}': {v},")
>>> print("    }\n")

Simulation status Response
----------------------------------------
HTML status code: 200
Response:
    {
        'sim_id': e5f6d0e0719b45fea4aa9f098e12e7c3,
        'user_id': 1,
        'date': 2020-10-05T23:31:24.977484,
        'system': Chen-Lee-Attractor,
        'ini_cndtn': [10.0, 10.0, 0.0],
        'params': {'a': 3.0, 'b': -5.0, 'c': -1.0},
        'method': RK45,
        'route_pickle': /api/results/e5f6d0e0719b45fea4aa9f098e12e7c3/pickle,
        'route_results': /api/simulate/status/e5f6d0e0719b45fea4aa9f098e12e7c3,
        'route_plots': /api/results/e5f6d0e0719b45fea4aa9f098e12e7c3/plot,
        'plot_query_values': ['threeD', 'project'],
        'plot_query_receipe': 'route_plots' + '?value=' + 'plot_query_value',
        'success': True,
        'message': Finished. You can request via GET: download simulation results (pickle) in route given in 'route_pickle', or;␣
↪download plots of simulation in route 'route_plots' using query params the ones given in 'plot_query_values', or; see results␣
↪online in route 'route_results'.,
```

(continues on next page)

(continued from previous page)

```
    }
```

Note the `sleep(5)` call at the beginning of this code block. We did this in order to be sure that the simulation was completed. If we request the simulation status too soon, it may not be available and an appropiate `message` will be returned in the response as well as `"success": False`.

Also note that in the `"message"` it is clearly stated how to access the results.

2.3.3 3. Request Simulation Results

Now we proceed to download the pickle and the plots, as stated in the simulation status `"message"`.

Lets start with the pickle

```
>>> ############################ Example: GET Results ##############################
>>>
>>> if not sim_status_response_json["success"]:
>>>     print("Warning: pickle and plot files not available.\n")
>>>     sys.exit(1)
>>>
>>> # Pickle download route
>>> pickle_route = sim_status_response.json()["route_pickle"]
>>>
>>> # Request simulation status via HTTP using `requests` module
>>> pickle_response = requests.get(url + pickle_route, stream=True)
>>>
>>> # Get our directory absolute path
>>> this_directory = os.path.dirname(__file__)
>>>
>>> # Save Pickle
>>> with open(this_directory + '/simulation.pickle', 'wb') as file:
>>>     file.write(pickle_response.content)
```

The generated file is named `simulation.pickle` and contains all the information of the simulation results.

Finally, lets download the plots

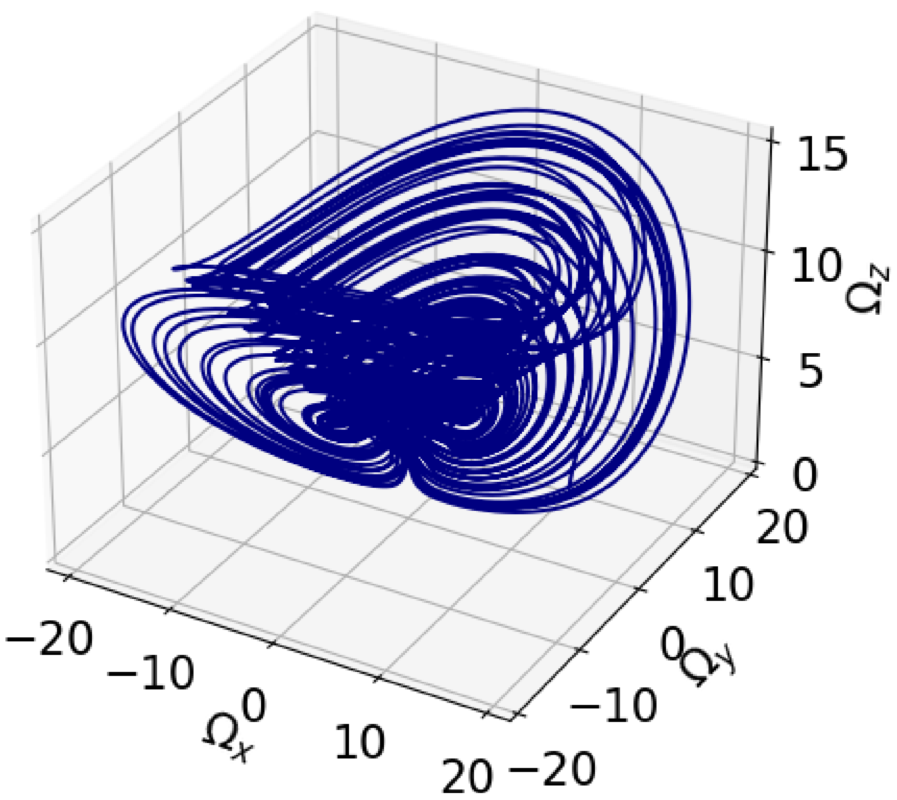
```
>>> # Plots download route
>>> plots_route = sim_status_response.json()["route_plots"]
```

(continues on next page)

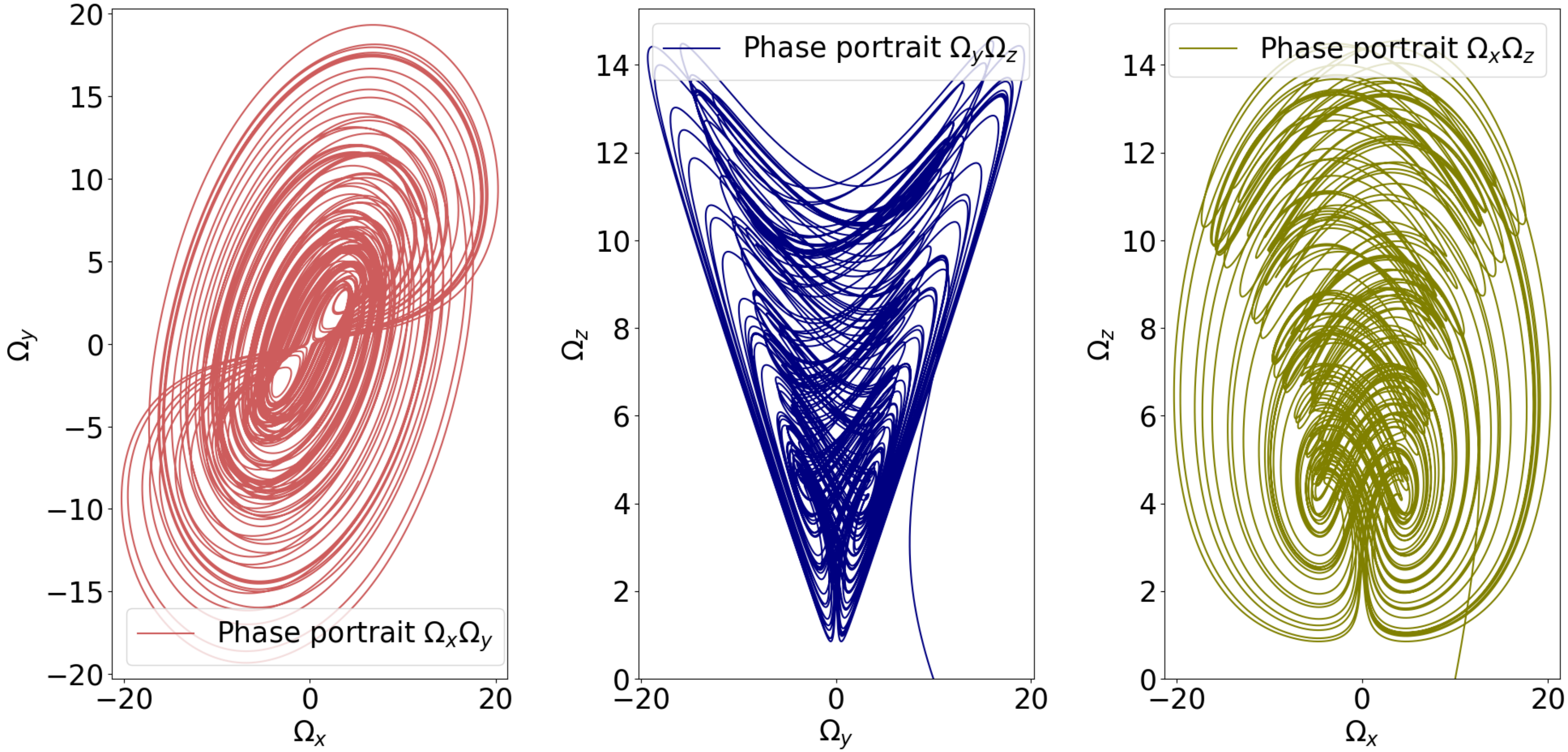
(continued from previous page)

```
>>>
>>> # Plot query values
>>> plot_query_values = sim_status_response.json()["plot_query_values"]
>>>
>>> for qv in plot_query_values:
>>>     # Construct the URI query for each plot
>>>     plot_query_url = url + plots_route + "?value=" + qv
>>>
>>>     # Request the plot
>>>     plot_response = requests.get(plot_query_url, stream=True)
>>>
>>>     # Save the plot in a file
>>>     plot_file_name = this_directory + "/plot_" + qv + ".png"
>>>     with open(plot_file_name, 'wb') as file:
>>>         file.write(plot_response.content)
```

The two generated plots are named `plot_threeD.png` and `plot_project.png` and are respectively displayed below.



15
10
5
0
Ω_z
20
10
0
Ω_y
−10
−20
−20
−10
0
Ω_x
10
20



Phase portrait $\Omega_x\Omega_y$
Ω_y
Ω_x
20
15
10
5
0
−5
−10
−15
−20
Phase portrait $\Omega_y\Omega_z$
Ω_z
Ω_y
14
12
10
8
6
4
2
0
Phase portrait $\Omega_x\Omega_z$
Ω_z
Ω_x

2.4 Website

This API has a very simple frontend, where you can request your simulations and see all the available results, as well as some pretty plots of each simulation.

Right now we are not serving online, but plan to do so some day. If you want to run the frontend and API locally in your machine please go to the section *Install and run PHYS Simulation*.

- *Overview*
- *Install and run locally*
- *Examples*
- *Website*

CHAPTER

THREE

FOR DEVELOPERS

This API is built on top of FastAPI. If you want to contribute to this project, you may find their docs very useful.

In this section you can find all the source code docuentation. If you have a question on how some function, class or method works, maybe it's answered here. If not, do not hesitate to contact us.

3.1 The Code

3.1.1 simulation_api Package

3.1.1.1 Package contents

This module initializes our application.

3.1.1.2 Subpackages

3.1.1.2.1 `simulation_api.controller`

3.1.1.2.1.1 Package contents

3.1.1.2.1.2 Submodules

3.1.1.2.1.3 `simulation_api.controller.main`

This file manages all requests that are made to our app

async `simulation_api.controller.main.`**`api_results_sim_id_pickle`**(*sim_id: str*)

Download pickle of previously requested simulation.

Parameters **`sim_id`** (*`str`*) – ID of the simulation.

Returns `FileResponse` containing the simulation results in pickle format.

Return type starlette.responses.FileResponse

async `simulation_api.controller.main.`**`api_results_sim_id_plot`**(*sim_id: str*, *value: Union[*simulation_api.controller.schemas.PlotQueryValues_HO, simulation_api.controller.schemas.PlotQueryValues_ChenLee*]*)

Download plot of previously requested simulation.

Note one query param is required here.

Here we use FileResponse from starlette.responses

Parameters

- **`sim_id`** (*`str`*) – ID of the simulation.
- **`value`** (*`PlotQueryValues`*) – Query values. Must be a member of one of the Enum classes given in PlotQueryValues.

Returns `FileResponse` containing the requested plot.

Return type starlette.responses.FileResponse

async `simulation_api.controller.main.`**`api_simulate_sim_system`**(*sim_system:* simulation_api.controller.schemas.SimSystem, *sim_params:* simulation_api.controller.schemas.SimRequest, *background_tasks: starlette.background.BackgroundTasks*, *db: sqlalchemy.orm.session.Session = Depends(get_db)*) → *simulation_api.controller.schemas.SimIdResponse*

In this route the client can request a simulation.

When the client requests a simulation via `/api/simulate/{sim_system}`, he/she obtains relevant information on how to get the results of the simulation.

Note: Here we use `fastapi.BackgroudTasks` to make the simulation in the background.

Parameters

- **sim_system** (`SimSystem`) – System to be simulated.
- **sim_params** (`SimRequest`) – Simulation request information with schema given by SimRequest and decalared in schemas.py.
- **background_tasks** (*BackgroundTasks*) – Background task FastAPI manager (Class). This is handled internally.
- **db** (*Session*) – Database Session, needed to interact with database. This is handled internally.

Returns **sim_id_response** – Contains relevant information about the simulation and how to get the results.

Return type *SimIdResponse*

async `simulation_api.controller.main.`**`api_simulate_status_sim_id`**(*sim_id: str*, *db: sqlalchemy.orm.session.Session = Depends(get_db)*) → *simulation_api.controller.schemas.SimStatus*

Obtains status of requested simulation.

Parameters

- **sim_id** (*str*) – ID of the simulation.
- **db** (*Session*) – Database Session, needed to interact with database. This is handled internally.

Returns Status information of the simulation and how to get the results.

Return type *SimStatus*

async `simulation_api.controller.main.`**`custom_http_exception_handler`**(*request: starlette.requests.Request*, *exc: starlette.exceptions.HTTPException*)

Handles 404 exceptions by rendering a template.

`simulation_api.controller.main.`**`get_db`**()

Starts and ends session in each route that needs database access

async `simulation_api.controller.main.`**`index`**(*request: starlette.requests.Request*)

Index frontend web page.

Parameters **request** (*Request*) – HTTP request, used internally by FastAPI.

Returns Renders greeting template and hyperlinks to simulation rquests and results.

Return type `fastapi.templating.Jinja2Templates.TemplateResponse`

async `simulation_api.controller.main.`**results**(*request: starlette.requests.Request*, *db: sqlalchemy.orm.session.Session = Depends(get_db)*)

Renders web page showing a list of all the available simulation results.

Parameters

- **request** (*Request*) – HTTP request, used internally by FastAPI.
- **db** (*Session*) – Database Session, needed to interact with database. This is handled internally.

Returns Template displaying all the available simulation results.

Return type `fastapi.templating.Jinja2Templates.TemplateResponse`

async `simulation_api.controller.main.`**results_sim_system_sim_id**(*request: starlette.requests.Request*, *sim_system: simulation_api.controller.schemas.SimSystem*, *sim_id: str*)

Shows graphic results of a simulation in frontend for a given `sim_id`.

Parameters

- **request** (*Request*) – HTTP request, used internally by FastAPI.
- **sim_system** (`SimSystem`) – System to be simulated.
- **sim_id** (*str*) – ID of the simulation.

Returns Template displaying all the generated plots for a given simulation.

Return type `fastapi.templating.Jinja2Templates.TemplateResponse`

async `simulation_api.controller.main.`**simulate**(*request: starlette.requests.Request*)

Simulate web page.

Here the clients can select between the available systems to simulate their preferred one.

Parameters **request** (*Request*) – HTTP request, used internally by FastAPI.

Returns Template displaying the available systems to be simulated.

Return type `fastapi.templating.Jinja2Templates.TemplateResponse`

async `simulation_api.controller.main.`**simulate_id_sim_id**(*request: starlette.requests.Request*, *sim_id: str*)

Shows simulation id after asking for simulation in frontend form.

Parameters

- **request** (*Request*) – HTTP request, used internally by FastAPI.
- **sim_id** (*str*) – ID of the simulation.

Returns Template displaying the simulation ID and a hyperlink to further information about the simulation.

Return type `fastapi.templating.Jinja2Templates.TemplateResponse`

async `simulation_api.controller.main.`**simulate_sim_system**(*request: starlette.requests.Request*, *sim_system:* simulation_api.controller.schemas.SimSystem, *error_message: Optional[str] = ''*)

Simulation's form web page.

The clients input the desired parameters for the simulation of their choosing.

Parameters

- **request** (*Request*) – HTTP request, used internally by FastAPI.
- **sim_system** (`SimSystem`) – System to be simulated.
- **error_message** (*str*) – Internally used by the backend to display error messages in forntend form.

Returns Template displaying the simulation request form.

Return type `fastapi.templating.Jinja2Templates.TemplateResponse`

async `simulation_api.controller.main.`**simulate_sim_system_post**(*request: starlette.requests.Request*, *sim_system:* simulation_api.controller.schemas.SimSystem, *background_tasks: starlette.background.BackgroundTasks*, *db: sqlalchemy.orm.session.Session = Depends(get_db)*, *sim_sys:* simulation_api.controller.schemas.SimSystem *= Form(Ellipsis)*, *username: str = Form(Ellipsis)*, *t0: float = Form(Ellipsis)*, *tf: float = Form(Ellipsis)*, *dt: float = Form(Ellipsis)*, *t_steps: int = Form(Ellipsis)*, *method:* simulation_api.controller.schemas.IntegrationMethods = *Form(Ellipsis)*)

Receives the simulation request information from the frontend form and requests the simulation to the backend.

This route receives the form requesting a simulation (filled in frontend via GET in route `/simulate/{sim_system}`). The simulation is internally requested using the function *`simulation_api.controller.tasks._api_simulation_request()`*. Finally the client is redirected to the "Simulation ID" frontend web page, where further information about the simulation is displayed.

Parameters

- **request** (*Request*) – HTTP request, used internally by FastAPI.

- **sim_system** (`SimSystem`) – System to be simulated.
- **background_tasks** (*BackgroundTasks*) – Needed to request simulation to the backend (in background). Handled internally by the API.
- **db** (*Session*) – Database Session, needed to interact with database. This is handled internally.
- **sim_sys** (`SimSystem`) – Form entry: system to be simulated. Must be one of the members of SimSystem.
- **username** (*str*) – Form entry: name of the user requesting the simulation.
- **t0** (*float*) – Form entry: initial time of simulation.
- **tf** (*float*) – Form entry: final time of simulation.
- **dt** (*float*) – Form entry: timestep of simulation.
- **t_steps** – Form entry: number of time steps. If different from 0 or `None`, `dt` is ignored.
- **method** (`IntegrationMethods`) – Form entry: method of integration. Must be a member of IntegrationMethods.

Returns If the client made a mistake filling the form renders the simulation request form again. Otherwise, redirects the client to "Simulation ID" frontend web page.

Return type `fastapi.templating.Jinja2Templates.TemplateResponse` or `starlette.responses.RedirectResponse`

Note: The values of the form accessed by `fastapi.Form` are only declared as parameters so that pydantic checks the types, but they are not used directly to request the simulation. Here, we access the form directly by using the method `fastapi.Request.form()` as can be seen in the first lines of this function. This allows us a better control over the data and also to handle different type of forms –which depend on the simulation because parameters and initial conditions are intrinsically different for different systems.

async `simulation_api.controller.main.`**`simulate_status_sim_id`**(*request: starlette.requests.Request*, *sim_id: str*, *db: sqlalchemy.orm.session.Session = Depends(get_db)*)

Shows simulation status for a given simulation via its `sim_id`.

Parameters

- **request** (*Request*) – HTTP request, used internally by FastAPI.
- **sim_id** (*str*) – ID of the simulation.
- **db** (*Session*) – Database Session, needed to interact with database. This is handled internally.

Returns Template displaying the simulation status and a hyperlinks to simulation results in several formats. If simulation id is not available (not yet in database), renders a message about the situation.

Return type `fastapi.templating.Jinja2Templates.TemplateResponse`

3.1.1.2.1.4 `simulation_api.controller.schemas`

This module defines all the models/schemas needed in our application, except the database models required for sqlalchemy ORM.

class `simulation_api.controller.schemas.`**`ChenLeeParams`**(*, *a: float*, *b: float*, *c: float*)

Bases: `pydantic.main.BaseModel`

List of parameters of the Chen-Lee Attractor system.

Note: For more information about Chen-Lee Attactor's parameters see *`ChenLeeAttractor`*.

Warning: This needs update each time a new simulation is added: add an appropiate new class similar to this one or to *`HOParams`*.

a: float

ω_x parameter.

b: float

ω_y parameter.

c: float

ω_z parameter.

class `simulation_api.controller.schemas.`**`ChenLeeSimForm`**(*, *username: str = 'Pepito'*, *t0: float = 0.0*, *tf: float = 6.283185307179586*, *dt: float = 0.3141592653589793*, *t_steps: int = 0*, *method: simulation_api.controller.schemas.IntegrationMethods = 'RK45'*, *sim_sys: simulation_api.controller.schemas.SimSystem = 'Chen-Lee-Attractor'*, *ini0: float = 10.0*, *ini1: float = 10.0*, *ini2: float = 0.0*, *param0: float = 3.0*, *param1: float = - 5.0*, *param2: float = - 1.0*)

Bases: *`simulation_api.controller.schemas.SimForm`*

Schema used to Request Chen Lee Simulation in Frontend via form.

Note: For more information about Chen-Lee Attactor simulation see *`ChenLeeAttractor`*.

ini0: Optional[float]
ω_x initial condition.

ini1: Optional[float]
ω_y initial condition.

ini2: Optional[float]
ω_z initial condition.

param0: Optional[float]
Parameter of name *ChenLeeParams.a*.

param1: Optional[float]
Parameter of name *ChenLeeParams.b*.

param2: Optional[float]
Parameter of name *ChenLeeParams.c*.

sim_sys: *simulation_api.controller.schemas.SimSystem*
System to be simulated.

class simulation_api.controller.schemas.**HOParams**(*, *m: float*, *k: float*)
Bases: pydantic.main.BaseModel

List of parameters of the Harmonic Oscillator system.

Note: For more information about Harmonic Oscillator's parameters see *HarmonicOsc1D*.

Warning: This needs update each time a new simulation is added: add an appropiate new class similar to this one or to *ChenLeeParams*.

k: float
Force constant of object.

m: float
Mass of object.

class simulation_api.controller.schemas.**HOSimForm**(**, username: str = 'Pepito', t0: float = 0.0, tf: float = 6.283185307179586, dt: float = 0.3141592653589793, t_steps: int = 0, method: simulation_api.controller.schemas.IntegrationMethods = 'RK45', sim_sys: simulation_api.controller.schemas.SimSystem = 'Harmonic-Oscillator', ini0: float = 1.0, ini1: float = 0.0, param0: float = 1.0, param1: float = 1.0*)

Bases: *simulation_api.controller.schemas.SimForm*

Schema used to Request Harmonic Oscillator Simulation in Frontend via form.

Note: For more information about Chen-Lee Attractor simulation see *HarmonicOsc1D*.

ini0: Optional[float]
q initial value.

ini1: Optional[float]
p initial value.

param0: Optional[float]
Parameter of name *HOParams.m*.

param1: Optional[float]
Parameter of name *HOParams.k*.

sim_sys: *simulation_api.controller.schemas.SimSystem*
System to be simulated.

class simulation_api.controller.schemas.**IntegrationMethods**(*value*)

Bases: str, enum.Enum

List of available integration methods

Note: For more information about these integration methods see scipy.integrate.solve_ivp.

Warning: Please update this class with relvant simulation methods available in scipy.integrate.solve_ivp –only the ones that do not require more parameters than the ones provided in *SimRequest*.

RK23 = 'RK23'
Explicit Runge-Kutta method of order 3(2).

RK45 = 'RK45'
Explicit Runge-Kutta method of order 5(4).

class simulation_api.controller.schemas.**IntegrationMethodsFrontend**(*value*)
Bases: `str`, `enum.Enum`

List of captions of available integration methods. These are displayed in frontend simulation form.

> **Warning:** This class needs update each time a new simulation is added: add an appropiate new attribute.

RK23 = 'Runge-Kutta 3(2)'

RK45 = 'Runge-Kutta 5(4)'

class simulation_api.controller.schemas.**ParamType**(*value*)
Bases: `str`, `enum.Enum`

These are the possible values of `param_type` column in `parameters` table in `simulations.db` database.

ini_cndtn = 'initial condition'

param = 'parameter'

class simulation_api.controller.schemas.**ParameterDBSch**(**, sim_id: str = None, param_type:* simulation_api.controller.schemas.ParamType *= None, param_key: str = None, ini_cndtn_id: int = None, value: float = None, param_id: int = None*)
Bases: *simulation_api.controller.schemas.ParameterDBSchBase*

Model for API type checking when reading a row in `parameters` table in `simulations.db` database.

class Config
Bases: `object`

This class is needed for database reading optimization (thanks to Object Relational Mapper –ORM.)

> **Note:** Read more in FastAPI docs.

orm_model = True

param_id: Optional[int]

class simulation_api.controller.schemas.**ParameterDBSchBase**(*, *sim_id: str = None*, *param_type: simulation_api.controller.schemas.ParamType = None*, *param_key: str = None*, *ini_cndtn_id: int = None*, *value: float = None*)

Bases: `pydantic.main.BaseModel`

Basemodel for API type checking when querrying `parameters` table in `simulations.db` database.

ini_cndtn_id: Optional[int]

param_key: Optional[str]

param_type: Optional[*simulation_api.controller.schemas.ParamType*]

sim_id: Optional[str]

value: Optional[float]

class simulation_api.controller.schemas.**ParameterDBSchCreate**(*, *sim_id: str*, *param_type: simulation_api.controller.schemas.ParamType*, *param_key: str = None*, *ini_cndtn_id: int = None*, *value: float*)

Bases: *simulation_api.controller.schemas.ParameterDBSchBase*

Model for API type checking when creating a row in `parameters` table in `simulations.db` database.

param_type: *simulation_api.controller.schemas.ParamType*

sim_id: str

value: float

class simulation_api.controller.schemas.**PlotDBSch**(*, *sim_id: str = None*, *plot_query_value: str = None*, *plot_id: int = None*)

Bases: *simulation_api.controller.schemas.PlotDBSchBase*

Model for API type checking when reading a row in `plots` table in `simulations.db` database.

class Config

Bases: `object`

This class is needed for database reading optimization (thanks to Object Relational Mapper –ORM.)

Note: Read more in FastAPI docs.

orm_model = True

plot_id: Optional[int]

class simulation_api.controller.schemas.**PlotDBSchBase**(*, *sim_id: str = None*, *plot_query_value: str = None*)

Bases: pydantic.main.BaseModel

Basemodel for API type checking when querrying plots table in simulations.db database.

plot_query_value: Optional[str]

sim_id: Optional[str]

class simulation_api.controller.schemas.**PlotDBSchCreate**(*, *sim_id: str*, *plot_query_value: str*)

Bases: *simulation_api.controller.schemas.PlotDBSchBase*

Model for API type checking when creating a 'plot information' row in plots table in simulations.db database.

plot_query_value: str

sim_id: str

simulation_api.controller.schemas.**PlotQueryValues = typing.Union[simulation_api.controller.schemas.PlotQueryValues_HO, simulation**

Union of the classes defining Enums of plot query values for each system. This is needed in SimStatus.

class simulation_api.controller.schemas.**PlotQueryValues_ChenLee**(*value*)

Bases: str, enum.Enum

List of tags of each different plot generated automatically by the backend when a Chen-Lee simulation is requested.

These tags are used as the possible values of the querry param value in route /api/results/{sim_id}/plot?value=<plot_query_value>.

Note: simualtion_API.controller.schemas.SimIdResponse.sim_id must be related to the Chen Lee system.

---

project = 'project'

threeD = 'threeD'

class simulation_api.controller.schemas.**PlotQueryValues_HO**(*value*)

Bases: str, enum.Enum

List of tags of each different plot generated automatically by the backend when a Harmonic Oscillator simulation is requested.

These tags are used as the possible values of the querry param value in route /api/results/{sim_id}/plot?value=<plot_query_value>.

Note: `simualtion_API.controller.shcemas.SimIdResponse.sim_id` must be related to the Harmonic Oscillator system.

coord = 'coord'

phase = 'phase'

class simulation_api.controller.schemas.**SimForm**(**, username: str = 'Pepito', t0: float = 0.0, tf: float = 6.283185307179586, dt: float = 0.3141592653589793, t_steps: int = 0, method: simulation_api.controller.schemas.IntegrationMethods = 'RK45'*)

Bases: `pydantic.main.BaseModel`

Basemodel of schema used to Request a Simulation in Frontend.

Warning: Needs update each time a new simulation is added:

1. Create a new appropiate class similar to *HOSimForm* or to *ChenLeeSimForm*.
2. Add the class to the `dict` *SimFormDict* defined somewhere in this module.

dt: Optional[float]
Time step size of simulation.

method: Optional[*simulation_api.controller.schemas.IntegrationMethods*]
Integration method used in simulation.

t0: Optional[float]
Initial time of simulation.

t_steps: Optional[int]
Number of time steps. If different from 0, dt is ignored.

tf: Optional[float]
Final time of simulation.

username: Optional[str]

simulation_api.controller.schemas.**SimFormDict = {'Chen-Lee-Attractor': <class 'simulation_api.controller.schemas.ChenLeeSimForm'**
Maps the name of each available system to its simulation form model.

> **Warning:** Needs update each time a new simulation is added: add a new appropiate item to this dict.

class simulation_api.controller.schemas.**SimIdResponse**(**, sim_id: str = None, user_id: int = None, username: str = None, sim_sys:* simulation_api.controller.schemas.SimSystem *= None, sim_status_path: str = None, sim_pickle_path: str = None, message: str = None*)

Bases: `pydantic.main.BaseModel`

Schema for the response of a simulation request (requested via POST in route `/api/simulate/{sim_sys}`.)

Note: The request of the simulation must follow the model *SimRequest*.

message: Optional[str]
Explanatory message.

sim_id: Optional[str]
ID of simulation.

sim_pickle_path: Optional[str]
Path to GET (download) a pickle with the results of the simulation.

sim_status_path: Optional[str]
Path to GET the status of the simulation.

sim_sys: Optional[*simulation_api.controller.schemas.SimSystem*]
Simulated system.

user_id: Optional[int]
User id number stored in database.

username: Optional[str]

class simulation_api.controller.schemas.**SimRequest**(**, system: simulation_api.controller.schemas.SimSystem = <SimSystem.HO: 'Harmonic-Oscillator'>, t_span: List[float] = [], t_eval: List[float] = [], t_steps: int = 0, ini_cndtn: List[float] = [], params: Dict[str, float], method: simulation_api.controller.schemas.IntegrationMethods = 'RK45', sim_id: str = None, user_id: int = 0, username: str = 'Pepito Perez'*)

Bases: `pydantic.main.BaseModel`

Schema needed to request simulations via POST in `/api/request/{sim_system}`.

For the attributes that do not have a description see *simulation_api.simulation.simulations.Simulation*.

Note: Most of the attributes in this pydantic class are arguments of the classes defined in the module *simulation_api.simulation.simulations*, for more information please refer to It.

---

ini_cndtn: List[float]

method: Optional[*simulation_api.controller.schemas.IntegrationMethods*]

params: Dict[str, float]

sim_id: Optional[str]
ID of simulation. This is handled internally, leave it blank when requesting a simulation.

system: *simulation_api.controller.schemas.SimSystem*

t_eval: Optional[List[float]]

t_span: List[float]

t_steps: Optional[int]

user_id: Optional[int]
User id number stored in database. This is handled internally, leave it blank when requesting a simulation.

username: str

class simulation_api.controller.schemas.**SimResults**(**, sim_results: scipy.integrate._ivp.ivp.OdeResult*)
Bases: `pydantic.main.BaseModel`

Results of simulation as returned by `scipy.integrate.solve_ivp`

sim_results: scipy.integrate._ivp.ivp.OdeResult

class simulation_api.controller.schemas.**SimStatus**(**, sim_id: str, user_id: int, date: datetime.datetime, system:* simulation_api.controller.schemas.SimSystem *= None, ini_cndtn: List[float] = None, params: Dict[str, float] = None, method:* simulation_api.controller.schemas.IntegrationMethods *= None, route_pickle: str = None, route_results: str = None, route_plots: str = None, plot_query_values: List[Union[*simulation_api.controller.schemas.PlotQueryValues_HO, simulation_api.controller.schemas.PlotQueryValues_ChenLee*]] = None, plot_query_receipe: str = "'route_plots' + '?value=' + 'plot_query_value'", success: bool = None, message: str = None*)
Bases: `pydantic.main.BaseModel`

Schema of the status of simulations.

This pydantic model is intended to store paths of results of the simulations algong with some metadata. This information can be accessed via GET in `/api/simulate/status/{sim_id}`.

date: datetime.datetime
Date of request of simulation.

ini_cndtn: Optional[List[float]]

message: Optional[str]
Additional information on status of simulation.

method: Optional[*simulation_api.controller.schemas.IntegrationMethods*]

params: Optional[Dict[str, float]]

plot_query_receipe: Optional[str]

plot_query_values: Optional[List[Union[*simulation_api.controller.schemas.PlotQueryValues_HO*, *simulation_api.controller.schem*
Query params values of different automatically generated plots. These values are needed to download the plots in route `/api/results/{sim_id}/plot?value=<plot_query_value>`.

route_pickle: Optional[str]
Route of pickle file generated by the simulation.

route_plots: Optional[str]
Route of plots generated by the simulation backend.

route_results: Optional[str]
Route of frontend showing results.

sim_id: str
ID of simulation.

success: Optional[bool]
Success status of simulation.

system: Optional[*simulation_api.controller.schemas.SimSystem*]
Simulated system.

user_id: int
User id number stored in database.

class simulation_api.controller.schemas.**SimSystem**(*value*)

Bases: str, enum.Enum

List of available systems for simulation.

> **Warning:** The values of the attributes of this class must coincide with the dictionary keys defined in *simulation_api.simulation.simulations.Simulations*, otherwise the system won't be simulated by the backend.

> **Warning:** This class needs update each time a new simulation is added: add an appropiate new attribute.

ChenLee: str = 'Chen-Lee-Attractor'

Chen-Lee Attractor Enum attribute.

HO: str = 'Harmonic-Oscillator'

Harmonic Oscillator Enum attribute.

simulation_api.controller.schemas.**SimSystem_to_SimParams = {'Chen-Lee-Attractor': <class 'simulation_api.controller.schemas.Chen**

Maps the name of each available system to its parameters model

> **Warning:** Needs update each time a new simulation is added: add a new appropiate item to this dict.

class simulation_api.controller.schemas.**SimulationDBSch**(**, sim_id: str = None, user_id: int = None, date: str = None, system: str = None, method: str = None, route_pickle: str = None, route_results: str = None, route_plots: str = None, success: bool = None, message: str = None*)

Bases: *simulation_api.controller.schemas.SimulationDBSchBase*

Model for API type checking when reading a row in simulations table in simulations.db database.

class Config

Bases: object

This class is needed for database reading optimization (thanks to Object Relational Mapper –ORM.)

> **Note:** Read more in FastAPI docs.

orm_model = True

date: Optional[str]

message: Optional[str]

method: Optional[str]

route_pickle: Optional[str]

route_plots: Optional[str]

route_results: Optional[str]

sim_id: Optional[str]

success: Optional[bool]

system: Optional[str]

user_id: Optional[int]

class simulation_api.controller.schemas.**SimulationDBSchBase**(**, sim_id: str = None, user_id: int = None, date: str = None, system: str = None, method: str = None, route_pickle: str = None, route_results: str = None, route_plots: str = None, success: bool = None, message: str = None*)

Bases: `pydantic.main.BaseModel`

Basemodel for API type checking when querrying `simulations` table in `simulations.db` database.

date: Optional[str]

message: Optional[str]

method: Optional[str]

route_pickle: Optional[str]

route_plots: Optional[str]

route_results: Optional[str]

sim_id: Optional[str]

success: Optional[bool]

system: Optional[str]

user_id: Optional[int]

class simulation_api.controller.schemas.**SimulationDBSchCreate**(**, sim_id: str, user_id: int, date: str, system: str, method: str = None, route_pickle: str = None, route_results: str = None, route_plots: str = None, success: bool, message: str = None*)

Bases: *simulation_api.controller.schemas.SimulationDBSchBase*

Model for API type checking when creating a 'simulation information' row in `simulations` table in `simulations.db` database.

date: str

sim_id: str

success: bool

system: str

user_id: int

class simulation_api.controller.schemas.**UserDBSch**(**, username: str = None, user_id: int = None*)

Bases: *simulation_api.controller.schemas.UserDBSchBase*

Model for API type checking when reading a user information in `users` table in `simulations.db` database.

class Config

Bases: `object`

This class is needed for database reading optimization (thanks to Object Relational Mapper –ORM.)

Note: Read more in FastAPI docs.

---

orm_model = True

user_id: Optional[int]

class simulation_api.controller.schemas.**UserDBSchBase**(**, username: str = None*)

Bases: `pydantic.main.BaseModel`

Basemodel for API type checking when querrying `users` table in `simulations.db` database.

username: Optional[str]

class simulation_api.controller.schemas.**UserDBSchCreate**(**, username: str, hash_value: str = None*)

Bases: `pydantic.main.BaseModel`

Model for API type checking when creating a 'user information' row in `users` table in `simulations.db` database.

hash_value: Optional[str]

username: str

simulation_api.controller.schemas.**params_mapping_ChenLee = {'param0': 'a', 'param1': 'b', 'param2': 'c'}**
Maps the name of each Chen-Lee Attractor parameter in frontend form to its name in backend (defined by its corresponding attribute in class `ChenLeeAttractor`)

simulation_api.controller.schemas.**params_mapping_HO = {'param0': 'm', 'param1': 'k'}**
Maps the name of each Harmonic Oscillator parameter in frontend form to its name in backend (defined by its corresponding attribute in class `HarmonicOsc1D`)

simulation_api.controller.schemas.**system_to_params_dict = {'Chen-Lee-Attractor': {'param0': 'a', 'param1': 'b', 'param2': 'c'**
Maps the name of each available system to its parameter change-of-convention mapping (e.g. `params_mapping_HO` or `params_mapping_ChenLee`.)

This is used to translate the parameters name convention in frontend simulation request to the parameters name convention in backend simulation request (with appropiate schema given by `SimRequest`.)

3.1.1.2.1.5 simulation_api.controller.tasks

This file will do background tasks e.g. the simulation

simulation_api.controller.tasks.**_api_simulation_request**(*sim_system:* simulation_api.controller.schemas.SimSystem, *sim_params:* simulation_api.controller.schemas.SimRequest, *background_tasks:* *starlette.background.BackgroundTasks*, *db:* *sqlalchemy.orm.session.Session*) → *simulation_api.controller.schemas.SimIdResponse*
Requests simulation to BackgroundTasks.

Parameters

- **sim_system** (`SimSystem`) – System to be simulated.
- **sim_params** (`SimRequest`) – Contains all the information about the simulation request.
- **background_tasks** (`fastapi.BackgroundTasks`) – Object needed to request simulation in the background.
- **db** (`sqlalchemy.orm.Session`) – Needed for interaction with database.

Returns **sim_id_response** – Contains information about simulation request, such as simulation ID and others. See `SimIdResponse` for more information.

Return type *SimIdResponse*

simulation_api.controller.tasks.**_check_chen_lee_params**(*a: float*, *b: float*, *c: float*)
Checks that the set of Chen-Lee parameters satisfy chaotic conditions, therefore bound solutions.

The conditions are

$$a > 0 \text{ and } b < 0 \text{ and } c < 0 \text{ and } a < -(b + c)$$

Note: This conditions are stated in this reference.

Parameters

- **a** (*float*) – ω_x parameter.
- **b** (*float*) – ω_y parameter.
- **c** (*float*) – ω_z parameter.

`simulation_api.controller.tasks._create_pickle_path_disk` (*sim_id: str*) → str

Creates disk path to simulation results (pickle) by `sim_id`.

`simulation_api.controller.tasks._create_plot_path_disk` (*sim_id: str*, *query_param: Union[*simulation_api.controller.schemas.PlotQueryValues_HO, simulation_api.controller.schemas.PlotQueryValues_ChenLee*]*, *plot_format: str = '.png'*) → str

Creates disk path to plots of simulation results by `sim_id`.

`simulation_api.controller.tasks._pickle` (*file_name: str*, *path: str = ''*, *data: Optional[dict] = None*) → Any

Saves `data` to pickle or reads Python object from pickle.

Saves `data` to pickle if `data`. Otherwise it will try to read a pickle from `path + '/' + file_name` and return the python object stored in it.

Parameters

- **file_name** (*str*) – Name of file to be read from or to write on.
- **path** (*str*) – Path of directory in which the file will be saved or read from.
- **data** (*dict or None, optional*) – If you want to save data to a pickle, provide the data as dictionary. Default is None.

Returns **loaded_object** – Object loaded when no data is provided.

Return type Any or None

`simulation_api.controller.tasks._plot_solution` (*sim_results:* simulation_api.controller.schemas.SimResults, *system:* simulation_api.controller.schemas.SimSystem, *plots_basename: str = '00000'*) → List[str]

Generates relevant simulation's plots and saves them.

Parameters

- **sim_results** (SimResults) – Simulation results as returned by *simulate()*.
- **system** (SimSystem) – System to be simulated.
- **plots_basename** (*str*) – Base name of the plots. Actual name of each plot will be <plotbasename>_<plot_query_value>.png, where <plot_query_value> is a special tag for each type of plot. In this API, baseplot will always be the value of *sim_id*.

Returns **plot_query_values** – Names of each type of plot. These are very important since they are needed to access the plots in the API route (these are the possible values for the query param "value" in route /api/results/{sim_id}/plot).

Return type List[str]

simulation_api.controller.tasks.**_run_simulation**(*sim_params:* simulation_api.controller.schemas.SimRequest) → None

Runs the requested simulation and stores the outcome in a database.

This function runs the simulation, stores the simulation parameters in a database, stores the simulation result in a pickle and creates and saves relevant plots of the simulation.

Parameters **sim_params** (SimRequest) – Contains all the information needed for the simulation.

Returns

Return type None

simulation_api.controller.tasks.**_sim_form_to_sim_request**(*form: Dict[str, str]*) → *simulation_api.controller.schemas.SimRequest*

Translates simulation form –from frontend– to simulation request which is understood by backend in *_api_simulation_request()*.

Parameters **form** (*Dict[str, str]*) – Simulation request information as obtained by frontend.

Returns Simulation request information in a format the backend understands.

Return type *SimRequest*

3.1.1.2.2 `simulation_api.model`

3.1.1.2.2.1 Package contents

3.1.1.2.2.2 Submodules

3.1.1.2.2.3 `simulation_api.model.crud`

This program manages database querys. CRUD comes from: Create, Read, Update, and Delete.

`simulation_api.model.crud.`**`_create_parameters`**(*db: sqlalchemy.orm.session.Session*, *parameters: List[*simulation_api.controller.schemas.ParameterDBSchCreate*]*) → None

Insert parameter entry into parameters table.

Parameters

- **db** (*Session*) – Database Session.
- **parameters** (*List[*ParameterDBSchCreate*]*) – Parameter row in `parameters`' table.

Returns

Return type None

`simulation_api.model.crud.`**`_create_plot_query_values`**(*db: sqlalchemy.orm.session.Session*, *plot_query_params: List[*simulation_api.controller.schemas.PlotDBSchCreate*]*) → None

Insert row in plots table (contains plot query params)

Parameters

- **db** (*Session*) – Database Session.
- **plot_query_params** (*List[*PlotDBSchCreate*]*) – List of rows to be inserted in `plots` table.

Returns

Return type None

`simulation_api.model.crud.`**`_create_simulation`**(*db: sqlalchemy.orm.session.Session*, *simulation:* simulation_api.controller.schemas.SimulationDBSchCreate) → *simulation_api.model.models.SimulationDB*

Inserts simulation in simulations table.

Parameters

- **db** (*Session*) – Database Session.
- **simulation** (SimulationDBSchCreate) – Simulation row in simulations table.

Returns **db_simulation** – Updated simulation's row.

Return type *SimulationDB*

simulation_api.model.crud.**_create_user**(*db: sqlalchemy.orm.session.Session*, *user:* simulation_api.controller.schemas.UserDBSchCreate) → *simulation_api.model.models.UserDB*

Inserts user in users table.

Parameters

- **db** (*Session*) – Database Session.
- **user** (UserDBSchCreate) – User row in database.

Returns **db_user** – Updated inserted row (with *user_id*.)

Return type *UserDB*

simulation_api.model.crud.**_get_all_simulations**(*db: sqlalchemy.orm.session.Session*) → Tuple[*simulation_api.model.models.SimulationDB*]

Get all simulation entries in simulations table.

Parameters **db** (*Session*) – Database Session.

Returns Querry of all rows in simulations table.

Return type sqlalchemy.orm.Query

simulation_api.model.crud.**_get_parameters**(*db: sqlalchemy.orm.session.Session*, *sim_id: str*, *param_type:* simulation_api.controller.schemas.ParamType) → Union[List[float], Dict[str, float]]

Get parameters from parameters table.

Parameters

- **db** (*Session*) – Database Session.
- **sim_id** (*str*) – Simulation ID.
- **param_type** (ParamType) – Type of parameter, wether 'initial condition' or 'parameter'.

Returns list of initial conditions or dict mapping parameter names to parameter values.

Return type List[float] or Dict[str, float] `simulation_api.model.crud.`**`_get_plot_query_values`**(*db: sqlalchemy.orm.session.Session*, *sim_id: str*)

Return plot query parameters for a given simulation.

Parameters

- **db** (*Session*) – Database Session.
- **sim_id** (*str*) – Simulation ID.

Returns Plot query values associated to `sim_id`.

Return type List[*PlotQueryValues*]

`simulation_api.model.crud.`**`_get_simulation`**(*db: sqlalchemy.orm.session.Session*, *sim_id: str*) → *simulation_api.model.models.SimulationDB*

Get simulation with specific id from simulations table.

Parameters

- **db** (*Session*) – Database Session.
- **sim_id** (*str*) – Simulation ID.

Returns Query with simulation information of `sim_id`.

Return type `sqlalchemy.orm.Query`

`simulation_api.model.crud.`**`_get_username`**(*db: sqlalchemy.orm.session.Session*, *user_id: int*)

Gets `user` from `users` table.

Parameters

- **db** (*Session*) – Database Session.
- **user_id** (*int*) –

Returns Query with information about username with given `user_id`.

Return type `sqlalchemy.orm.Query`

3.1.1.2.2.4 `simulation_api.model.db_manager`

This module starts the database engine, the database session and the basemodel for the database tables.

3.1.1.2.2.5 `simulation_api.model.models`

This program creates all the models and tables in the database

class `simulation_api.model.models.`**`ParameterDB`**`(**kwargs)`

Bases: `sqlalchemy.ext.declarative.api.Base`

Parmaeters table model.

Stores parameters and initial conditions of simulations.

`__init__``(**kwargs)`

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

`ini_cndtn_id`

Initial condition position in array of initial conditions.

`param_id`

`param_key`

Name of parameter. Must be one of the required parameters related to the system being simulated.

`param_type`

Parameter type, wether `'initial condition'` or `'parameter'`.

`sim_id`

Simulation ID.

`simulation`

ORM relationship with simulations' table.

`value`

Value of `'parameter'` or `initial contidion`.

class simulation_api.model.models.**PlotDB**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base

Plots table model.

Stores query parameter values of plots needed to access simulation results via GET in route /api/results/{sim_id}/plot?value={plot_query_value}.

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

plot_id

Primary key.

plot_query_value

Label of each plot, used as query value for query param value in route /api/results/{sim_id}/plot.

sim_id

Simulation ID.

simulation

ORM relationship with simulations' table.

class simulation_api.model.models.**SimulationDB**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base

Simulation Status table model.

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

date

message

Message with further information about the simulation status.

method

Integration method.

parameters
ORM relationship with parameters' table.

plots
ORM relationship with plots' table.

route_pickle
API route to GET simulation results in pickle format.

route_plots
API route to GET simulation plots.

route_results
API route to GET simulation results displayed in frontend web page.

sim_id
Simulation ID.

success
Tells if the simulation was successful or not.

system
Simulated system.

user
ORM relationship with users' table.

user_id
user_id.

Type Foreign key

class simulation_api.model.models.**UserDB**(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

Users table model.

This table stores basic user information.

Note: *hash_value* and this `users` table is not appropiately used yet because logging is not yet implemented in the app.

__init__(***kwargs*)
A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

hash_value
Hash value of the usser's password.

simulations
ORM relationship with simulations' table.

user_id

username

3.1.1.2.3 simulation_api.simulation

3.1.1.2.3.1 Package contents

3.1.1.2.3.2 Submodules

3.1.1.2.3.3 simulation_api.simulation.demo_run_simulation

Test for simulations defined in simulation module

3.1.1.2.3.4 simulation_api.simulation.simulations

This module simulates mechanical systems

class simulation_api.simulation.simulations.**ChenLeeAttractor**(*t_span: Optional[Tuple[float, float]] = [0, 400], t_eval: Optional[tuple] = None, ini_cndtn: List[float] = [10.0, 10.0, 0.0], params: dict = {'a': 3.0, 'b': - 5.0, 'c': - 1.0}, method: str = 'RK45', user_name: Optional[str] = None*)

Bases: *simulation_api.simulation.simulations.Simulation*

Simulates Chen-Lee Attractor.

a

ω_x parameter.

Type float

b

ω_y parameter.

Type float

c

ω_z parameter.

Type float

Notes

The Chen-Lee Attractor is a dynamical system defined by:[1]

$$\frac{d\omega_x}{dt} = -\omega_y\omega_z + a\,\omega_x$$
$$\frac{d\omega_y}{dt} = \omega_z\omega_x + b\,\omega_y$$
$$\frac{d\omega_z}{dt} = \frac{1}{3}\omega_x\omega_y + c\,\omega_z$$

Its origin is closely related to the motion of a rigid body in a rotating frame of reference.

References

__init__(*t_span: Optional[Tuple[float, float]] = [0, 400], t_eval: Optional[tuple] = None, ini_cndtn: List[float] = [10.0, 10.0, 0.0], params: dict = {'a': 3.0, 'b': - 5.0, 'c': - 1.0}, method: str = 'RK45', user_name: Optional[str] = None*) → None

Extends `Simulation.__init__()`

Adds attributes `ChenLeeAttractor.a`, `ChenLeeAttractor.b` and `ChenLeeAttractor.c`.

Parameters

- **ini_cndtn** (*array_like, shape (3,)*) – Initial condition of 1D Harmonic Oscillator. Convention:

```
{
    "a": float,      # `\omega_x` parameter.
    "b": float,      # `\omega_x` parameter.
```

(continues on next page)

[1] https://doi.org/10.1142/S0218127403006509

(continued from previous page)

```
    "c": float,      # `\omega_z` parameter.
}
```

Default is `{"a": 3.0, "b": - 5.0, "c": - 1.0}`.

dyn_sys_eqns (*t: float*, *w: List[float]*) → List[float]
Chen-Lee Dynamical system definition

Note: Overwrites *Simulation.dyn_sys_eqns*.

Parameters

- **w** (*array_like, shape (3,)*) – Vector of angular velocity. Convention: $\mathtt{w} = [\omega_x, \omega_y, \omega_z]$.
- **t** (*float*) – Time.

Returns **dwdt** – Dynamical system equations of Chen Lee attractor evaluated at `w`.

Return type array_like, shape (3,)

system = 'Chen-Lee-Attractor'

class simulation_api.simulation.simulations.**HarmonicOsc1D** (*t_span: Optional[Tuple[float, float]] = [0, 6.283185307179586]*, *t_eval: Optional[tuple] = None*, *ini_cndtn: List[float] = [0.0, 1.0]*, *params: dict = {'k': 1.0, 'm': 1.0}*, *method: str = 'RK45'*, *user_name: Optional[str] = None*)

Bases: *simulation_api.simulation.simulations.Simulation*

1-D Harmonic Oscillator simulation

m

Mass of object.

Type float

k

Force constant of harmonic oscilltor.

Type float

Notes

The hamiltonian describing the Harmonic Oscillator is defined dy

$$H = \frac{1}{2m}p^2 + \frac{1}{2}kq^2$$

__init__(*t_span: Optional[Tuple[float, float]] = [0, 6.283185307179586]*, *t_eval: Optional[tuple] = None*, *ini_cndtn: List[float] = [0.0, 1.0]*, *params: dict = {'k': 1.0, 'm': 1.0}*, *method: str = 'RK45'*, *user_name: Optional[str] = None*) → None

Extends `Simulation.__init__()`

Adds attributes `HarmonicOsc1D.m` and `HarmonicOsc1D.k`.

Parameters

- **ini_cndtn** (*array_like, shape (2,)*) – Initial condition of 1D Harmonic Oscillator. Convention:

-
```
{
    "m": float,      # Mass of object.
    "k": float,      # Force constant of harmonic oscilltor.
}
```

Default is `{"m": 1., "k": 1.}`.

dyn_sys_eqns(*t: float*, *y: List[float]*) → List[float]

Hamilton's equations for 1D-Harmonic Oscillator.

Note: Overwrites `Simulation.dyn_sys_eqns`.

Parameters

- **t** (*float*) – Time of evaluation of Hamilton's equations.
- **y** (*array_like, shape (2,)*) – Canonical coordinates. Convention: $\mathtt{y} = [q, p]$ where q is the generalised position and p is the generalised momentum.

Returns dydt – Hamilton's equations for 1D Harmonic Oscillator. $\mathtt{dydt} = \left[\frac{dq}{dt}, \frac{dp}{dt}\right] = \left[\frac{\partial H}{\partial p}, -\frac{\partial H}{\partial q}\right]$

Return type array_like, shape (2,)

system = 'Harmonic-Oscillator'

class simulation_api.simulation.simulations.**Simulation**(*t_span: Optional[List[float]] = None, t_eval: Optional[list] = None, ini_cndtn: Optional[list] = None, params: Optional[dict] = None, method: Optional[str] = 'RK45', user_name: Optional[str] = None*)

Bases: `object`

Simulation of a continuous dynamical system described by first order coupled differential equations.

t_span
Interval of integration (t0, tf).

Type List[float, float] or None

t_eval
Times at which to store the computed solution, must be sorted and lie within t_span.

Type array_like or None

ini_cndtn
Initial condition of simulation, its specification depends on the system being simulated.

Type array_like or None

params
Contains all the parameters of the simulation (e.g. for the harmonic oscillator `self.params = {"m": 1., "k": 1.}`)

Type dict or None

method
Method of integration.

Type str, optional

user_name
Username that instantiated the simulation.

Type str or None

date
UTC date and time of instantiation of object.

Type datetime (str)

results
Results of simulation.

Type `scipy.integrate._ivp.ivp.OdeResult` or None

__init__(*t_span: Optional[List[float]] = None*, *t_eval: Optional[list] = None*, *ini_cndtn: Optional[list] = None*, *params: Optional[dict] = None*, *method: Optional[str] = 'RK45'*, *user_name: Optional[str] = None*) → None
Initializes all `self` attributes except `self.system`

dyn_sys_eqns(*t: float*, *y: List[float]*) → List[float]
Trivial 2D dynamical system. Just for reference.

Note: The actual simulations that inherit this class will replace this method with the relevant dynamical equations.

simulate() → scipy.integrate._ivp.ivp.OdeResult
Simulates `self.system` abstracted in `self.dyn_sys_eqns` and using `scipy.integrate.solve_ivp`.

Returns

self.results –

Bunch object with the following fields defined:

t [ndarray, shape (n_points,)] Time points.

y [ndarray, shape (n, n_points)] Values of the solution at t.

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: -1, Integration step failed; 0, The solver successfully reached the end of tspan; 1, A termination event occurred.

message [string] Human-readable description of the termination reason.

success [bool] True if the solver reached the interval end or a termination event occurred (status >= 0).

Return type OdeResult

system = None
Name of system.

`simulation_api.simulation.simulations.`**`Simulations = {'Chen-Lee-Attractor': <class 'simulation_api.simulation.simulations.ChenLee`**
Maps the names of the available systems to their corresponding classes.

> **Warning:** Must be updated each time a new simulation is added (add the new relevant item to the dictionary).

3.1.1.3 Submodules

3.1.1.3.1 simulation_API.config module

General configurations of our API.

3.1.2 Add a new simulation

Keep in mind that the systems we can add to our API are those that can be integrated by `scipy.integrate.solve_ivp`. Those are basically first order coupled differential equations i.e. systems of the form

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t).$$

If you want to add a new simulation to this API just follow the steps described below.

3.1.2.1 1. Add the simulation to `simulations` module

1. Add a relevant class to *simulations*. This class will define the relevant parameters used in the simulation as well as its dynamic equations. It must inherit *Simulation* and must have the same structure as *HarmonicOsc1D* and *ChenLeeAttractor*. Don't forget to test your simulation by playing around in *demo_run_simulation*.
2. Don't forget to add the attribute *system* to your class. The value of this attribute must be the name of the system separated by dashes. Use only alphanumerical characters and dashes.

3. Add the relevant simulation class you just created to the `dict` *Simulations*. This will tell the API that the simulation exists and it is available.

3.1.2.2 2. Add relevant schemas and models to `schemas`

Follow the steps mentioned below –some of them may not make sense at first glance, but until you write the code.

1. Add the *system* attribute value of the recently created class to *SimSystem*.
2. Create a class that inherits *SimForm*. It must be similar to *HOSimForm* and *ChenLeeSimForm*. This class will be used to check the simulation information provided in frontend.
3. Add a relevant item –related to the class created in the last numeral– to the `dict` *SimFormDict*. This will map the name of the system to its simulation form model.
4. Add a new class similar to *HOParams* and *ChenLeeParams*. The names of the attributes must match the names of the parameters defined in the relevant simulation class, created in the first numeral of this list.
5. Add an appropiate item to the `dict` *SimSystem_to_SimParams*.
6. Create an appropiate `dict` similar to *params_mapping_HO* and *params_mapping_ChenLee*.
7. Add an appropiate item to the dict *system_to_params_dict*.
8. Create a new class similar to *PlotQueryValues_HO* and *PlotQueryValues_ChenLee*.
9. Add an appropiate item to *PlotQueryValues*.

If you do not understand some of the steps above or how to implement them, refer to *the documentaton* of the relevant classes or schemas for the already available systems –Chen-Lee Attractor or Harmonic Oscillator–, it may enlighten you.

3.1.2.3 3. Add relevant plots to `_plot_solution()`

Here you can add two or three intersting plots related to the simulation you just added and tested. The code that generates the plots must be placed in *simulation_api.controller.tasks._plot_solution()*.

A few things to take into account:

1. We use matplotlib, but we use the class `Figure` directly, we do not use pyplot. This is related to some problems that may arise with the pyplot package and the web applocation backend, as mentioned in matplotlib's documentation.
2. Note that the plots related to the simulations are defined in an `if` or `elif` block each one. Add a new block for the simulation you want to add.
3. The first two lines of code that generate each plot related to the recently created simulation must look something like:

```
plot_query_value = PlotQueryValues_HO.phase.value
plot_query_values.append(plot_query_value)
```

For each generated plot,we define a `plot_query_value` that comes directly from the class defined in item number 8 of the *last section*. In the example given above,the class was named `PlotQueryValues_HO`,the attribute related to the `plot_query_value` of the relevant plot was named `coord` and the value of the latter is accessed by using `.value`. Each `plot_query_value` is appended to the list `plot_query_values`,which is the return value of *`_plot_solution()`*. This item is very important,since the values we define here are used to name the plots as well as to look them up.

4. Finally,the last line of code that generates each plot must be:

```
fig.savefig(_create_plot_path_disk(plots_basename, plot_query_value))
```

This will ensure that the name of the plot has always the same format

3.1.2.4 4. Add relevant form entries in frontend

Modify appropiately the template `simulation_api/templates/request-simulation.html`. This template is the one that asks for the simulation parameters in the frontend.

Some things to take into account:

1. Note that each system has its own `if` or `elif` block. Stick to this convention and add a new block related to the new simulation (the new system).
2. In the `if` block mentioned above there are only two main things the form should ask for: initial conditions and parameters of simulation.

3. For the initial conditions the value of the HTML attribute `name` should start with the string `"ini"` followed by the index in the initial condition array defined in your simulation class attribute `ini_cndtn`. For example, for the harmonic oscillator the convention of initial condition is

3.1.2.5 5. Modify `results.html` template to show results

Finally, we need to add a relevant `elif` block to the template `simulation_api/templates/results.html`. This template should show the generated plots, give the option to download them with a button and give the option to download the pickle file as well.

`simulation_api` Initialization of web application.

`simulation_api.controller` The core package of our API. Here you can find the main app, schemas and background tasks.

simulation_api.model Database-related package.

simulation_api.simulation Simulation-related package. Here you can find all the programs we use to simulate the available systems.

simulation_api.config Configuration module. Some very basic configurations of our web application.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER
FIVE

A SPECIAL AKNOWLEDGEMENT

To Camilo Hincapié who guided me in this process.

PYTHON MODULE INDEX

S