hadar-simulator

Release 0.5.0

Dec 03, 2020

Contents

Hadar is a adequacy python library for deterministic and stochastic computation

You are in the technical documentation.

- If you want to discover Hadar and the project, please go to https://www.hadar-simulator.org for an overview
- If you want to start using Hadar, you can begin with *Tutorials*
- If you want to understand Hadar engine, see *Architecture*
- If you want to look at a method or object behavior search inside *Reference*
- If you want to help us coding Hadar, please read *Contributing* before.
- If you want to see Mathematics model used in Hadar, go to *Mathematics*.

CHAPTER 1

Tutorials

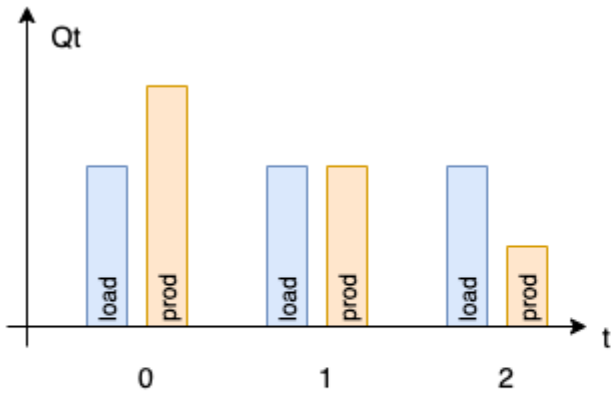
1.1 Get Started

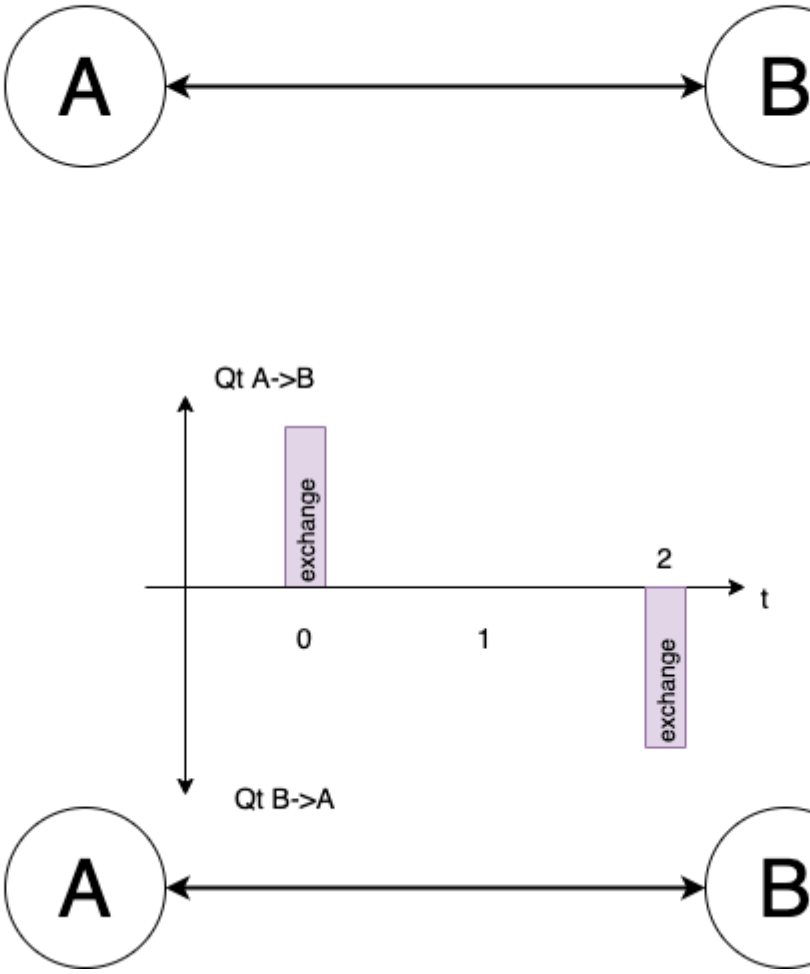
Hadar is a adequacy python library for deterministic and stochastic computation

1.1.1 Adequacy problem

Each kind of network has a needs of adequacy. On one side, some network nodes need to consume items such as watt, litter, package. And other side, some network nodes produce items. Applying adequacy on network, is tring to find the best available exchanges to avoid any lack at the best cost.

For example, a electric grid can have some nodes wich produce too more power and some nodes wich produce not enough power.





In this case, at t=0, A produce 10 more and B need 10 more. Then nodes are well balanced. And at t=2, B produce 10 more and A need 10 more.

For this example, perform adequacy will done ten quantities exachanges from A to B, then zero and at the end 10 quantities from B to A.

Hadar compute adequacy from simple to complex network. For example, to compute above network, just few line need:

Firstly, install hadar : **"pip install hadar"**

```
import hadar as hd
```

```
study = hd.Study(horizon=3)\
    .network()\
        .node('a')\
            .consumption(cost=10 ** 6, quantity=[20, 20, 20], name='load')\
            .production(cost=10, quantity=[30, 20, 10], name='prod')\
        .node('b')\
            .consumption(cost=10 ** 6, quantity=[20, 20, 20], name='load')\
```

(continues on next page)

(continued from previous page)

```
            .production(cost=10, quantity=[10, 20, 30], name='prod')\
        .link(src='a', dest='b', quantity=[10, 10, 10], cost=2)\
        .link(src='b', dest='a', quantity=[10, 10, 10], cost=2)\
    .build()
```

```
optimizer = hd.LPOptimizer()
res = optimizer.solve(study)
```

Then you can analyze by yourself result or use hadar aggragator and plotting

```
plot = hd.HTMLPlotting(agg=hd.ResultAnalyzer(study, res),
                       node_coord={'a': [2.33, 48.86], 'b': [4.38, 50.83]})
```

```
plot.network().node('a').stack()
```

At starts **A** export it production. Then it needs to import.

```
plot.network().node('b').stack(scn=0)
```

At start **B** needs to import then it can export its productions

```
plot.network().map(t=0, zoom=2.5)
```

```
plot.network().map(t=2, zoom=2.5)
```


1.2 Cost and Prioritization

Welcome to the next tutorial !

We will discover why hadar use cost and how to use it.

Hadar is an adequacy optimizer, like every optimizer it needs cost to determinie the best solution. In hadar, the cost to optimize represent a kind of cost needed to perform network adequacy. Than means Hadar will always try to: - use the cheaper production - use the cheaper path inside network - if hadar can't match consumption asked, it will turn off cheaper unavailable consumption cost

1.2.1 Production Prioritize

Let's start an example with a single node, there are 3 types of productions: solar, nuclear, oil. We want to use first all solar, then switch to nuclear and use oil only as last chance. To see production prioritize, we attach a growing consumption to this node.

```
import numpy as np
import hadar as hd
```

build study

```
study = hd.Study(horizon=30)\
    .network()\
        .node('a')\
            .consumption(name='load',    cost=10**6, quantity=np.arange(30))\
            .production(name='solar',    cost=10,    quantity=10)\
            .production(name='nuclear', cost=100,    quantity=10)\
            .production(name='oil',      cost=1000,  quantity=10)\
    .build()

# tips: If you give just one element, hadar will extended it according horizon size␣
↪and scenario size
```

solve study

```
optimizer = hd.LPOptimizer()
res = optimizer.solve(study)
```

instance an aggragator to analyze result

```
agg = hd.ResultAnalyzer(study=study, result=res)
```

inject aggregator inside plottting to visualize result

```
plot = hd.HTMLPlotting(agg=agg)
```

```
plot.network().node('a').stack()
```

1.2.2 Consumption Prioritize

Consumption is bit different. Consumption cost is a unavailabilty cost. Therefore, unlike production, Hadar must to use the highest consumption cost first.

For this example, imagine your are the futur. Hydrogen is the only energy source. You have the classic load, you need to match absolutely. Then you have car recharging consumption, has to be matched but could be stopped time to time. And you have also bitcoin mining, which could be stopped as you want.

```
study = hd.Study(horizon=30)\
    .network()\
        .node('a')\
            .consumption(name='load',    cost=10**6, quantity=10)\
            .consumption(name='car',     cost=10**4, quantity=10)\
            .consumption(name='bitcoin', cost=10**3, quantity=10)\
            .production(name='hydrogen', cost=10,  quantity=np.arange(30))\
    .build()
```

```
res = optimizer.solve(study)
agg = hd.ResultAnalyzer(study=study, result=res)
plot = hd.HTMLPlotting(agg=agg)
```

```
plot.network().node(node='a').stack(cons_kind='given')
```

1.2.3 Border Prioritize

As for production, border cost is a cost of use. Hadar will always select the cheapest cost at first.

For example, Belgium produces many eolien power. It's a good new because England and France has a peek of consumption. However send energy to England by submarin cable is more expansive than send it to France by traditional line. When we modelize network, we keep this technical cost gap. Like that Hadar will firstly send energy to France and if some energy remain, it will be send to England.

```
study = hd.Study(horizon=2)\
    .network()\
        .node('be').production(name='eolien', cost=100, quantity=[10, 20])\
        .node('fr').consumption(name='load',  cost=10**6, quantity=10)\
        .node('uk').consumption(name='load',  cost=10**6, quantity=10)\
        .link(src='be', dest='fr', cost=10, quantity=10)\
        .link(src='be', dest='uk', cost=50, quantity=10)\
    .build()
```

```
res = optimizer.solve(study)
agg = hd.ResultAnalyzer(study=study, result=res)
plot = hd.HTMLPlotting(agg=agg,
                       node_coord={'fr': [2.33, 48.86], 'be': [4.38, 50.83], 'uk': [0,␣
↪52]})
```

```
plot.network().map(t=0, zoom=2.7)
```

At t=0, Belgium has not enough energy for both. Hadar will send it to France to optimize transfert cost.

```
plot.network().map(t=1, zoom=2.7)
```

At t=1, Belgium has enough energy for both.

1.3 FR-DE Adequacy

In this example, we will test hadar on a realistic (yet simplify) use case. We will perform adequacy between France and Germainy during one day.

```
import pandas as pd
import numpy as np
import hadar as hd
```

1.3.1 Import simplify dataset

```
fr = pd.read_csv('fr.csv')
de = pd.read_csv('de.csv')
```

1.3.2 Build study

```
study = hd.Study(horizon=48).network()
```

France loves nuclear, so in this example most of production are nuclear. France has also a bit of solar and when needed country can turn on/off coal generator. We want to optimize adequacy by reduce CO2 production. Therefore: - solar is the cheaper at 10 - then we use nuclear at 30 - and coal at 100

```
study = study.node('fr')\
    .consumption(name='load', cost=10**6, quantity=fr['cons'])\
    .production(name='solar', cost=10, quantity=fr['solar'])\
    .production(name='nuclear', cost=30, quantity=fr['nuclear'])\
    .production(name='coal', cost=100, quantity=fr['coal'])
```

Germainy has stopped nuclear to switch from renewable energy. So we increase solar and eolien production. When renewable energy are off, Germainy need to start coal generation to match its consumption. Like for France, we want to minimize CO2 production: - solar at 10 - eolien at 15 - coal at 100

```
study = study.node('de')\
    .consumption(name='load', cost=10**6, quantity=de['cons'])\
    .production(name='solar', cost=10, quantity=de['solar'])\
    .production(name='eolien', cost=15, quantity=de['eolien'])\
    .production(name='coal', cost=100, quantity=de['coal'])
```

Then both side links are set with same cost at 5. In this network, Germany will be import from nuclear french before to start coal. And France will use germain coal to avoid any loss of load.

```
study = study\
    .link(src='fr', dest='de', cost=5, quantity=4000)\
    .link(src='de', dest='fr', cost=5, quantity=4000)\
    .build()
```

```
optimizer = hd.LPOptimizer()
res = optimizer.solve(study)
```

```
agg = hd.ResultAnalyzer(study, res)
plot = hd.HTMLPlotting(agg=agg,
                       unit_symbol='MW', # Set unit quantity
                       time_start='2020-02-01', # Set time interval
                       time_end='2020-02-02')
```

```
plot.network().rac_matrix()
```

```
plot.network().node(node='fr').stack(prod_kind='used', cons_kind='asked')
```

```
plot.network().node('fr').consumption('load').gaussian(scn=0)
```

```
plot.network().node(node='de').stack()
```

Hadar found a loss of load near 6h in Germany and import from France. Then France had a loss of load, and Hadar exports to France.

```
plot.network().node('de').consumption(name='load').gaussian(scn=0)
```

1.4 Analyze Result

In this example, you learn to use `ResultAnalyzer`. You has already use it in preivous example to instanciate plotting: `agg = hd.ResultAnalyzer(study, result)`

Let's begin by build little study with two nodes (A and B) both has a sinus-like load from 1500 to 500. Node A has a constant nuclear plan, node B has eolien with linear random.

```
import hadar as hd
import numpy as np
import pandas as pd
```

```
t = np.linspace(0, np.pi * 14, 168)
load = 1000 + np.sin(t) * 500
eolien = np.random.rand(t.size) * 1000
```

```
study = hd.Study(horizon=t.size, nb_scn=1)\
    .network()\
        .node('a')\
            .consumption(name='load', cost=10 ** 6, quantity=load)\
            .production(name='nuclear', cost=100, quantity=1500)\
        .node('b')\
            .consumption(name='load', cost=10 ** 6, quantity=load)\
            .production(name='eolien', cost=50, quantity=eolien)\
        .link(src='a', dest='b', cost=5, quantity=2000)\
        .link(src='b', dest='a', cost=5, quantity=2000)\
    .build()
```

```
opt = hd.LPOptimizer()
res = opt.solve(study)
```

```
agg = hd.ResultAnalyzer(study=study, result=res)
```

1.4.1 Low API

Analyzer provide a *low* api, that means result could not be ready-to-use, but it's a very flexible way to analyze data. Low API enable to thinks: - set order. data has for level : node, element, scn and time. Low API can organize for your these level - filtering: for each level you can apply a filter, to only select node 'a', or time from 10 to 35 timestep

For examples you want select consumption named load other all node just for 57 to 78 timestep

```
agg.network().scn(0).consumption('load').node().time(slice(57, 78))
```

TIP If filter return only one element, set it at first. First indexes with one element are removed to avoir useless indexes.

Another example: Analyze all production first 24 timestep

```
agg.network().scn(0).node().production().time(slice(0,24))
```

To summrize low api, you can organize and filter data by network, scenarios, time, node and elements on node.

1.4.2 High API

High API is ready to use data. It gives you a business oriented data about adequacy. Today we have: - Get balance to compute net position on a node - Get cost to compute cost on a node - Get Remain Available Capacities

```
import plotly.graph_objects as go
```

```
def plot(y):
    return go.Figure(go.Scatter(x=t, y=y.flatten()))
```

```
data = agg.get_balance(node='a') # Compute net exchange for all scenario and timestep
plot(data)
```

```
data = agg.get_cost(node='b') # Compute cost for all scenario and timestep
plot(data)
```

```
data = agg.get_rac() # Compute Remain Available Capacities for all scenarios and␣
↪timestep
plot(data)
```


1.5 Network Investment

Welcome to this new tutorial. Off course Hadar is well designed to compute study for network adequacy. You can launch Hadar to compute adequacy for the next second or next year.

But Hadar can also be used like a asset investment tool. In this example, thanks to Hadar, we will make the best choice for renewable energy and network investment.

We have a small region, with metropole which doesn't produce anything, a nuclear plan and two small cities with production.

First step parse data with pandas (and plot them)

```
import numpy as np
import pandas as pd
import hadar as hd
import plotly.graph_objects as go
```

1.5.1 Input data

```
a = pd.read_csv('a.csv', index_col='date')
fig = go.Figure()
fig.add_traces(go.Scatter(x=a.index, y=a['consumption'], name='load'))
fig.add_traces(go.Scatter(x=a.index, y=a['gas'], name='gas'))
fig.update_layout(title_text='Node A', yaxis_title='MW')
```

```
b = pd.read_csv('b.csv', index_col='date')
fig = go.Figure()
fig.add_traces(go.Scatter(x=b.index, y=b['consumption'], name='load'))
fig.update_layout(title_text='Node B (only consumption)', yaxis_title='MW')
```

```
c = pd.read_csv('c.csv', index_col='date')
fig = go.Figure()
fig.add_traces(go.Scatter(x=c.index, y=c['nuclear'], name='load'))
fig.update_layout(title_text='Node C (only production)', yaxis_title='MW')
```

```
d = pd.read_csv('d.csv', index_col='date')
fig = go.Figure()
fig.add_traces(go.Scatter(x=d.index, y=d['consumption'], name='load'))
fig.add_traces(go.Scatter(x=d.index, y=d['eolien'], name='eolien'))
fig.update_layout(title_text='Node D', yaxis_title='MW')
```

Base Study

Next step, code this network with Hadar

```
line = np.ones(8760) * 2000 # 2000 MW
```

```
base = hd.Study(horizon=8760)\
    .network()\
        .node('a')\
            .consumption(name='load', cost=10**6, quantity=a['consumption'])\
            .production(name='gas', cost=80, quantity=a['gas'])\
        .node('b')\
            .consumption(name='load', cost=10**6, quantity=b['consumption'])\
        .node('c')\
            .production(name='nuclear', cost=50, quantity=c['nuclear'])\
        .node('d')\
            .consumption(name='load', cost=10**6, quantity=d['consumption'])\
            .production(name='eolien', cost=20, quantity=d['eolien'])\
        .link(src='a', dest='b', cost=5, quantity=line)\
        .link(src='b', dest='c', cost=5, quantity=line)\
        .link(src='c', dest='a', cost=5, quantity=line)\
        .link(src='c', dest='b', cost=10, quantity=line)\
        .link(src='c', dest='d', cost=10, quantity=line)\
        .link(src='d', dest='c', cost=10, quantity=line)\
    .build()
```

```
optimizer = hd.LPOptimizer()
```

```
def compute_cost(study):
    res = optimizer.solve(study)
    agg = hd.ResultAnalyzer(study=study, result=res)
    return agg.get_cost().sum(axis=1), res.benchmark
```

```
def print_bench(bench):
    print('mapper', bench.mapper)
    print('modeler', sum(bench.modeler))
    print('solver', sum(bench.solver))
    print('total', bench.total)
```

```
base_cost, bench = compute_cost(base)
base_cost = base_cost[0]
```

1.5.2 Find best place for solar

An investissor want to build a solar park with solar cells. According to last last data meteo, he could except the amount of production from this park. (Solar radiation is the same on each node of network).

What is the best node to install these solar pans ? (B is excluded because there are not enough space)

```
park = pd.read_csv('solar.csv', index_col='date')
fig = go.Figure()
fig.add_traces(go.Scatter(x=park.index, y=park['solar'], name='solar'))
fig.update_layout(title_text='Forecast Solar Park Power', yaxis_title='MW')
```

We can build one study for each different scenarios. However, Hadar can compute many scenarios at once for a more efficient compute. Result are the same. The possibility to compute many scenarios at once, is very important for next topic Stochastic Study.

```
def build_sparce_data(total: int, at, data) -> np.ndarray:
    """
    Build many scenarios input where all scenario is empty but one.
    :param total: number of scenario to generate
    :param at: scenario index to fill
    :param data: data to fill in selected scenario

    :return: matrix with shape (nb_scn, horizon) where only one scenario is not zero.
    """
    if isinstance(data, pd.DataFrame):
        data = data.values.flatten()
    sparce = np.ones((total, data.size))
    sparce[at, :] = data
    return sparce
```

We use start three studies one for each node.

```
solar = hd.Study(horizon=8760, nb_scn=3)\
    .network()\
        .node('a')\
            .consumption(name='load', cost=10**6, quantity=a['consumption'])\
            .production(name='gas', cost=80, quantity=a['gas'])\
            .production(name='solar', cost=10, quantity=build_sparce_data(total=3,␣
↪at=0, data=park))\
        .node('b')\
            .consumption(name='load', cost=10**6, quantity=b['consumption'])\
        .node('c')\
            .production(name='nuclear', cost=50, quantity=c['nuclear'])\
            .production(name='solar', cost=10, quantity=build_sparce_data(total=3,␣
↪at=1, data=park))\
        .node('d')\
            .consumption(name='load', cost=10**6, quantity=d['consumption'])\
            .production(name='eolien', cost=20, quantity=d['eolien'])\
            .production(name='solar', cost=10, quantity=build_sparce_data(total=3,␣
↪at=2, data=park))\
        .link(src='a', dest='b', cost=5, quantity=line)\
        .link(src='b', dest='c', cost=5, quantity=line)\
        .link(src='c', dest='a', cost=5, quantity=line)\
        .link(src='c', dest='b', cost=10, quantity=line)\
        .link(src='c', dest='d', cost=10, quantity=line)\
        .link(src='d', dest='c', cost=10, quantity=line)\
    .build()
```

```
costs, bench = compute_cost(solar)
costs = pd.Series(data=costs, name='cost', index=['a', 'c', 'd'])
```

```
(base_cost - costs) / base_cost * 100
```

```
a    8.070145
c    2.342062
d    2.736793
Name: cost, dtype: float64
```

As we can see, network is more efficient if solar park is installed one **node A** (8% more efficient than only 2-3% for other node)

1.5.3 Find best place with on more line

Add an extra difficulties ! Region want to invest in a new line between A->C, D->B, A->D, D->A.

In this case, **What is the best place to install solar park and what is the more usefull line to build ?**

```
solar_line = hd.Study(horizon=8760, nb_scn=12)\
    .network()\
        .node('a')\
            .consumption(name='load', cost=10**6, quantity=a['consumption'])\
            .production(name='gas', cost=80, quantity=a['gas'])\
            .production(name='solar', cost=10, quantity=build_sparce_data(total=12,␣
↪at=[0, 3, 6, 9], data=park))\
        .node('b')\
            .consumption(name='load', cost=10**6, quantity=b['consumption'])\
        .node('c')\
            .production(name='nuclear', cost=50, quantity=c['nuclear'])\
            .production(name='solar', cost=10, quantity=build_sparce_data(total=12,␣
↪at=[1, 4, 7, 10], data=park))\
        .node('d')\
            .consumption(name='load', cost=10**6, quantity=d['consumption'])\
            .production(name='eolien', cost=20, quantity=d['eolien'])\
            .production(name='solar', cost=10, quantity=build_sparce_data(total=12,␣
↪at=[2, 5, 8, 11], data=park))\
        .link(src='a', dest='b', cost=5, quantity=line)\
        .link(src='b', dest='c', cost=5, quantity=line)\
        .link(src='c', dest='a', cost=5, quantity=line)\
        .link(src='c', dest='b', cost=10, quantity=line)\
        .link(src='c', dest='d', cost=10, quantity=line)\
        .link(src='d', dest='c', cost=10, quantity=line)\
        .link(src='a', dest='c', cost=10, quantity=build_sparce_data(total=12, at=[0,␣
↪1, 2], data=line))\
        .link(src='d', dest='b', cost=10, quantity=build_sparce_data(total=12, at=[3,␣
↪4, 5], data=line))\
        .link(src='a', dest='d', cost=10, quantity=build_sparce_data(total=12, at=[6,␣
↪7, 8], data=line))\
        .link(src='d', dest='a', cost=10, quantity=build_sparce_data(total=12, at=[9,␣
↪10, 11], data=line))\
    .build()
```

```
costs2, bench = compute_cost(solar_line)
costs2 = pd.DataFrame(data=costs2.reshape(4, 3),
                      index=['a->c', 'd->b', 'a->d', 'd->a'], columns=['a', 'c', 'd'])
```

```
(base_cost - costs2) / base_cost * 100
```

Very interesting, new line is a game changer. D->A and D->B seem most valuable lines. If D->B is created, it's more efficient to install solar park on **node D** !

1.6 Begin Stochastic

1.6.1 What is a stochastic study ?

When you want to simulate a network adequacy, you can perform a deterministic computation. That means you believe you won't have too much fluky behavior in the future. If you perform adequacy for the next hour or day, it's a good hypothesis. But if you simulate network for the next week, month or year, it's sound curious.

Are you sur wind will blow next week or sun will shines ? If not, you eolian or solar production could change. Can you warrant that no failure will occur on your network next month or next year ?

Of course, we can not predict future with such precision. It's why we use stochastic computation. Stochastic means there are fluky behavior in the physics we want simulate. An single simulation is quiet useless, if result can change due to little variation.

The best solution could be to compute a *God function* which tell you for each input variation (solar production, line, consumptions) what is the adequacy result. Like that, Hadar has just to analyze function, its derivatives, min, max, etc to predict future. But this God function doesn't exist, we just have an algorithm which tell us adequacy according to one fixed set of input data.

It's why we use Monte Carlo algorithm. Monte Carlo run many scenarios to analyze many different behavior. Scenario with more consumption in cities, less solar production, less coal production or one line deleted due to crash. By this method we recreate God function by sampling it with the Monte-Carlo method.

1.6.2 Describe example

We will reuse network seen in *Network Investment*. If you don't read this part, don't worry we just reuse network no more. It's look like

We use data generated in the next topic Workflow. Input data representes 10 scenarios with different load and eolien productions. There are also random faults for nuclear and gas. These 10 scenarios are unique. They are 10 random sampling on the *God function* to try to predict more widely network adequacy

```
import hadar as hd
import numpy as np
```

```
def read_csv(name):
    return np.genfromtxt('%s.csv' % name, delimiter=' ').T
```

```
line = 2000
study = hd.Study(horizon=168, nb_scn=10)\
    .network()\
        .node('a')\
            .consumption(name='load', cost=10**6, quantity=read_csv('load_A'))\
            .production(name='gas', cost=80, quantity=read_csv('gas'))\
        .node('b').consumption(name='load', cost=10**6, quantity=read_csv('load_B'))\
```

(continues on next page)

(continued from previous page)

```
        .node('c').production(name='nuclear', cost=50, quantity=read_csv('nuclear'))\
        .node('d')\
            .consumption(name='load', cost=10**6, quantity=read_csv('load_D'))\
            .production(name='eolien', cost=20, quantity=read_csv('eolien'))\
        .link(src='a', dest='b', cost=5, quantity=line)\
        .link(src='b', dest='c', cost=5, quantity=line)\
        .link(src='c', dest='a', cost=5, quantity=line)\
        .link(src='c', dest='b', cost=10, quantity=line)\
        .link(src='c', dest='d', cost=10, quantity=line)\
        .link(src='d', dest='c', cost=10, quantity=line)\
    .build()
```

```
optimizer = hd.LPOptimizer()
res = optimizer.solve(study)
```

```
agg = hd.ResultAnalyzer(study, res)
plot = hd.HTMLPlotting(agg=agg, unit_symbol='MW', time_start='2020-06-19', time_end=
↪'2020-06-27',
                       node_coord={'a': [1.6264, 47.8842], 'b': [1.9061, 47.9118], 'c':
↪ [1.6175, 47.7097], 'd': [1.9314, 47.7090]})
```

Let's start by a quick overview of adequacy by plotting a remain available capacity. Blue squares mean network as enough energy to sustain consumption. Red square mean network has a lack of adequacy.

```
plot.network().rac_matrix()
```

As you see it, stochastic is important. Some scenario like 5th is completly success. But if there are more consumption and less production due to unpredictable event, you will have unadequacy.

```
plot.network().node('b').consumption('load').timeline()
```

```
plot.network().node(node='b').stack(scn=7)
```

Hadar can also display valuable information about production. For examples, gas plan seems turn off most of the time

```
plot.network().node('a').production('gas').monotone(scn=7)
```

```
plot.network().node('d').production('eolien').timeline()
```

Then we can plot map to see exchange inside network

```
plot.network().map(t=4, scn=7, zoom=1.6)
```


1.7 Workflow

1.7.1 What is Worflow ?

When you want to simulate adequacy in a network for the next weeks or month, you need to create stochastic study, and generate scenarios (c.f. Begin Stochastic)

Workflow is the preprocessing module for Hadar. Workflow will help user to generate scenarios and sample them to create a stochastic study. It's a toolbox to create pipelines to transform data for optimizer.

With workflow, you will plug stage themself to create pipeline. Stages can already be developed or you can develop your own Stage.

1.7.2 Recreate data used in *Begin Stochastic*

To understand workflow power we will generate data previously used in Begin Stochastic

Build fault pipelines

Let's begin by constant production like nuclear and gas. These productions are not stochastic by default. However fault can occur and it's what we will generate. For this example all stages belongs to hadar ready-to-use library.

```
import hadar as hd
import numpy as np
import pandas as pd
import plotly.graph_objects as go
```

```
# We generate 5 fault scenarios where a fault remove 100 MW with an odd of 1% by␣
↪timestep,
# minimum downtime are one step (one hour) and maximum downtime are 12 step.
fault_pipe = hd.RepeatScenario(n=5) + hd.Fault(loss=300, occur_freq=0.01, downtime_
↪min=1, downtime_max=12) + hd.ToShuffler('quantity')
```

1.7.3 Build stochastic pipelines

In this case, we have to develop our own stage. Let's begin with wind. We know max wind power, we will apply a linear random between 0 to max for each timestep

```
class WindRandom(hd.Stage):
    def __init__(self):
        hd.Stage.__init__(self, plug=hd.FreePlug()) # We will see in other example␣
↪what is FreePlug

    # Method to implement from Stage to create your own Stage with its behaviour
    def _process_timeline(self, timeline: pd.DataFrame) -> pd.DataFrame:
        return timeline * np.random.rand(*timeline.shape)
```

```
wind_pipe = hd.RepeatScenario(n=3) + WindRandom() + hd.ToShuffler('quantity')
```

Then we generate load. For load we will apply a cumulative normal distribution with given value as mean.

```
class LoadRandom(hd.Stage):
    def __init__(self):
        hd.Stage.__init__(self, plug=hd.FreePlug()) # We will see in other example␣
↪what is FreePlug

    # Method to implement from Stage to create your own Stage with its behaviour
    def _process_timeline(self, timeline: pd.DataFrame) -> pd.DataFrame:
        return timeline + np.cumsum(np.random.randn(*timeline.shape) * 10, axis=0)
```

```
load_pipe = hd.RepeatScenario(n=3) + LoadRandom() + hd.ToShuffler('quantity')
```

1.7.4 Generate and sample

We use Shuffler object to generate data by pipeline and then sample 10 scenarios

```
ones = pd.DataFrame({'quantity': np.ones(168)})
# Load are simply a sinus shape
sinus = pd.DataFrame({'quantity': np.sin(np.linspace(-1, -1+np.pi*14, 168))*.2 + .8})

shuffler = hd.Shuffler()
shuffler.add_pipeline(name='gas', data=ones * 1000, pipeline=fault_pipe)
shuffler.add_pipeline(name='nuclear', data=ones * 5000, pipeline=fault_pipe)
shuffler.add_pipeline(name='eolien', data=ones * 1000, pipeline=wind_pipe)
shuffler.add_pipeline(name='load_A', data=sinus * 2000, pipeline=load_pipe)
shuffler.add_pipeline(name='load_B', data=sinus * 3000, pipeline=load_pipe)
shuffler.add_pipeline(name='load_D', data=sinus * 1000, pipeline=load_pipe)
```

```
sampling = shuffler.shuffle(nb_scn=10)
```

```
def input_plot(title, raw, generate):
    x = np.arange(raw.size)
    fig = go.Figure()
    for i, scn in enumerate(generate):
        fig.add_trace(go.Scatter(x=x, y=scn, name='scn %d' % i, line=dict(color=
↪'rgba(100, 100, 100, 0.2)')))

    fig.add_traces(go.Scatter(x=x, y=raw.values.T[0], name='raw'))

    fig.update_layout(title_text=title)
    return fig
```

```
input_plot('Gas', ones * 1000, sampling['gas'])
```

```
input_plot('Nuclear', ones * 5000, sampling['nuclear'])
```

```
input_plot('eolien', ones * 1000, sampling['eolien'])
```

```
input_plot('load_A', sinus * 2000, sampling['load_A'])
```

```
# for name, values in sampling.items():
#     np.savetxt('../Begin Stochastic/%s.csv' % name, values.T, delimiter=' ', fmt='
↪%04.2f')
```

1.8 Workflow Advanced

In Workflow we saw how to easly create simple stage and links stages to build pipeline. It's time to see complet workflow features to create more complex Stage.

1.8.1 Data format

First takes a look at how data are represented inside stage, *a, b, c* are column names provide by user and used by stages:

scn 0

scn 1

scn . . .

t

a

b

. . .

a

b

. . .

a

b

. . .

0

–

–

–

–

–

–

–

–

–

```
<tr>
    <td style="border: 1px solid black">1</td>
    <td style="border: 1px solid black">_</td><td style="border: 1px solid black">_</
↪td><td style="border: 1px solid black">_</td>
    <td style="border: 1px solid black">_</td><td style="border: 1px solid black">_</
↪td><td style="border: 1px solid black">_</td>
    <td style="border: 1px solid black">_</td><td style="border: 1px solid black">_</
↪td><td style="border: 1px solid black">_</td>
</tr>
    <tr>
    <td style="border: 1px solid black">...</td>
    <td style="border: 1px solid black">_</td><td style="border: 1px solid black">_</
↪td><td style="border: 1px solid black">_</td>
    <td style="border: 1px solid black">_</td><td style="border: 1px solid black">_</
↪td><td style="border: 1px solid black">_</td>
```

(continues on next page)

(continued from previous page)

```
    <td style="border: 1px solid black">_</td><td style="border: 1px solid black">_</
↪td><td style="border: 1px solid black">_</td>
</tr>
```

Pipeline could be more flexible, and allow user input without scenarios. Like that, it will be standardized by adding a default 0th scenario.

t

a

b

...

0

_

_

_

```
<tr>
    <td style="border: 1px solid black">1</td>
    <td style="border: 1px solid black">_</td><td style="border: 1px solid black">_</
↪td><td style="border: 1px solid black">_</td>
</tr>
    <tr>
    <td style="border: 1px solid black">...</td>
    <td style="border: 1px solid black">_</td><td style="border: 1px solid black">_</
↪td><td style="border: 1px solid black">_</td>
</tr>
```

1.8.2 Constraint on input and output

As you see above, data contains scenarios and each scenario contains columns with generic names. These names become a constraint. For example some stages expectes to receive strict name, or will produce other columns with new name. Hadar provide a mechanism to handle this complexity called *Plug*. You has already seen `hd.FreePlug` which mean stage has no constraint: It doesn't expected any particular input and doesn't produce specific column.

For example, if you juste need to multiply by twice data, you can create a Stage with `FreePlug`:

```
import hadar as hd
import numpy as np
import pandas as pd
```

```
class Twice(hd.Stage):
    def __init__(self):
        Stage.__init__(plug=hd.FreePlug())
    def _process_timeline(tl):
        return tl * 2
```

It simple, but some time, you expected strictly column name to process timeline. In this case you will use `hd.RestrictedPlug(input, output)`, *input* declare what column names you expected to perform calcul, *output* says what is new column names created during calcul.

Now we care about column name, we often need to apply calcul scenario by scenario and not at the global dataframe. To handle, this mechanism, hadar provides you a `FocusStage` which give you a `_process_scenario(scn, tl)` to implement.

In last example, we created a Stage to generate wind power, just by apply a linear random generation. Now we want more precise generation. Whereas previous stage just use max variables to generate linear random, we use two variables *mean* and *std* to generate normal random.

```
class Wind(hd.FocusStage): # Compute will be done scenario by scenario so we use␣
↪FocusStage
    def __init__(self):
        # Use Restricted plug to force constraint
        hd.FocusStage.__init__(self, plug=hd.RestrictedPlug(inputs=['mean', 'std'],␣
↪outputs=['wind']))

    def _process_scenarios(self, nb_scn, tl):
        return tl['mean'] + np.random.randn(tl.shape[0]) * tl['std']
```

Wind can be plug, upstream stages have to provide *mean* and *std*, downstream stage should use *wind*. For example, `hd.Clip` and `hd.RepeadScenario` are a free plug, you can plug them every where

```
pipe = hd.RepeatScenario(5) + Wind() + hd.Clip(lower=0) # Make sur no negative␣
↪production are generated
```

But if you want to plug `Fault`, error will raise, because `Fault` expectes a *quantity* column

```
try:
    pipe = hd.RepeatScenario(5) +  Wind() + hd.Clip(lower=0) \
         + hd.Fault(occur_freq=0.01, loss=100, downtime_min=1, downtime_max=10)
except ValueError as e:
    print('ValueError:', e)
```

```
ValueError: Pipeline can't be added current outputs are ['wind'] and Fault has input [
↪'quantity']
```

In this case, you can use `hd.Rename` to refix stages with good column name. To summerize pipeline : 1. copy 5 time data in new scenarios 2. apply random generation for each scenarios 3. cap data below 0 (a negativ productoin doesn't exist) 4. Rename data column from *wind* to *quantity* 5. Generate random fault for each scenarios

```
pipe = hd.RepeatScenario(5) +  Wind() + hd.Clip(lower=0) \
     + hd.Rename(wind='quantity') + hd.Fault(occur_freq=0.01, loss=100, downtime_
↪min=1, downtime_max=10)
```

Check is performed when stages are linked together, but also when user give input data. Lines above will raise error since input doesn't have *mean* columns name

```
t = np.linspace(0, 4*3.14, 168)
```

```
try:
    i = pd.DataFrame({'NOT-mean': np.sin(t) * 1000 + 1000, 'std': np.sin(t*2)* 200 +␣
↪200})
    o = pipe(i)
except ValueError as e:
    print('ValueError:', e)
```

```
ValueError: Pipeline accept ['mean', 'std'] in input, but receive ['NOT-mean' 'std']
```

```
i = pd.DataFrame({'mean': np.sin(t) * 1000 + 1000, 'std': np.sin(t*2) * 200 + 200})
o = pipe(i.copy())
```

```
import plotly.graph_objects as go
```

```
fig = go.Figure()

fig.add_traces(go.Scatter(x=t, y=i['mean'], name='mean'))
fig.add_traces(go.Scatter(x=t, y=i['std']+i['mean'], name='std+', line=dict(color='red
↪', dash='dash')))
fig.add_traces(go.Scatter(x=t, y=-i['std']+i['mean'], name='std-', line=dict(color=
↪'red', dash='dash')))
for n in range(5):
    fig.add_traces(go.Scatter(x=t, y=o[n]['quantity'], name='wind %d' % n,␣
↪line=dict(color='rgba(100, 100, 100, 0.5)')))

fig
```

1.9 Storage

We has already seen Consumption, Production and Link to attach on node. Hadar has also a Stockage element. We will work on a simple network with two nodes : one with two producitons (stochastic and constant) other with consumption and stockage

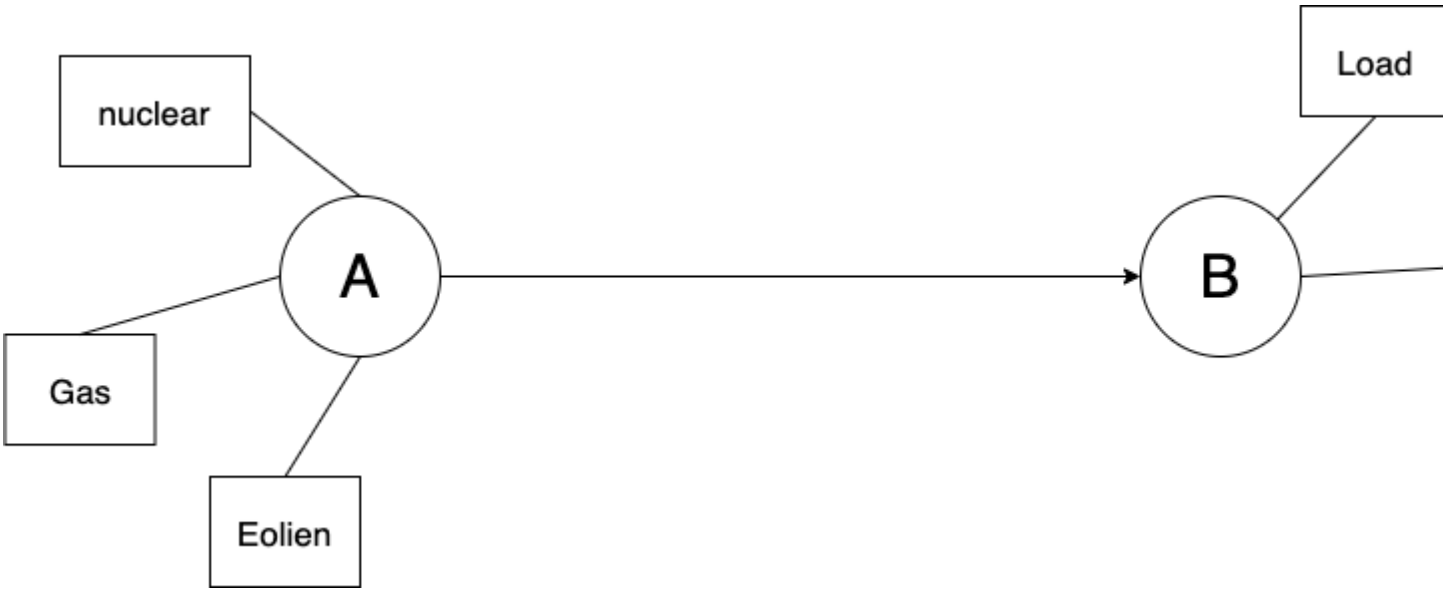


Fig. 1: img

```
import numpy as np
import hadar as hd
```

1.9.1 Create data

```
np.random.seed(12684681)
eolien = np.random.rand(168) * 500 + 200 # random from 200 to 700
load = np.sin(np.linspace(-1, -1+np.pi*14, 168)) * 250 + 750 # sinus moving 500 to␣
↪1000
```

1.9.2 Adequacy without storage

Start storage by remove storage !

```
study = hd.Study(horizon=eolien.size)\
    .network()\
        .node('a')\
            .production(name='gas', cost=100, quantity=200)\
            .production(name='nulcear', cost=50, quantity=300)\
            .production(name='eolien', cost=10, quantity=eolien)\
        .node('b')\
            .consumption(name='load', cost=10 ** 6, quantity=load)\
        .link(src='a', dest='b', cost=1, quantity=2000)\
    .build()

optim = hd.LPOptimizer()
res = optim.solve(study)

plot_without = hd.HTMLPlotting(agg=hd.ResultAnalyzer(study=study, result=res), unit_
↪symbol='MW')
```

```
plot_without.network().node('b').stack()
```

Node B has a lot of lost of load. Network has not enough power to sustain consumption during peak.

```
plot_without.network().node('a').stack()
```

Productions are used immediately just to match load

1.9.3 Use storage

Now we add a storage. In our case cell efficiency is 80%, efficient must be < 1, Hadar use `eff=0.99` as default. Other important parameter is `cost` it represents cost of storage per quantity during on time-step. `cost` at 0 or positive mean we want to minimize storage used. By default Hadar use `cost=0`.

So in the configuration, `cost=0`and `eff=0.80`. Therefore, a quantity stored *costs* 25% ($\frac{1}{0.8} = 1.25$) higher than same production without stored before. At any time Hadar has choice between these productions and cost.

Prod	use cost	stored before use cost
eolien	10	12,5
nuclear	50	62,75
gas	100	125

Moreover than just fix lost of load, storage can also optimize productions. Looks, a stored nuclear or eolien production is cheaper than a direct gas production. Hadar knows it and will use it !

```
study = hd.Study(horizon=eolien.size)\
    .network()\
        .node('a')\
            .production(name='gas', cost=100, quantity=200)\
            .production(name='nulcear', cost=50, quantity=300)\
            .production(name='eolien', cost=10, quantity=eolien)\
        .node('b')\
            .consumption(name='load', cost=10 ** 6, quantity=load)\
            .storage(name='cell', init_capacity=200, capacity=800, flow_in=400, flow_
↪out=400, eff=.8)\
        .link(src='a', dest='b', cost=1, quantity=2000)\
    .build()

res = optim.solve(study)
plot = hd.HTMLPlotting(agg=hd.ResultAnalyzer(study=study, result=res), unit_symbol='MW
↪')
```

```
plot.network().node('b').stack()
```

Yeah ! We avoid network shutdown !

```
plot.network().node('b').storage('cell').candles()
```

Hadar fills cell before each peaks.

```
plot.network().node('a').stack()
```

And yes, Hadars starts nuclear before peak, and use less gas during peak.

1.9.4 Use storage with negative cost

What happen, if we use negative cost ?

In this case, storage has some interest. If interest is higher than gain from optimizing productions. Hadar will automatically fill cell.

```
study = hd.Study(horizon=eolien.size)\
    .network()\
        .node('a')\
            .production(name='gas', cost=100, quantity=200)\
            .production(name='nulcear', cost=50, quantity=300)\
            .production(name='eolien', cost=10, quantity=eolien)\
        .node('b')\
            .consumption(name='load', cost=10 ** 6, quantity=load)\
            .storage(name='cell', init_capacity=200, capacity=800, flow_in=400, flow_
↪out=400, eff=.8, cost=-10)\
        .link(src='a', dest='b', cost=1, quantity=2000)\
    .build()

res = optim.solve(study)
plot_cost_neg = hd.HTMLPlotting(agg=hd.ResultAnalyzer(study=study, result=res), unit_
↪symbol='MW')
```

```
plot_cost_neg.network().node('b').storage('cell').candles()
```

Hadar doesn't try to optimize import, now it saves into storage to earn interest.

1.10 Multi-Energies

Hadar is designed to manage many kind of energy. Indeed, the only restriction is the mathematical equation applies on each node, if user case fill in this equation, Hadar can handle user case.

That means Hadar is not designed for a specific energy. And moreover, Hadar can handle many energies in one study, called *multi-energies*. To do that, Hadar use *network* which organize node inside the same energy. Node inside the same network manage the same energy, we use `Link` to plug them together. If user has many network, therefore many energies, he has to use `Converter`. `Converter` is more powerfull than `Link`, user can specify conversion from many different nodes to one node.

1.10.1 Engine example

Set problem data

No electricity in this tutorial, we will modelize an explosion engine. There are three kind of energies in an engine: oil (*gramme*), compressed air (*gramme*) and work (*Joule*).

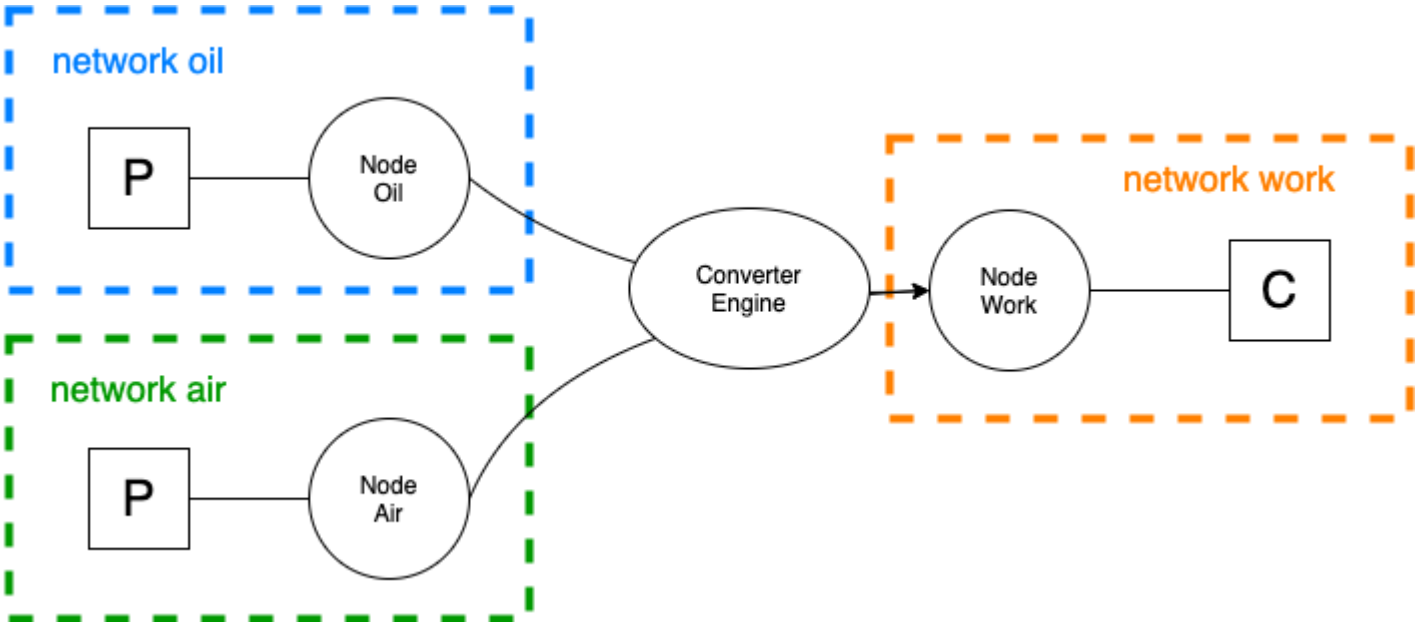


Fig. 2: figure

Data problem: - 1g of oil = 41868J - for engine, ratio oil/air is 15:1, 1g of oil for 15g of air - engine has an efficiency about 36%

Find Hadar ratios

In Hadar, we have to set ratio R_i such as $In_i * R_i = Out$ for each input i.

Equation applies to oil conversion gives:

$$\begin{aligned} & & In_{oil} * R_{oil} &= Work \\ With\ 1g\ of\ oil, & & 1 * R_{oil} &= 41868 * 0,36 \\ & & R_{oil} &= 15072.5 \end{aligned}$$

Equation applies to air conversion gives:

$$
\begin{aligned}
In_{air} * R_{air} &= Work \\
Replace\ with\ first\ equation,\ In_{air} * R_{air} &= In_{oil} * R_{oil} \\
With\ 1g\ of\ oil,\ 15 * R_{air} &= 1 * R_{oil} \\
R_{air} &= R_{oil}/15 \\
R_{air} &= 1005
\end{aligned}
$$

```
import hadar as hd
import numpy as np
```

Work is modellized by a consumption such as $10000 * (1 - e^{-t/25})$

```
work = 10000*(1 - np.exp(-np.arange(100)/25))
```

```
study = hd.Study(horizon=100)\
    .network('work')\
        .node('work')\
            .consumption(name='work', cost=10**6, quantity=work)\
    .network('oil')\
        .node('oil')\
            .production(name='oil', cost=10, quantity=10)\
            .to_converter(name='engine', ratio=15072.5)\
    .network('air')\
        .node('air')\
            .production(name='air', cost=10, quantity=150)\
            .to_converter(name='engine', ratio=1005)\
    .converter(name='engine', to_network='work', to_node='work', max=10000)\
    .build()
```

```
optim = hd.LPOptimizer()
res = optim.solve(study)
```

```
agg = hd.ResultAnalyzer(study=study, result=res)
plot = hd.HTMLPlotting(agg=agg, unit_symbol='J')
```

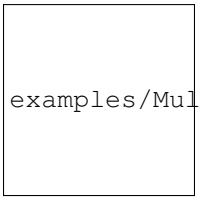
```
plot.network('work').node('work').stack()
```

Work energy comes from engine converter. If we analyze oil and air used in result, we found correct ratio.

```
oil = agg.network('oil').scn(0).node('oil').production('oil').time()['used']
air = agg.network('air').scn(0).node('air').production('air').time()['used']
```

```
(air / oil).plot()
```

```
<AxesSubplot:xlabel='t'>
```



CHAPTER 2

Architecture

2.1 Overview

Welcome to the Hadar Architecture Documentation.

Hadar purpose is to be *an adequacy library for everyone*.

1. Term *everyone* is important, Hadar must be such easy that everyone can use it.
2. And Hadar must be such flexible that everyone business can use it or customize it.

Why these goals ?

We design Hadar in the same spirit of python libraries like numy or scipy, and moreover like scikit-learn. Before scikit-learn, people who want to develop machine learning have to had strong skill in mathematics background to develop their own code. Some *ready to go* codes existed but were not easy to use and flexible.

Scikit-learn release the power of machine learning by abstract complex algorithms into very straight forward API. It was designed like a toolbox to handle full machine learning framework, where user can just assemble scikit-learn component or build their own.

Hadar want to be the next scikit-learn for adequacy. Hadar has to be easy to use and flexible, which if we translate into architecture terms become **high abstraction level** and **independent modules**.

2.1.1 Independent modules

User has the choice : Use only Hadar components, assemble them and create a full solution to generate, solve and analyze adequacy study. Or build their parts.

To reach this constraint, we split Hadar into 4 main modules which can be use together or apart :

- **workflow:** module used to generate data study. Hadar handle deterministic computation like stochastic. For stochastic computation user needs to generate many scenarios. Workflow will help user by providing a highly customizable pipeline framework to transform and generate data.

- **optimizer:** more complex and mathematical module. User will use it to describe study adequacy to resolve. No need to understand mathematics, Hadar will handle data input given and translate it to a linear optimization problem before to call a solver.
- **analyzer:** input data given to optimizer and output date with study result can be heavy to analyze. To avoid that every user build their own toolbox, we develop the most used features once for everyone.
- **viewer** analyzer output will be numpy matrix or pandas Dataframe, it great but not enough to analyze result. Viewer uses the analyzer feature and API to generate graphics from study data.

As said, these modules can be used together to handle complete adequacy study lifecycle or used apart.

TODO graph architecture module

2.1.2 High Abstraction API

Each above modules are like a tiny independent libraries. Therefore each module has a high level API. High abstraction, is a bit confuse to handle and benchmark. For us a high abstraction is when user doesn't need to know mathematics or technicals stuffs when he uses library.

Scikit-learn is the best example of high abstraction level API. For example, if we just want to start a complete SVM research

```
from sklean.svm import SVR
svm = SVR()
svm.fit(X_train, y_train)
y_pred = svm.predict(X_test)
```

How many people using this feature know that scikit-learn tries to project data into higher space to find a linear regression inside. And to accelerate computation, it uses mathematics a feature called *a kernel trick* because problem respect strict requirements ? Perhaps just few people and it's all the beauty of an high level API, it hidden background gear.

Hadar tries to keep this high abstraction features. Look at the Get Started example

```
import hadar as hd

study = hd.Study(horizon=3)\
    .network()\
        .node('a')\
            .consumption(cost=10 ** 6, quantity=[20, 20, 20], name='load')\
            .production(cost=10, quantity=[30, 20, 10], name='prod')\
        .node('b')\
            .consumption(cost=10 ** 6, quantity=[20, 20, 20], name='load')\
            .production(cost=10, quantity=[10, 20, 30], name='prod')\
        .link(src='a', dest='b', quantity=[10, 10, 10], cost=2)\
        .link(src='b', dest='a', quantity=[10, 10, 10], cost=2)\
    .build()

optim = hd.LPOptimizer()
res = optim.solve(study)
```

Create a study like you will draw it on a paper. Put your nodes, attach some production, consumption, link and run optimizer.

Optimizer, Analayzer and Viewer parts are build around the same API called inside code *Fluent API Selector*. Each part has its flavours.

2.1.3 Go Next

Now goals are fixed, we can go deeper into specific module documentation. All architecture focuses on : High Abstraction and Independent module. You can also read the best practices guide to understand more development choice made in Hadar.

Let't start code explanation.

2.2 Workflow

2.2.1 What is a stochastic study ?

Workflow is the preprocessing module for Hadar. It's a toolbox to create pipelines to transform data for optimizer.

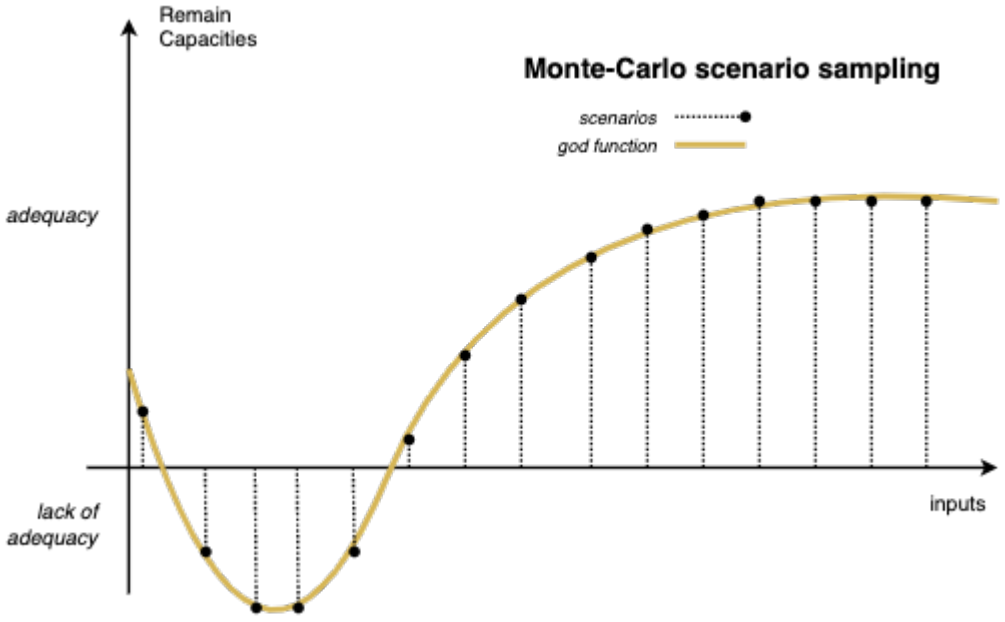
When you want to simulate a network adequacy, you can perform a *deterministic* computation. That means you believe you won't have too much fluky behavior in the future. If you perform adequacy for the next hour or day, it's a good hypothesis. But if you simulate network for the next week, month or year, it's sound curious.

Are you sur wind will blow next week or sun will shines ? If not, you eolian or solar production could change. Can you warrant that no failure will occur on your network next month or next year ?

Of course, we can not predict future with such precision. It's why we use *stochastic* computation. *Stochastic* means there are fluky behavior in the physics we want simulate. Simulation is quiet useless, if result is a unique result.

The best solution could be to compute a *God function* which tell you for each input variation (solar production, line, consumptions) what is the adequacy result. Like that, Hadar has just to analyze function, its derivatives, min, max, etc to predict future. But this *God function* doesn't exist, we just have an algorithm which tell us adequacy according to one fixed set of input data.

It's why we use *Monte Carlo* algorithm. Monte Carlo run many *scenarios* to analyze many different behavior. Scenario with more consumption in cities, less solar production, less coal production or one line deleted due to crash. By this method we recreate *God function* by sampling it with the Monte-Carlo method.



Workflow will help user to generate these scenarios and sample them to create a stochastic study.

The main issue when we want to *help people generating their scenarios* is they are as many generating process as user. Therefore workflow is build upon a Stage and Pipeline Architecture.

2.2.2 Stages, Pipelines & Plug

Stage is an atomic process applied on data. In workflow, data is a pandas Dataframe. Index is time. First column level is for scenario, second is for data (it could be anything like mean, max, sigma, ...). Dataframe is represented below:

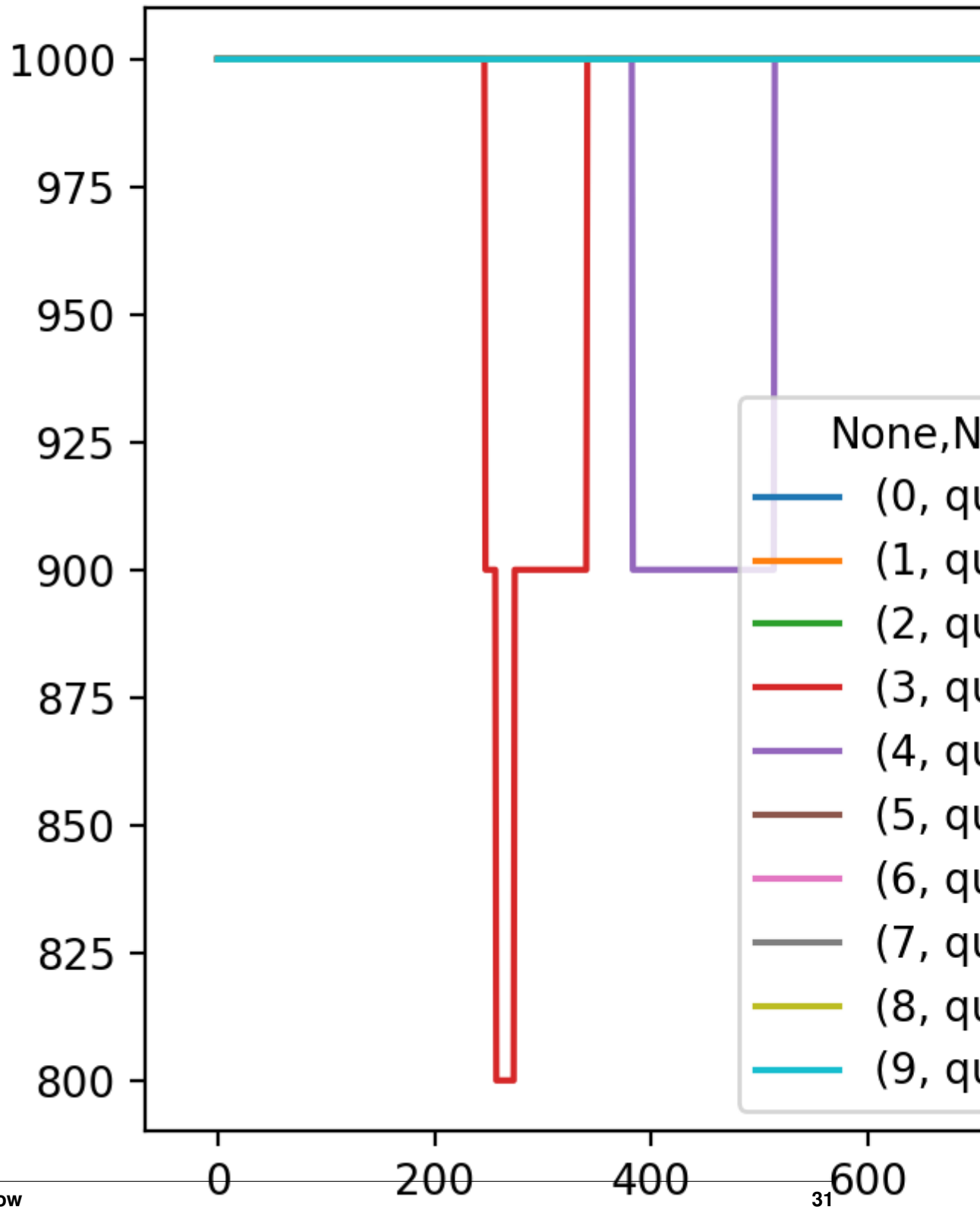
	scn 1				scn n ...			
t	mean	max	min	...	mean	max	min	...
0	10	20	2	...	15	22	8	...
1	12	20	2	...	14	22	8	...
...	...	...	...	...	...	...	...	...

A stage will perform compute to this Dataframe. As you assume it, stages can be linked together to create pipeline. Hadar has its own stages very generic, each user can build these stages and create these pipelines.

For examples, you have many coal production. Each production plan has 10 generators of 100 MW. That means a coal plan production has 1,000 MW of power. You know that sometime, some generators crash or need shutdown for maintenance. With Hadar you can create a pipeline to generate these fault scenarios.

```
# In this example, one timestep = one hour
import hadar as hd
import numpy as np
import hadar as hd
import matplotlib.pyplot as plt

# coal production over 8 weeks with hourly step
coal = pd.DataFrame({'quantity': np.ones(8 * 168) * 1000})

# Copy scenarios ten times
copy = hd.RepeatScenario(n=10)

# Apply on each scenario random fault, such as power drop is 100 MW, there is 0.1%␣
↪chance of failure each hour
# if failure, it's a least for the whole day and until next week.
fault = hd.Fault(loss=100, occur_freq=0.001, downtime_min=24, downtime_max=168)

pipe = copy + fault
out = pipe.compute(coal)

out.plot()
plt.show()
```

Output:


1000
975
950
925
900
875
850
825
800
0
200
400
600
None,N
(0, qu
(1, qu
(2, qu
(3, qu
(4, qu
(5, qu
(6, qu
(7, qu
(8, qu
(9, qu

Create its own Stage

`RepeatScenario`, `Fault` and all other are build upon `Stage` abstract class. A Stage is specified by its `Plug` (we will see sooner) and a `_process_timeline(self, timeline: pd.DataFrame) -> pd.DataFrame` to implement. `timeline` variable inside method is the data passed thought pipeline to transform.

For example, you need to multiply by 2 during your pipeline. You can create your stage by

```
class Twice(Stage):
 def __init__(self):
     Stage.__init__(self, FreePlug())

 def _process_timeline(self, timelines: pd.DataFrame) -> pd.DataFrame:
     return timelines * 2
```

Implement Stage will work every time. Often, you want to apply function independently for each scenario. You can of course handle yourself this mechanism to split current `timeline` apply method and rebuild at the end. Or use `FocusStage`, same thing but already coded. In this case, you need to inherent from `FocusStage` and implement `_process_scenarios(self, n_scn: int, scenario: pd.DataFrame) -> pd.DataFrame` method.

For example, you have thousand of scenarios, your stage has to generate gaussian series according to mean and sigma given.

```
class Gaussian(FocusStage):
    def __init__(self):
        FocusStage.__init__(self, plug=RestrictedPlug(input=['mean', 'sigma'],␣
↪output=['gaussian']))

    def _process_scenarios(self, n_scn: int, scenario: pd.DataFrame) -> pd.DataFrame:
        scenario['gaussian'] = np.random.randn(scenario.shape[0])
        scenario['gaussian'] *= scenario['sigma']
        scenario['gaussian'] += scenario['mean']

        return scenario.drop(['mean', 'sigma'], axis=1)
```

What's Plug ?

You are already see `FreePlug` and `RestrictedPlug`, what's it ?

Stage are linked together to build pipeline. Some Stage accept every thing as input, like `Twice`, but other need specific data like `Gaussian`. How we know that stage can be link together and data given at the beginning of pipeline is correct for all pipeline.

First solution is saying : *We don't care about. During execution, if data is missing, error will be raised and it's enough.* Indeed... That's work, but if pipeline job is heavy, takes hour, and failed just due to a misspelling column name, it's ugly.

`Plug` object describe linkable constraint for Stage and Pipeline. Like Stage, Plug can be added together. In this case, constraint are merged. You can use `FreePlug` telling this Stage is not constraint and doesn't expected any column name to run. Or use `RestrictedPlug(inputs=[], outputs=[])` to specify inputs mandatory columns and new columns generated.

Plug arithmetic rules are described below (∅ = `FreePlug`)

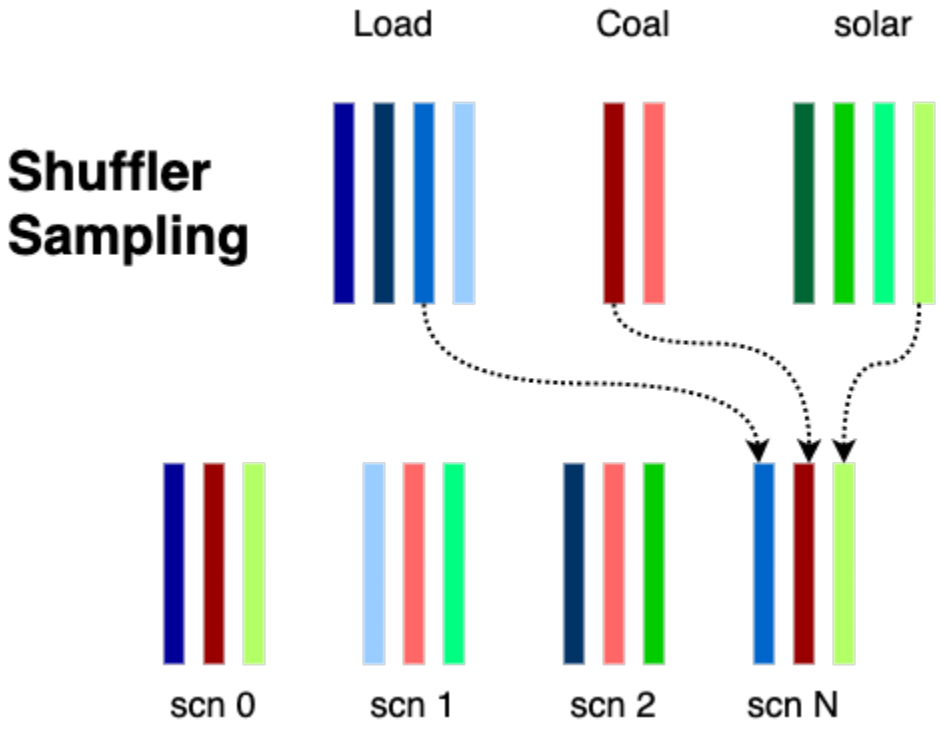
$$
\begin{array}{ccc}
\emptyset & + & \emptyset \\
= & \emptyset a \to \alpha > a \to \alpha & \\
 & + & \emptyset \\
= & [a \to \alpha] a \to \alpha > a \to \alpha & \\
 & + & [\alpha \to A] \\
= & [a \to A] a \to \alpha, \beta > a \to \alpha, \beta & \\
 & + & [\alpha \to A] \\
= & [a \to A, \beta] &
\end{array}
$$

2.2.3 Shuffler

User can create as many pipeline as he want. At the end, he could have some pipelines and input data or directly input data pre-generated. He needs to sampling this dataset to create study. For example, he could have 10 coal generation, 25 solar, 10 consumptions. He needs to create study with 100 scenarios.

Of course he can develop sampling algorithm, but he can also use `Shuffler`. Indeed Shuffler does a bit more than just sampling:

1. It is like a sink where user put pipeline or raw data. Shuffler will homogeneous them to create scenarios. Behind code, we use `Timeline` and `PipelineTimeline` class to homogenize data according to raw data or data from output pipeline.
2. It will schedule pipelines compute. If shuffler is used with pipeline, it will distribute pipeline running over computer cores. A good tips !
3. It samples data to create study scenarios.



Below an example how to use Shuffler

```
shuffler = Shuffler()
# Add raw data as a numpy array
shuffler.add_data(name='solar', data=np.array([[1, 2, 3], [5, 6, 7]]))

# Add pipeline and its input data
i = pd.DataFrame({(0, 'a'): [3, 4, 5], (1, 'a'): [7, 8, 9]})
pipe = RepeatScenario(2) + ToShuffler('a')
shuffler.add_pipeline(name='load', data=i, pipeline=pipe)

# Shuffle to sample 3 scenarios
res = shuffler.shuffle(3)

# Get result according name given
solar = res['solar']
load = res['load']
```

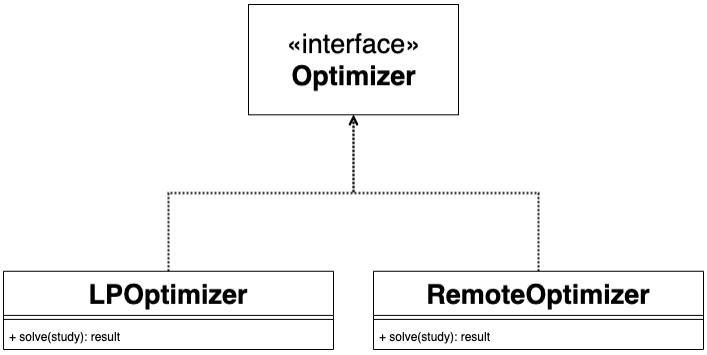
2.3 Optimizer

Optimizer is the heart of Hadar. Behind it, there are :

1. Input object called `Study`. Output object called `Result`. These two objects encapsulate all data needed to compute adequacy.
2. Many optimizers. User can chose which will solve study.

Therefore `Optimizer` is an abstract class build on *Strategy* pattern. User can select optimizer or create their own by implemented `Optimizer.solve(study: Study) -> Result`

Today, two optimizers are present `LPOptimizer` and `RemoteOptimizer`



2.3.1 RemoteOptimizer

Let's start by the simplest. `RemoteOptimizer` is a client to hadar server. As you may know Hadar exist like a python library, but has also a tiny project to package hadar inside web server. You can find more details on this server in this repository.

Client implements `Optimizer` interface. Like that, to deploy compute on a data-center, only one line of code changes.

```
import hadar as hd
# Normal : optim = hd.LPOptimizer()
optim = hd.RemoteOptimizer(host='example.com')
res = optim.solve(study=study)
```

2.3.2 LPOptimizer

Before read this chapter, we kindly advertise you to read *Linear Model*

`LPOptimizer` translate data into optimization problem. Hadar algorithms focus only on modeling problem and uses or-tools to solve problem.

To achieve modeling goal, `LPOptimizer` is designed to receive `Study` object, convert data into or-tools `Variables`. Then `Variables` are placed inside objective and constraint equations. Equations are solved by or-tools. Finally `Variables` are converted to `Result` object.

Analyze that in details.

InputMapper

If you look in code, you will see three domains. One at `hadar.optimizer.input`, `hadar.optimizer.output` and another at `hadar.optimizer.lp.domain` . If you look carefully it seems the same `Consumption` , `OutputConsumption` in one hand, `LPConsumption` in other hand. The only change is a new attribute in `LP*` called `variable` . Variables are the parameters of the problem. It's what or-tools has to find, i.e. power used for production, capacity used for border and lost of load for consumption.

Therefore, `InputMapper` roles are just to create new object with ortools Variables initialized, like we can see in this code snippet.

```
# hadar.optimizer.lp.mapper.InputMapper.get_var
LPLink(dest=l.dest,
       cost=float(l.cost),
       src=name,
       quantity=l.quantity[scn, t],
       variable=self.solver.NumVar(0, float(l.quantity[scn, t]),
          'link on {} to {} at t={} for scn={}'.format(name, l.dest, t, scn)
       )
 )
```

OutputMapper

At the end, `OutputMapper` does the reverse thing. `LP*` objects have computed `Variables`. We need to extract result found by or-tool to `Result` object.

Mapping of `LPProduction` and `LPLink` are straight forward. I propose you to look at `LPConsumption` code

```
self.nodes[name].consumptions[i].quantity[scn, t] =
vars.consumptions[i].quantity - vars.consumptions[i].variable.solution_value()
```

Line seems strange due to complex indexing. First we select good node *name*, then good consumption *i*, then good scenario *scn* and at the end good timestep *t*. Rewriting without index, this line means :

$$Cons_{final} = Cons_{given} - Cons_{var}$$

Keep in mind that $Cons_{var}$ is the lost of load. So we need to subtract it from initial consumption to get really consumption sustained.

Modeler

Hadar has to build problem optimization. These algorithms are encapsulated inside two builders.

`ObjectiveBuilder` takes node by its method `add_node`. Then for all productions, consumptions, links, it adds $variable * cost$ into objective equation.

`StorageBuilder` build constraints for each storage element. Constraints care about a strict volume integrity (i.e. volume is the sum of last volume + input - output)

`ConverterBuilder` build ratio constraints between each inputs converter to output.

`AdequacyBuilder` is a bit more tricky. For each node, it will create a new adequacy constraint equation (c.f. *Linear Model*). Coefficients, here are 1 or -1 depending of *inner* power or *outer* power. Have you seen these line ?

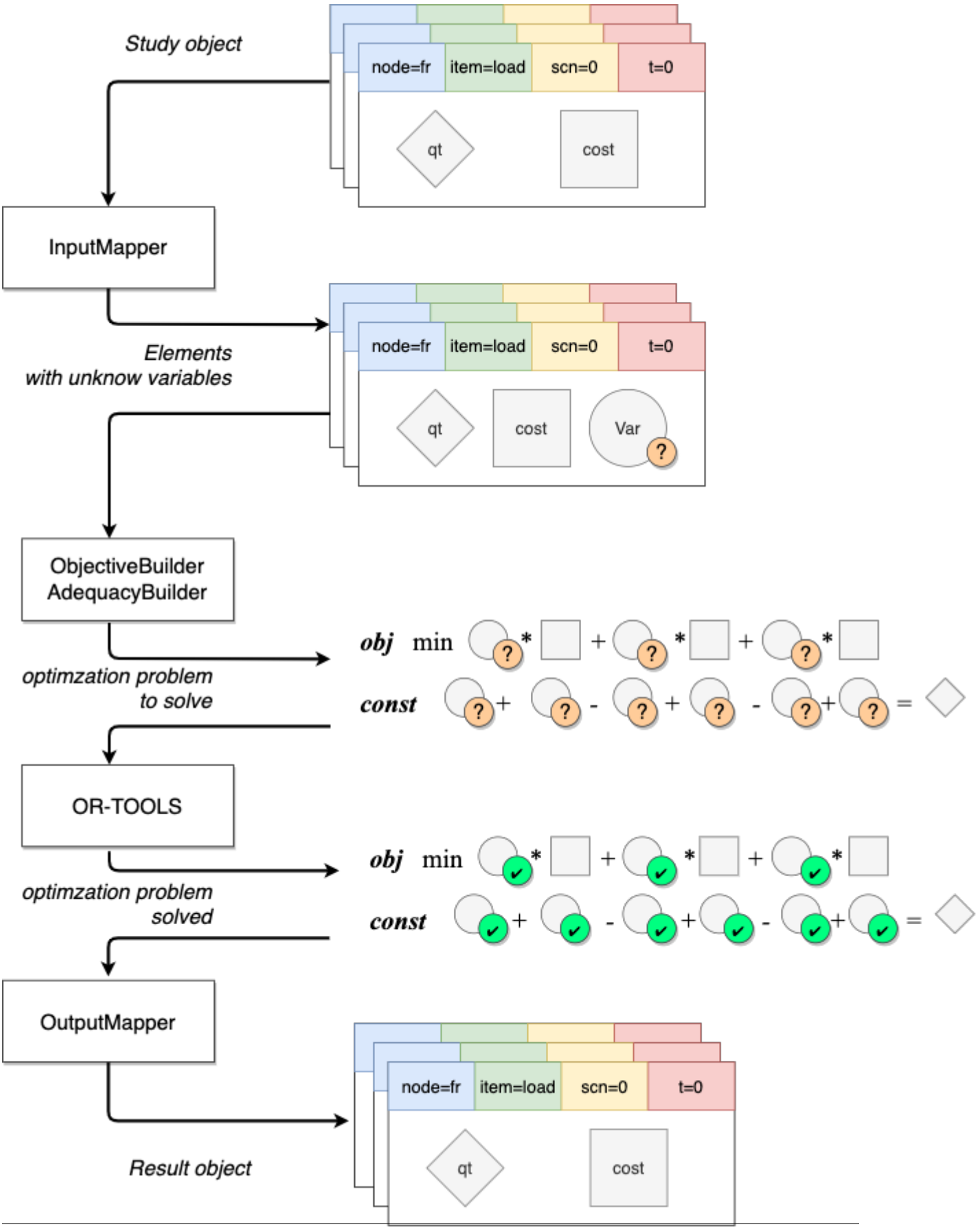
```
self.constraints[(t, link.src)].SetCoefficient(link.variable, -1)  # Export from src
self.importations[(t, link.src, link.dest)] = link.variable  # Import to dest
```

Hadar has to set power importation to *dest* node equation. But maybe this node is not yet setup and its constraint equation doesn't exist yet. Therefore it has to store all constraint equations and all link capacities. And at the end `build()` is called, which will add importation terms into all adequacy constraints to finalize equations.

```
def build(self):
    """
    Call when all node are added. Apply all import flow for each node.

    :return:
    """
    # Apply import link in adequacy
    for (t, src, dest), var in self.importations.items():
        self.constraints[(t, dest)].SetCoefficient(var, 1)
```

`solve_batch` method resolve study for one scenario. It iterates over node and time, calls `InputMapper`, then constructs problem with `*Buidler`, and asks or-tools to solve problem.

`solve_lp` applies the last iteration over scenarios and it's the entry point for linear programming optimizer. After all scenarios are solved, results are mapped to `Result` object.


Study object
node=fr
item=load
scn=0
t=0
qt
cost
InputMapper
Elements
with unknow variables
node=fr
item=load
scn=0
t=0
qt
cost
Var
ObjectiveBuilder
AdequacyBuilder
optimzation problem
to solve
obj min
const
OR-TOOLS
optimzation problem
solved
obj min
const
OutputMapper
Result object
node=fr
item=load
scn=0
t=0
qt
cost

Or-tools, multiprocessing & pickle nightmare

Scenarios are distributed over cores by mutliprocessing library. `solve_batch` is the compute method called by multiprocessing. Therefore all input data received by this method and output data returned must be serializable by pickle (used by multiprocessing). However, output has ortools `Variable` object which is not serializable.

Hadar doesn't need complete `Variable` object. Indeed, it just want value solution found by or-tools. So we will help pickle by creating more simpler object, we carefully recreate same API `solution_value()` to be compliant with downstream code

```
class SerializableVariable(DTO):
    def __init__(self, var: Variable):
        self.val = var.solution_value()

    def solution_value(self):
        return self.val
```

Then specify clearly how to serialize object by implementing `__reduce__` method

```
# hadar.optimizer.lp.domain.LPConsumption
def __reduce__(self):
    """
    Help pickle to serialize object, specially variable object
    :return: (constructor, values...)
    """
    return self.__class__, (self.quantity, SerializableVariable(self.variable), self.
↪cost, self.name)
```

It should work, but in fact not... I don't know why, when multiprocessing want to serialize returned data, or-tools `Variable` are empty, and mutliprocessing failed. Whatever, we just need to handle serialization oneself

```
# hadar.optimizer.lp.solver._solve_batch
return pickle.dumps(variables)
```

2.3.3 Study

code Study' is a *API object* I means it encapsulates all data needed to compute adequacy. It's the glue between workflow (or any other preprocessing) and optimizer. Study has an hierarchical structure of 3 levels :

1. study level with set of networks and converter (`Converter`)
2. network level (`InputNetwork`) with set of nodes.
3. node level (`InputNode`) with set of consumptions, productions, storages and links elements.
4. element level (`Consumption`, `Production`, `Storage`, `Link`). According to element type, some attributes are numpy 2D matrix with shape(nb_scn, horizon)

Most important attribute could be `quantity` which represent quantity of power used in network. For link, is a transfert capacity. For production is a generation capacity. For consumption is a forced load to sustain.

Fluent API Selector

User can construct Study step by step thanks to a *Fluent API* Selector

```
import hadar as hd

study = hd.Study(horizon=3)\
    .network()\
        .node('a')\
            .consumption(cost=10 ** 6, quantity=[20, 20, 20], name='load')\
            .production(cost=10, quantity=[30, 20, 10], name='prod')\
        .node('b')\
            .consumption(cost=10 ** 6, quantity=[20, 20, 20], name='load')\
            .production(cost=10, quantity=[10, 20, 30], name='prod')\
        .link(src='a', dest='b', quantity=[10, 10, 10], cost=2)\
        .link(src='b', dest='a', quantity=[10, 10, 10], cost=2)\
    .build()

optim = hd.LPOptimizer()
res = optim.solve(study)
```

In the case of optimizer, *Fluent API Selector* is represented by `NetworkFluentAPISelector` , and `NodeFluentAPISelector` classes. As you assume with above example, optimizer rules for API Selector are :

- API flow begin by `network()` and end by `build()`
- You can only downstream deeper step by step (i.e. `network()` then `node()`, then `consumption()`)
- But you can upstream as you want (i.e. go direcly from `consumption()` to `network()` or `converter()`)

To help user, quantity and cost fields are flexible:

- lists are converted to numpy array
- if user give a scalar, hadar extends to create (scenario, horizon) matrix size
- if user give (horizon,) matrix or list, hadar copies N time scenario to make (scenario, horizon) matrix size
- if user give (scenario, 1) matrix or list, hadar copies N time timestep to make (scenario, horizon) matrix size

Study includes also check mechanism to be sure: node exist, consumption is unique, etc.

2.3.4 Result

`Result` look like `Study`, it has the same hierarchical structure, same element, just different naming to respect *Domain Driven Development* . Indeed, `Result` is used as output computation, therefore we can't reuse the same object. `Result` is the glue between optimizer and analyzer (or any else postprocessing).

`Result` shouldn't be created by user. User will only read it. So, `Result` has not fluent API to help construction.

2.4 Analyzer

For a high abstraction and to be agnostic about technology, Hadar uses objects as glue for optimizer. Objects are cool, but are too complicated to manipulated for data analysis. Analyzer contains tools to help analyzing study and result.

Today, there is only `ResultAnalyzer`, with two features level:

- **high level** user asks directly to compute global cost and global remain capacity, etc.
- **low level** user build query and get *raw* data represented inside pandas Dataframe.

Before speaking about this features, let's see how data are transformed.

2.4.1 Flatten Data

As said above, object is nice to encapsulate data and represent it into agnostic form. Objects can be serialized into JSON or something else to be used by another software maybe in another language. But keep object to analyze data is awful.

Python has a very efficient tool for data analysis : pandas. Therefore challenge is to transform object into pandas Dataframe. Solution is to flatten data to fill into table.

Consumption

For example with consumption. Data into `Study` is cost and asked quantity. And in `Result` it's cost (same) and given quantity. This tuple *(cost, asked, given)* is present for each node, each consumption attached on this node, each scenario and each timestep. If we want to flatten data, we need to fill this table

cost	asked	given	node	name	scn	t	network
10	5	5	fr	load	0	0	default
10	7	7	fr	load	0	1	default
10	7	5	fr	load	1	0	default
10	6	6	fr	load	1	1	default
...	...	...	...	...		...	...

It is the purpose of `_build_consumption(study: Study, result: Result) -> pd.Dataframe` to build this array

Production

Production follow the same pattern. However, they don't have *asked* and *given* but *available* and *used* quantity. Therefore table looks like

cost	avail	used	node	name	scn	t	network
10	100	21	fr	coal	0	0	default
10	100	36	fr	coal	0	1	default
10	100	12	fr	coal	1	0	default
10	100	81	fr	coal	1	1	default
...	...	...	...	...		...	...

It's done by `_build_production(study: Study, result: Result) -> pd.Dataframe` method.

Storage

Storage follow the same pattern. Therefore table looks like.

max_capacity	avail-capacity	max_flow_fr	flow_in	max_flow_cell	flow_out	cost	init_capacity	eff	node	name	scn	t	network
12000	678	400	214	400	0	10	0	.99	fr	cell	0	0	default
12000	892	400	53	400	0	10	0	.99	fr	cell	0	1	default
12000	945	400	0	400	87	10	0	.99	fr	cell	1	0	default
12000	853	400	0	400	0	10	0	.99	fr	cell	1	1	default
…	…	…	…	…	…	…	…	…	…	…		…	…

It's done by `_build_storage(study: Study, result: Result) -> pd.Dataframe` method.

Link

Link follow the same pattern. Hierarchical structure naming change. There are not *node* and *name* but *source* and *destination*. Therefore table looks like.

cost	avail	used	src	dest	scn	t	network
10	100	21	fr	uk	0	0	default
10	100	36	fr	uk	0	1	default
10	100	12	fr	uk	1	0	default
10	100	81	fr	uk	1	1	default
…	…	…	…	…			…

It's done by `_build_link(study: Study, result: Result) -> pd.Dataframe` method.

Converter

Converter follow the same pattern, it just split in two tables. One for source element:

max	ratio	flow	node	name	scn	t	network
100	.4	52	fr	conv	0	0	default
100	.4	87	fr	conv	0	1	default
100	.4	23	fr	conv	1	0	default
100	.4	58	fr	conv	1	1	default
…	…	…	…	…		…	…

It's done by `_build_src_converter(study: Study, result: Result) -> pd.Dataframe` method.

And an other for destination element, tables are near identical. Source has special attributes called *ratio* and destintion has special attribute called *cost*:

max	cost	flow	node	name	scn	t	network
100	20	52	fr	conv	0	0	default
100	20	87	fr	conv	0	1	default
100	20	23	fr	conv	1	0	default
100	20	58	fr	conv	1	1	default
...	...	...	...	...		...	...

It's done by `_build_dest_converter(study: Study, result: Result) -> pd.Dataframe` method.

2.4.2 Low level analysis power with a *FluentAPISelector*

When you observe flat data, there are two kind of data. *Content* like cost, given, asked and *index* describes by node, name, scn, t.

Low level API analysis provided by `ResultAnalyzer` lets user to

1. Organize index level, for example set time, then scenario, then name, then node.
2. Filter index, for example just time from 10 to 150, just 'fr' node, etc

User can said, *I want 'fr' node productions for first scenario to 50 until 60 timestep.* In this cas `ResultAnalyzer` will return

		used	cost	avail
t	name	21	fr	uk
50	oil	36	fr	uk
	coal	12	fr	uk
60	oil	81	fr	uk
...	...	...	...	...

If first index like node and scenario has only one element, there are removed.

This result can be done by this line of code.

```
agg = hd.ResultAnalyzer(study, result)
df = agg.network().node('fr').scn(0).time(slice(50, 60)).production()
```

For analyzer, Fluent API respect these rules:

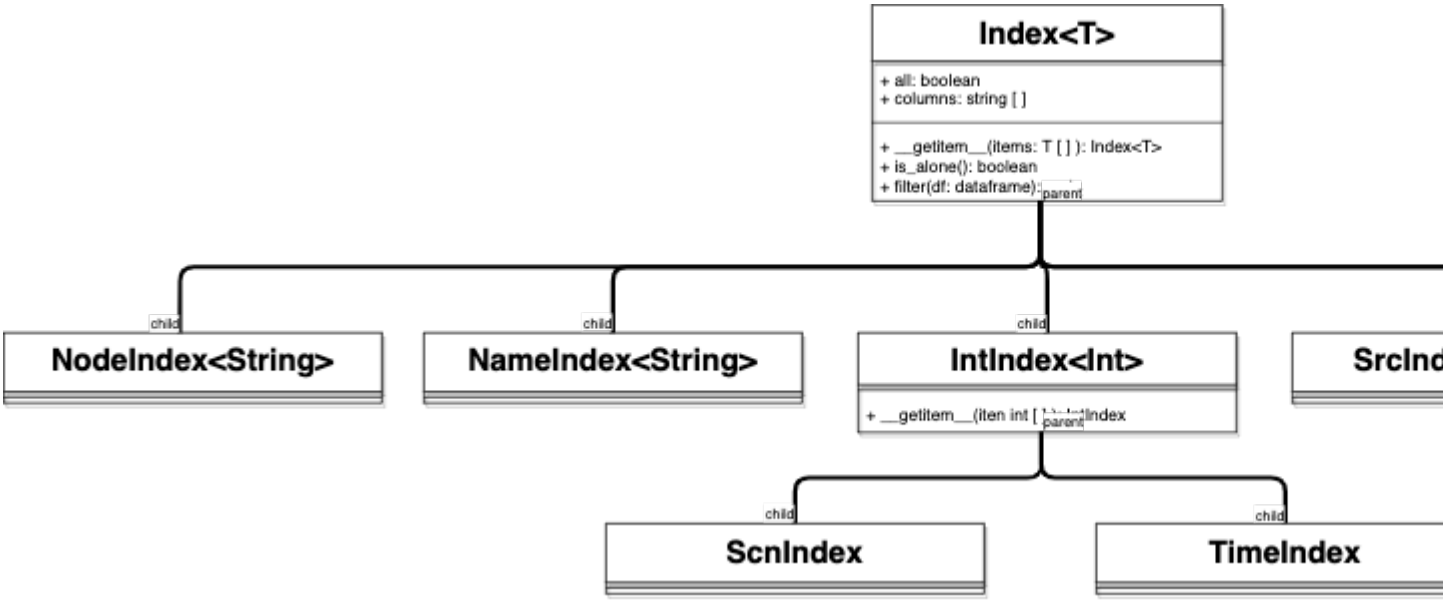
- API flow begin by `network()`
- API flow must contain strictly one of `node()` , `time()`, `scn()` element
- API flow must contain only one of element inside `link()` , `production()` , `consumption()`
- Except for `network()`, API has no order. Order is free for user to give hierarchy data.
- Therefore above rules, API will always be 5 elements length.

Behind this mechanism, there are `Index` objects. As you can see directly in the code

```
...
self.consumption = lambda x=None: self._append(ConsIndex(x))
...
self.time = lambda x=None: self._append(TimeIndex(x))
...
```

Each kind of index has to inherent from this class. `Index` object encapsulate column metadata to use and range of filtered elements to keep (accessible by overriding `__getitem__` method). Then, Hadar has child classes with good parameters : `ConsIndex` , `ProdIndex` , `NodeIndex` , `ScnIndex` , `TimeIndex` , `LinkIndex` , `DestIndex` . For example you can find below `NodeIndex` implementation

```
class NodeIndex(Index[str]):
    """Index implementation to filter nodes"""
    def __init__(self):
        Index.__init__(self, column='node')
```



Index instantiation are completely hidden for user. Then, hadar will

1. check that mandatory indexes are given with `_assert_index` method.
2. pivot table to recreate indexing according to filter and sort asked with `_pivot` method.
3. remove one-size top-level index with `_remove_useless_index_level` method.

As you can see, low level analyze provides efficient method to extract data from adequacy study result. However data returned remains a kind of *roots* and is not ready for business purposes.

2.4.3 High Level Analysis

Unlike low level, high level focus on provides ready to use data. Unlike low level, features should be designed one by one for business purpose. Today we have 2 features:

- `get_cost(self, node: str) -> np.ndarray`: method which according to node given returns a matrix (scenario, horizon) shape with summarize cost.
- `get_balance(self, node: str) -> np.ndarray` method which according to node given returns a matrix (scenario, horizon) shape with exchange balance (i.e. sum of exportation minus sum of importation)

2.5 Viewer

Even with the highest level analyzer features. Data remains simple matrix or tables. Viewer is the end of Hadar framework, it will create amazing plot to bring most valuable data for human analysis.

Viewer use Analyzer API to build plots. It like an extract layer to convert numeric result to visual result.

Viewer is split in two domains. First part implements the *FluentAPISelector*, use ResultAnalyzer to compute result and perform last compute before display graphics. This behaviour are coded inside all `*FluentAPISelector` classes.

These classes are directly used by user when asking for a graphics

```
plot = ...
plot.network().node('fr').consumption('load').gaussian(t=4)
plot.network().map(t=0, scn=0)
plot.network().node('de').stack(scn=7)
```

For Viewer, Fluent API has these rules:

- API begins by `network`.
- User can only go downstream step by step into data. He must specify element choice at each step.
- When he reaches wanted scope (network, node, production, etc), he can call graphics available for the current scope.

Second part belonging to Viewer is only for plotting. Hadar can handle many different libraries and technologies for plotting. New plotting has just to implement `ABCPlotting` and `ABCElementPlotting` . Today one HTML implementation exist with plotly library inside `HTMLPlotting` and `HTMLElementPlotting`.

Data send to plotting classes are complete, pre-computed and ready to display.

CHAPTER 3

Mathematics

3.1 Linear Model

The main optimizer is `LPOptimizer`. It creates linear programming problem representing network adequacy. We will see mathematics problem, step by step

1. Basic adequacy equations
2. Add lack of adequacy terms (lost of load and spillage)

> As you will see, Γ_x represents a quantity in network, $\overline{\Gamma_x}$ is the maximum, $\underline{\Gamma_x}$ is the minimum, $\underline{\overline{\Gamma_x}}$ is the maximum and minimum a.k.a it's a forced quantity. Upper case grec letter is for quantity, and lower case grec letter is for cost γ_x associated to this quantity.

3.1.1 Basic adequacy

Let's begin by the first adequacy behavior. We have a graph $G(N, L)$ with N nodes on the graph and L unidirectional edges on this graph.

Variables

- $n \in N$ a node belongs to graph
- $T \in \mathbb{Z}_+$ time horizon

Edge variables

- $l \in L$ an unidirectional edge belongs to graphs
- $\overline{\Gamma_l} \in \mathbb{R}_+^T$ maximum power transfert capacity for l
- $\Gamma_l \in \mathbb{R}_+^T$ power transfered inside l
- $\gamma_l \in \mathbb{R}_+^T$ proportional cost when Γ_l is used
- $L_\uparrow^n \subset L$ set of edges with direction to node n (i.e. importation for n)

- $L^n_{\downarrow} \subset L$ set of edges with direction from node n (i.e. exportation for n)

Productions variables

- P^n set of productions attached to node n
- $p \in P^n$ a production inside set of productions attached to node n
- $\overline{\Gamma_p} \in \mathbb{R}^T_+$ maximum power capacity available for p production.
- $\Gamma_p \in \mathbb{R}^T_+$ power capacity of p used during adequacy
- $\gamma_p \in \mathbb{R}^T_+$ proportional cost when Γ_p is used

Consumptions variables

- C^n set of consumptions attached to node n
- $c \in C^n$ a consumption inside set of consumptions attached to node n
- $\underline{\overline{\Gamma_c}} \in \mathbb{R}^T_+$ forced consumptions of c to sustain.

Objective

$$\begin{aligned} objective &= \min \Omega_{transmission} + \Omega_{production} \\ \Omega_{transmission} &= \sum_l^L \Gamma_l * \gamma_l \\ \Omega_{production} &= \sum_n^N \sum_p^{P^n} \Gamma_p * \gamma_p \end{aligned}$$

Constraint

First constraint is from Kirschhoff law and describes balance between productions and consumptions

$$\Pi_{kirschhoff} : \forall n$$
$$, \quad \underbrace{\sum_c^{C^n} \underline{\overline{\Gamma_c}} + \sum_l^{L^n_{\downarrow}} \Gamma_l}_{Consuming\ Flow} = \underbrace{\sum_p^{P^n} \Gamma_p + \sum_l^{L^n_{\uparrow}} \Gamma_l}_{Producing\ Flow}$$

Then productions and edges need to be bounded

$$\Pi_{Edge\ bound} : \forall l \in L$$
$$, \quad 0 \leq \Gamma_l \leq \overline{\Gamma_l}$$
$$\Pi_{Prod\ bound} : \begin{cases} \forall n \in N \\ \forall p \in P^n \end{cases}$$
$$, \quad 0 \leq \Gamma_p \leq \overline{\Gamma_p}$$

3.1.2 Lack of adequacy

Variables

Sometime, there are a lack of adequacy because there are not enough production, called *lost of load*.

> Like Γ_x means quantity present in network, Λ_x represents a lack in network (consumption or production) to reach adequacy. Like for Γ_x , lower case grec letter λ_x is for cost associated to this lack.

- $\Lambda_c \in \mathbb{R}^T_+$ lost of load for c consumption
- $\lambda_c \in \mathbb{R}^T_+$ proportional cost when Λ_c is used

Objective

Objective has a new term

$$\begin{aligned} objective &= \min \Omega_{...} + \Omega_{lol} \\ \Omega_{lol} &= \sum_{n}^{N} \sum_{c}^{C^n} \Lambda_c * \lambda_c \end{aligned}$$

Constraints

Kirschhoff law needs an update too. Lost of Load is represented like a *fantom* import of energy to reach adequacy.

$$\Pi_{kirschhoff} : \forall n \in N \\ , \quad [Consuming\ Flow] = [Producing\ Flow] + \sum_{c}^{C^n} \Lambda_c$$

Lost of load must be bounded

$$\Pi_{Lol\ bound} : \begin{cases} \forall n \in N \\ \forall c \in C^n \end{cases} \\ , \quad 0 \leq \Lambda_c \leq \overline{\Gamma_c}$$

3.1.3 Storage

Variables

Storage is a element inside Hadar to store quantity on a node. We have:

- S^n : set of storage attached to node n
- $s \in S^n$ a storage element inside a set of storage attached to node n
- Γ_s current capacity inside storage s
- $\overline{\Gamma_s}$ max capacity for storage s
- Γ_s^0 initial capacity inside storage s
- γ_s linear cost of capacity storage s for one time step
- $\Gamma_s^{\downarrow}$ input flow to storage s
- $\overline{\Gamma_s^{\downarrow}}$ max input flow to storage s
- $\Gamma_s^{\uparrow}$ output flow to storage s
- $\overline{\Gamma_s^{\uparrow}}$ max output flow to storage s
- η_s storage efficiency for s

Objective

$$\begin{aligned} objective &= \min \Omega_{...} + \Omega_{storage} \\ \Omega_{storage} &= \sum_{n}^{N} \sum_{s}^{S^n} \Gamma_s * \gamma_s \end{aligned}$$

Constraints

Kirschhoff law needs an update too. **Warning with naming** : Input flow for storage is a output flow for node, so goes into consuming flow. And as you assume output flow for storage is a input flow for node, and goes into production flow.

$$\Pi_{kirschhoff} : \forall n \in N$$
$$, \quad [Consuming\ Flow] + \sum_{s}^{S^n} \Gamma_s^{\downarrow} = [Producing\ Flow] + \sum_{s}^{S^n} \Gamma_s^{\uparrow}$$

And all these things are bounded :

$$\Pi_{Store\ bound} : \begin{cases} \forall n \in N \\ \forall s \in S^n \end{cases}$$
$$, \quad \begin{array}{ccc} 0 & \leq & \Gamma_s \\ \leq & \overline{\Gamma_s} & \\ 0 & \leq & \Gamma_s^{\downarrow} \\ \leq & \overline{\Gamma_s^{\downarrow}} & \\ 0 & \leq & \Gamma_s^{\uparrow} \\ \leq & \overline{\Gamma_s^{\uparrow}} & \end{array}$$

Storage has also a new constraint. This constraint applies over time to ensure capacity integrity.

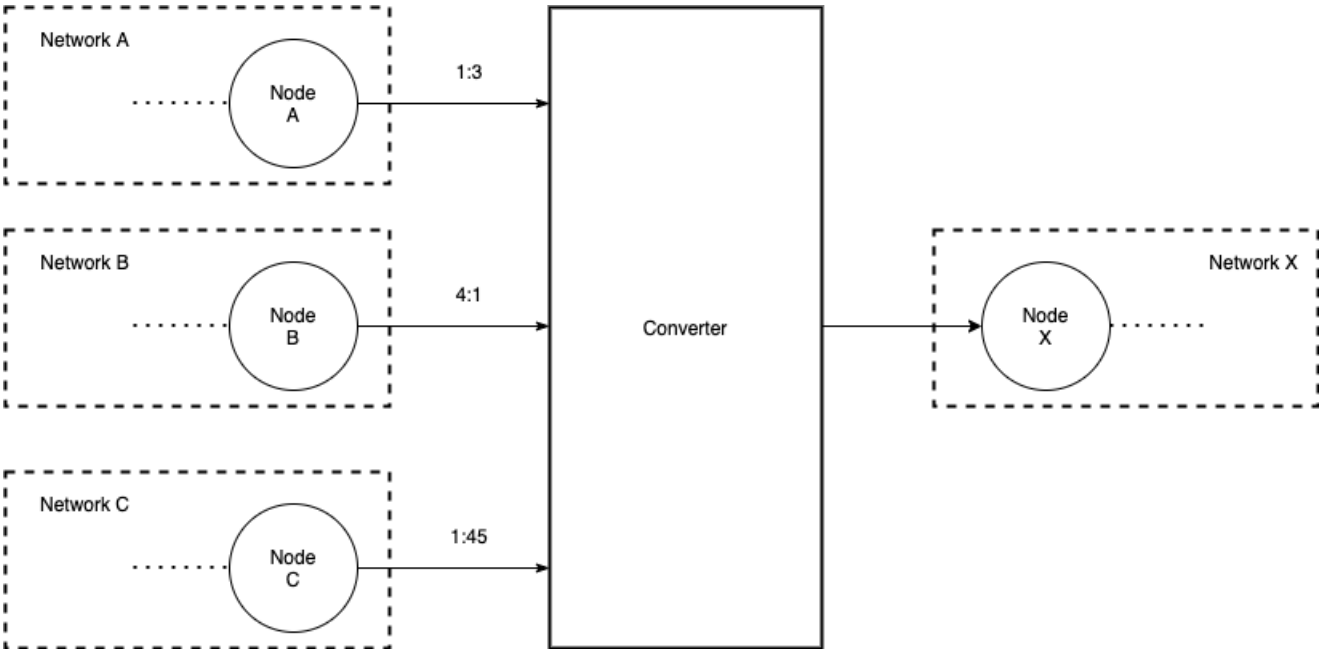
$$\Pi_{storage} : \begin{cases} \forall n \in N \\ \forall s \in S^n \\ \forall t \in T \end{cases}$$
$$, \quad \Gamma_s[t] = \begin{vmatrix} \Gamma_s[t-1] \\ \Gamma_s^0,\ t = 0 \end{vmatrix} + \Gamma_s^{\downarrow}[t] * \eta_s - \Gamma_s^{\uparrow}[t]$$

3.1.4 Multi-Energies

Hadar handle multi-energies. In the code, one energy lives inside one network. Multi-energies means multi-networks. Mathematically, there are all the same. That why we don't talk about multi graph, there are always one graph G, nodes remains the same, with same equation for every kind of energies.

The only difference is how we link node together. If nodes belongs to same network, we use *link (or edge)* seen before. When nodes belongs to different energies we need to use *converter*. All things above remains true, we just add now a new element V converters ont this graph $G(N, L, V)$.

Converter can take energy form many nodes in different network. Each converter input has a ratio between output quantity and input quantity. Converter has only one output to only on node.



Variables

- V set of converters
- $v \in V$ a converter in the set of converters
- $V^n_{\uparrow} \subset V$ set of converters **to** node n
- $V^n_{\downarrow} \subset V$ set of converters **from** node n
- $\Gamma^{\uparrow}_v$ flow **from** converter v.
- $\overline{\Gamma^{\uparrow}_v}$ max flow from converter v
- γ_v linear cost when $\Gamma^{\uparrow}_v$ is used
- $\Gamma^{\downarrow}_v$ flow(s) **to** converter. They can have many flows for $v \in V$, but only one for $v \in V^n_{\downarrow}$
- $\overline{\Gamma^{\downarrow}_v}$ max flow to converter
- α^n_v ratio conversion for converter v from node n

Objective

$$\begin{aligned} objective &= \min \Omega_{...} + \Omega_{converter} \\ \Omega_{converter} &= \sum_v^V \Gamma^{\uparrow}_v * \gamma_v \end{aligned}$$

Constraints

Of course Kirschhoff need a little update. Like for storage **Warning with naming !** Converter input is a consuming flow for node, converter output is a production flow for node.

$$\begin{aligned} \Pi_{kirschhoff} &: && \forall n \in N \\ , \quad & [Consuming\ Flow] + \sum_v^{V^n_{\downarrow}} \Gamma^{\downarrow}_v = [Producing\ Flow] + \sum_v^{V^n_{\uparrow}} \Gamma^{\uparrow}_v \end{aligned}$$

And all these things are bounded :

$$\Pi_{Conv\ bound} \quad : \quad \begin{cases} \forall n \in N \\ \forall v \in V^n \end{cases}$$
$$\begin{array}{rcl} 0 & \leq & \Gamma_v^{\downarrow} \\ \leq & & \overline{\Gamma_v^{\downarrow}} \\ , \quad 0 & \leq & \Gamma_v^{\uparrow} \\ \leq & & \overline{\Gamma_v^{\uparrow}} \end{array}$$

Now, we need to fix ratios conversion by a new constraints

$$\Pi_{converter} \quad : \quad \begin{cases} \forall n \in N \\ \forall v \in V_{\downarrow}^n \end{cases}$$
$$, \quad \Gamma_v^{\downarrow} * \alpha_v^n \quad = \quad \Gamma_v^{\uparrow}$$

CHAPTER 4

Contributing

4.1 How to Contribute

First off, thank you to considering contributing to Hadar. We believe technology can change the world. But only great community and open source can improve the world.

Following these guidelines helps to communicate that you respect the time of the developers managing and developing this open source project. In return, they should reciprocate that respect in addressing your issue, assessing changes, and helping you finalize your pull requests.

We try to describe most of Hadar behavior and organization to avoid any *shadow part*. Additionally, you can read *Dev Guide* section or *Architecture* to learn hadar purposes and processes.

4.1.1 What kind of contribution ?

You can participate on Hadar from many ways:

- just use it and spread it !
- write plugin and extension for hadar
- Improve docs, code, examples
- Add new features

Issue tracker are only for features, bug or improvment; not for support. If you have some question please go to TODO . Any support issue will be closed.

4.1.2 Feature / Improvement

Little changes can be directly send into a pull request. Like :

- Spelling / grammar fixes
- Typo correction, white space and formatting changes

- Comment clean up
- Adding logging messages or debugging output

For all other, you need first to create an issue. If issue receives good feedback. Then you can fork project, work on your side and send a Pull Request

4.1.3 Bug

If you find a security bug, please DON'T create an issue. Contact use at admin@hadar-simulator.org

First be sure it's a bug and not a misuse ! Issues are not for technical support. To speed up bug fixing (and avoid misuse), you need to clearly explain bug, with most simple step by step guide to reproduce bug. Specify us all details like OS, Hadar version and so on.

Please provide us response to these questions

```
- What version of Hadar and python are you using ?
- What operating system and processor architecture are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?
```

4.1.4 Best Practices

We try to code the most clear and maintainable software. Your Pull Request has to follow some good practices:

- respect PEP 8 style guide
- name meaningful variables, method, class
- respect SOLID , KISS , DRY , YAGNI principe
- make code easy testable (use dependencies injection)
- test code (at least 80% UT code coverage)
- Add docstring for each class and method.

TL;TR: code as Uncle Bob !

4.2 Repository Organization

Hadar repository is split in many parts.

- `hadar/` source code
- `tests/` unit and integration tests perform by unittest
- `examples/` set of notebooks used like End to End test when executed during CI or like tutorials when exported to html.
- `docs/` sphinx documentation hosted by readthedocs at https://docs.hadar-simulator.org . Main website is hosted by Github Pages and source code can be find in this repository
- `.github/` github configuration to use Github Action for CI.

4.2.1 Ticketing

We use all github features to organize development. We implement a Agile methodology and try to recreate Jira behavior in github. Therefore we swap Jira features to Github such as :

Jira	github swap
User Story / Bug	Issue
Version = Sprint	Project
task	check list in issue
Epic	Milestone

4.2.2 Devops

We respect *git flow* pattern. Main developments are on `develop` branch. We accept `feature/**` branch but is not mandatory.

CI pipelines are backed on *git flow*, actions are sum up in table below :

action	develop	release/**	master
TU + IT	3.6, 3.7, 3.8 / linux, mac, win	linux-3.7	linux-3.7
E2E		from source code	from test.pypip.org
Sonar	yes	yes	yes
package		to test.pypip.org	to pypip.org

CHAPTER 5

Reference

5.1 hadar package

5.1.1 Subpackages

hadar.analyzer package

Submodules

hadar.analyzer.result module

class hadar.analyzer.result.**ResultAnalyzer**(*study: hadar.optimizer.domain.input.Study*, *result: hadar.optimizer.domain.output.Result*)

Bases: object

Single object to encapsulate all postprocessing aggregation.

static check_index(*indexes: List[hadar.analyzer.result.Index]*, *type: Type[CT_co]*)
Check indexes cohesion :param indexes: list fo indexes :param type: Index type to check inside list :return: true if at least one type is in list False else

filter(*indexes: List[hadar.analyzer.result.Index]*) → pandas.core.frame.DataFrame
Aggregate according to index level and filter.

get_balance(*node: str*, *network: str = 'default'*) → numpy.ndarray
Compute balance over time on asked node.

Parameters

- **node** – node asked
- **network** – network asked. Default is 'default'

Returns timeline array with balance exchanges value

get_cost(*node: str = None*, *network: str = None*) → numpy.ndarray
Compute adequacy cost on a node, network or whole study.

Parameters

- **node** – node name. None by default to ask whole network.
- **network** – network name, 'default' as default if node is provided or None to ask whole network.

Returns matrix (scn, time)

get_elements_inside(*node: str = None*, *network: str = None*)
Get numbers of elements by node.

Parameters

- **node** – node name. None by default to ask whole network.
- **network** – network name, 'default' as default if node is provided or None to ask whole network.

Returns (nb of consumptions, nb of productions, nb of storages, nb of links (export), nb of converters (export), nb of converters (import)

get_rac(*network='default'*) → numpy.ndarray
Compute Remain Availabale Capacities on network.

Parameters **network** – selecto network to compute. Default is default.

Returns matrix (scn, time)

horizon
Shortcut to get study horizon.

Returns study horizon

nb_scn
Shortcut to get study number of scenarios.

Returns study number of scenarios

network(*name='default'*)
Entry point for fluent api :param name: network name. 'default' as default :return: Fluent API Selector

nodes(*network: str = 'default'*) → List[str]
Shortcut to get list of node names

Parameters **network** – network selected

Returns nodes name

class hadar.analyzer.result.**NetworkFluentAPISelector**(*indexes: List[hadar.analyzer.result.Index]*, *analyzer: hadar.analyzer.result.ResultAnalyzer*)

Bases: object

Fluent Api Selector to analyze network element.

User can join network, node, consumption, production, link, time, scn to create filter and organize hierarchy. Join can me in any order, except: - join begin by network - join is unique only one element of node, time, scn are expected for each query - production, consumption and link are excluded themself, only on of them are expected for each query

FULL_DESCRIPTION = 5

Module contents

hadar.optimizer package

Subpackages

hadar.optimizer.domain package

Submodules

hadar.optimizer.domain.input module

class hadar.optimizer.domain.input.**Consumption**(*quantity: hadar.optimizer.domain.numeric.NumericalValue*, *cost: hadar.optimizer.domain.numeric.NumericalValue*, *name: str = ''*)

Bases: *hadar.optimizer.utils.JSON*

Consumption element.

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.input.**Link**(*dest: str*, *quantity: hadar.optimizer.domain.numeric.NumericalValue*, *cost: hadar.optimizer.domain.numeric.NumericalValue*)

Bases: *hadar.optimizer.utils.JSON*

Link element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.input.**Production**(*quantity: hadar.optimizer.domain.numeric.NumericalValue*, *cost: hadar.optimizer.domain.numeric.NumericalValue*, *name: str = 'in'*)

Bases: *hadar.optimizer.utils.JSON*

Production element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.input.**Storage**(*name*, *capacity: hadar.optimizer.domain.numeric.NumericalValue*, *flow_in: hadar.optimizer.domain.numeric.NumericalValue*, *flow_out: hadar.optimizer.domain.numeric.NumericalValue*, *cost: hadar.optimizer.domain.numeric.NumericalValue*, *init_capacity: int*, *eff: hadar.optimizer.domain.numeric.NumericalValue*)

Bases: *hadar.optimizer.utils.JSON*

Storage element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.input.**Converter**(*name: str, src_ratios: Dict[Tuple[str, str], hadar.optimizer.domain.numeric.NumericalValue], dest_network: str, dest_node: str, cost: hadar.optimizer.domain.numeric.NumericalValue, max: hadar.optimizer.domain.numeric.NumericalValue*)

Bases: *hadar.optimizer.utils.JSON*

Converter element

static from_json(*dict: dict*, *factory=None*)

to_json() → dict

class hadar.optimizer.domain.input.**InputNetwork**(*nodes: Dict[str, hadar.optimizer.domain.input.InputNode] = None*)

Bases: *hadar.optimizer.utils.JSON*

Network element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.input.**InputNode**(*consumptions: List[hadar.optimizer.domain.input.Consumption], productions: List[hadar.optimizer.domain.input.Production], storages: List[hadar.optimizer.domain.input.Storage], links: List[hadar.optimizer.domain.input.Link]*)

Bases: *hadar.optimizer.utils.JSON*

Node element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.input.**Study**(*horizon: int*, *nb_scn: int = 1*, *version: str = None*)

Bases: *hadar.optimizer.utils.JSON*

Main object to facilitate to build a study

add_link(*network: str, src: str, dest: str, cost: Union[List[T], numpy.ndarray, float], quantity: Union[List[T], numpy.ndarray, float]*)

Add a link inside network.

Parameters

- **network** – network where nodes belong
- **src** – source node name
- **dest** – destination node name
- **cost** – cost of use
- **quantity** – transfer capacity

Returns

add_network(*network: str*)

add_node(*network: str*, *node: str*)

static from_json(*dict*, *factory=None*)

network(*name='default'*)

Entry point to create study with the fluent api.

Returns

to_json()

class hadar.optimizer.domain.input.**NetworkFluentAPISelector**(*study*, *selector*)

Bases: object

Network level of Fluent API Selector.

build()
Build study.

Returns return study

converter(*name: str, to_network: str, to_node: str, max: Union[List[T], numpy.ndarray, float], cost: Union[List[T], numpy.ndarray, float] = 0*)
Add a converter element.

Parameters

- **name** – converter name
- **to_network** – converter output network
- **to_node** – converter output node on network
- **max** – maximum quantity from converter
- **cost** – cost for each quantity produce by converter

Returns

link(*src: str, dest: str, cost: Union[List[T], numpy.ndarray, float], quantity: Union[List[T], numpy.ndarray, float]*)
Add a link on network.

Parameters

- **src** – node source
- **dest** – node destination
- **cost** – unit cost transfer
- **quantity** – available capacity

Returns NetworkAPISelector with new link.

network(*name='default'*)
Go to network level.

Parameters **name** – network level, 'default' as default name

Returns NetworkAPISelector with selector set to 'default'

node(*name*)
Go to node level.

Parameters **name** – node to select when changing level

Returns NodeFluentAPISelector initialized

class hadar.optimizer.domain.input.**NodeFluentAPISelector**(*study*, *selector*)
Bases: object

Node level of Fluent API Selector

build()
Build study.

Returns study

consumption(*name: str, cost: Union[List[T], numpy.ndarray, float], quantity: Union[List[T], numpy.ndarray, float]*)
Add consumption on node.

Parameters

- **name** – consumption name
- **cost** – cost of unsuitability
- **quantity** – consumption to sustain

Returns NodeFluentAPISelector with new consumption

converter(*name: str, to_network: str, to_node: str, max: Union[List[T], numpy.ndarray, float], cost: Union[List[T], numpy.ndarray, float] = 0*)
Add a converter element.

Parameters

- **name** – converter name
- **to_network** – converter output network
- **to_node** – converter output node on network
- **max** – maximum quantity from converter
- **cost** – cost for each quantity produce by converter

Returns

link(*src: str, dest: str, cost: int, quantity: Union[List[T], numpy.ndarray, float]*)
Add a link on network.

Parameters

- **src** – node source
- **dest** – node destination
- **cost** – unit cost transfer
- **quantity** – available capacity

Returns NetworkAPISelector with new link.

network(*name='default'*)
Go to network level.

Parameters **name** – network level, 'default' as default name

Returns NetworkAPISelector with selector set to 'default'

node(*name*)
Go to different node level.

Parameters **name** – new node level

Returns NodeFluentAPISelector

production(*name: str, cost: Union[List[T], numpy.ndarray, float], quantity: Union[List[T], numpy.ndarray, float]*)
Add production on node.

Parameters

- **name** – production name
- **cost** – unit cost of use
- **quantity** – available capacities

Returns NodeFluentAPISelector with new production

storage(*name, capacity: Union[List[T], numpy.ndarray, float], flow_in: Union[List[T], numpy.ndarray, float], flow_out: Union[List[T], numpy.ndarray, float], cost: Union[List[T], numpy.ndarray, float] = 0, init_capacity: int = 0, eff: Union[List[T], numpy.ndarray, float] = 0.99*)

Create storage.

Parameters

- **capacity** – maximum storage capacity (like of many quantity to use inside storage)
- **flow_in** – max flow into storage during on time step
- **flow_out** – max flow out storage during on time step
- **cost** – unit cost of storage at each time-step. default 0
- **init_capacity** – initial capacity level. default 0
- **eff** – storage efficient (applied on input flow stored). default 0.99

to_converter(*name: str, ratio: Union[List[T], numpy.ndarray, float] = 1*)

Add an ouptput to converter.

Parameters

- **name** – converter name
- **ratio** – ratio for output

Returns

hadar.optimizer.domain.numeric module

class hadar.optimizer.domain.numeric.**ColumnNumericValue**(*value: T, horizon: int, nb_scn: int*)

Bases: *hadar.optimizer.domain.numeric.NumpyNumericalValue*

Implementation with one time step by scenario with shape (nb_scn, 1)

flatten() → numpy.ndarray

flat data into 1D matrix. :return: [v[0, 0], v[0, 1], v[0, 2], ..., v[1, i], v[2, i], ..., v[j, i])

static from_json(*dict*)

class hadar.optimizer.domain.numeric.**MatrixNumericalValue**(*value: T, horizon: int, nb_scn: int*)

Bases: *hadar.optimizer.domain.numeric.NumpyNumericalValue*

Implementation with complex matrix with shape (nb_scn, horizon)

flatten() → numpy.ndarray

flat data into 1D matrix. :return: [v[0, 0], v[0, 1], v[0, 2], ..., v[1, i], v[2, i], ..., v[j, i])

static from_json(*dict*)

class hadar.optimizer.domain.numeric.**NumericalValue**(*value: T, horizon: int, nb_scn: int*)

Bases: *hadar.optimizer.utils.JSON*, abc.ABC, typing.Generic

Interface to handle numerical value in study

flatten() → numpy.ndarray

flat data into 1D matrix. :return: [v[0, 0], v[0, 1], v[0, 2], ..., v[1, i], v[2, i], ..., v[j, i])

class hadar.optimizer.domain.numeric.**NumericalValueFactory**(*horizon: int*, *nb_scn: int*)

Bases: object

create(*value: Union[float, List[float], str, numpy.ndarray, hadar.optimizer.domain.numeric.NumericalValue]*) → hadar.optimizer.domain.numeric.NumericalValue

class hadar.optimizer.domain.numeric.**NumpyNumericalValue**(*value: T*, *horizon: int*, *nb_scn: int*)

Bases: *hadar.optimizer.domain.numeric.NumericalValue*, abc.ABC

Half-implementation with numpy array as numerical value. Implement only compare methods.

class hadar.optimizer.domain.numeric.**RowNumericValue**(*value: T*, *horizon: int*, *nb_scn: int*)

Bases: *hadar.optimizer.domain.numeric.NumpyNumericalValue*

Implementation with one scenario wiht shape (horizon,).

flatten() → numpy.ndarray
flat data into 1D matrix. :return: [v[0, 0], v[0, 1], v[0, 2], . . . , v[1, i], v[2, i], . . . , v[j, i])

static from_json(*dict*)

class hadar.optimizer.domain.numeric.**ScalarNumericalValue**(*value: T*, *horizon: int*, *nb_scn: int*)

Bases: *hadar.optimizer.domain.numeric.NumericalValue*

Implement one scalar numerical value i.e. float or int

flatten() → numpy.ndarray
flat data into 1D matrix. :return: [v[0, 0], v[0, 1], v[0, 2], . . . , v[1, i], v[2, i], . . . , v[j, i])

static from_json(*dict*)

hadar.optimizer.domain.output module

class hadar.optimizer.domain.output.**OutputProduction**(*quantity: Union[numpy.ndarray, list]*, *name: str = 'in'*)

Bases: *hadar.optimizer.utils.JSON*

Production element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.output.**OutputNode**(*consumptions: List[hadar.optimizer.domain.output.OutputConsumption]*, *productions: List[hadar.optimizer.domain.output.OutputProduction]*, *storages: List[hadar.optimizer.domain.output.OutputStorage]*, *links: List[hadar.optimizer.domain.output.OutputLink]*)

Bases: *hadar.optimizer.utils.JSON*

Node element

static build_like_input(*input: hadar.optimizer.domain.input.InputNode*, *fill: numpy.ndarray*)
Use an input node to create an output node. Keep list elements fill quantity by zeros.

Parameters

- **input** – InputNode to copy

- **fill** – array to use to fill data

Returns OutputNode like InputNode with all quantity at zero

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.output.**OutputStorage**(*name: str, capacity: Union[numpy.ndarray, list], flow_in: Union[numpy.ndarray, list], flow_out: Union[numpy.ndarray, list]*)

Bases: *hadar.optimizer.utils.JSON*

Storage element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.output.**OutputLink**(*dest: str, quantity: Union[numpy.ndarray, list]*)

Bases: *hadar.optimizer.utils.JSON*

Link element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.output.**OutputConsumption**(*quantity: Union[numpy.ndarray, list], name: str = ''*)

Bases: *hadar.optimizer.utils.JSON*

Consumption element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.output.**OutputNetwork**(*nodes: Dict[str, hadar.optimizer.domain.output.OutputNode]*)

Bases: *hadar.optimizer.utils.JSON*

Network element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.domain.output.**OutputConverter**(*name: str, flow_src: Dict[Tuple[str, str], Union[numpy.ndarray, List[T]]], flow_dest: Union[numpy.ndarray, List[T]]*)

Bases: *hadar.optimizer.utils.JSON*

Converter element

static from_json(*dict: dict*, *factory=None*)

to_json() → dict

class hadar.optimizer.domain.output.**Result**(*networks: Dict[str, hadar.optimizer.domain.output.OutputNetwork], converters: Dict[str, hadar.optimizer.domain.output.OutputConverter], benchmark: hadar.optimizer.domain.output.Benchmark = None*)

Bases: *hadar.optimizer.utils.JSON*

Result of study

static from_json(*dict*, *factory=None*)

Module contents

hadar.optimizer.lp package

Submodules

hadar.optimizer.lp.domain module

class hadar.optimizer.lp.domain.**JSONLP**
Bases: *hadar.optimizer.utils.JSON*, abc.ABC

static from_json(*dict*, *factory=None*)

to_json()

class hadar.optimizer.lp.domain.**LPConsumption**(*quantity: int, variable: Union[ortools.linear_solver.pywraplp.Variable, float], cost: float = 0, name: str = ''*)
Bases: *hadar.optimizer.lp.domain.JSONLP*

Consumption element for linear programming.

static from_json(*dict*, *factory=None*)

class hadar.optimizer.lp.domain.**LPConverter**(*name: str, src_ratios: Dict[Tuple[str, str], float], var_flow_src: Dict[Tuple[str, str], Union[ortools.linear_solver.pywraplp.Variable, float]], dest_network: str, dest_node: str, var_flow_dest: Union[ortools.linear_solver.pywraplp.Variable, float], cost: float, max: float*)
Bases: *hadar.optimizer.lp.domain.JSONLP*

Converter element for linear programming

static from_json(*dict*, *factory=None*)

class hadar.optimizer.lp.domain.**LPLink**(*src: str, dest: str, quantity: int, variable: Union[ortools.linear_solver.pywraplp.Variable, float], cost: float = 0*)
Bases: *hadar.optimizer.lp.domain.JSONLP*

Link element for linear programming

static from_json(*dict*, *factory=None*)

class hadar.optimizer.lp.domain.**LPNetwork**(*nodes: Dict[str, hadar.optimizer.lp.domain.LPNode] = None*)
Bases: *hadar.optimizer.utils.JSON*

Network element for linear programming

static from_json(*dict*, *factory=None*)

class hadar.optimizer.lp.domain.**LPNode**(*consumptions: List[hadar.optimizer.lp.domain.LPConsumption], productions: List[hadar.optimizer.lp.domain.LPProduction], storages: List[hadar.optimizer.lp.domain.LPStorage], links: List[hadar.optimizer.lp.domain.LPLink]*)
Bases: *hadar.optimizer.utils.JSON*

Node element for linear programming

static from_json(*dict*, *factory=None*)

class hadar.optimizer.lp.domain.**LPProduction**(*quantity: int, variable: Union[ortools.linear_solver.pywraplp.Variable, float], cost: float = 0, name: str = 'in'*)

Bases: *hadar.optimizer.lp.domain.JSONLP*

Production element for linear programming.

static from_json(*dict*, *factory=None*)

class hadar.optimizer.lp.domain.**LPStorage**(*name, capacity: int, var_capacity: Union[ortools.linear_solver.pywraplp.Variable, float], flow_in: float, var_flow_in: Union[ortools.linear_solver.pywraplp.Variable, float], flow_out: float, var_flow_out: Union[ortools.linear_solver.pywraplp.Variable, float], cost: float = 0, init_capacity: int = 0, eff: float = 0.99*)

Bases: *hadar.optimizer.lp.domain.JSONLP*

Storage element

static from_json(*dict*, *factory=None*)

class hadar.optimizer.lp.domain.**LPTimeStep**(*networks: Dict[str, hadar.optimizer.lp.domain.LPNetwork], converters: Dict[str, hadar.optimizer.lp.domain.LPConverter]*)

Bases: *hadar.optimizer.utils.JSON*

static create_like_study(*study: hadar.optimizer.domain.input.Study*)

static from_json(*dict*, *factory=None*)

hadar.optimizer.lp.mapper module

class hadar.optimizer.lp.mapper.**InputMapper**(*solver: ortools.linear_solver.pywraplp.Solver, study: hadar.optimizer.domain.input.Study*)

Bases: `object`

Input mapper from global domain to linear programming specific domain

get_conv_var(*name: str*, *t: int*, *scn: int*) → hadar.optimizer.lp.domain.LPConverter
Map Converter to LPConverter.

Parameters

- **name** – converter name
- **t** – time step
- **scn** – scenario index

Returns LPConverter

get_node_var(*network: str*, *node: str*, *t: int*, *scn: int*) → hadar.optimizer.lp.domain.LPNode
Map InputNode to LPNode.

Parameters

- **network** – network name

- **node** – node name
- **t** – time step
- **scn** – scenario index

Returns LPNode according to node name at t in study

class hadar.optimizer.lp.mapper.**OutputMapper**(*study: hadar.optimizer.domain.input.Study*)
Bases: object

Output mapper from specific linear programming domain to global domain.

get_result() → hadar.optimizer.domain.output.Result
Get result.

Returns final result after map all nodes

set_converter_var(*name: str*, *t: int*, *scn: int*, *vars: hadar.optimizer.lp.domain.LPConverter*)

set_node_var(*network: str*, *node: str*, *t: int*, *scn: int*, *vars: hadar.optimizer.lp.domain.LPNode*)
Map linear programming node to global node (set inside intern attribute).

Parameters

- **network** – network name
- **node** – node name
- **t** – timestamp index
- **scn** – scenario index
- **vars** – linear programming node with ortools variables inside

Returns None (use get_result)

hadar.optimizer.lp.optimizer module

class hadar.optimizer.lp.optimizer.**AdequacyBuilder**(*solver: ortools.linear_solver.pywraplp.Solver*)
Bases: object

Build adequacy flow constraint.

add_converter(*conv: hadar.optimizer.lp.domain.LPConverter*, *t: int*)
Add converter element in equation. Sources are like consumptions, destination like production

Parameters

- **conv** – converter element to add
- **t** – time index to use

Returns

add_node(*name_network: str*, *name_node: str*, *node: hadar.optimizer.lp.domain.LPNode*, *t: int*)
Add flow constraint for a specific node.

Parameters

- **name_network** – network name. Used to differentiate each equation
- **name_node** – node name. Used to differentiate each equation
- **node** – node to map constraint

Returns

build()
Call when all node are added. Apply all import flow for each node.

Returns

class hadar.optimizer.lp.optimizer.**ConverterMixBuilder**(*solver: ortools.linear_solver.pywraplp.Solver*)

Bases: object

Build equation to determine ratio mix between sources converter.

add_converter(*conv: hadar.optimizer.lp.domain.LPConverter*)

build()

class hadar.optimizer.lp.optimizer.**ObjectiveBuilder**(*solver: ortools.linear_solver.pywraplp.Solver*)

Bases: object

Build objective cost function.

add_converter(*conv: hadar.optimizer.lp.domain.LPConverter*)
Add converter. Apply cost on output of converter.

Parameters **conv** – converter to cost

Returns

add_node(*node: hadar.optimizer.lp.domain.LPNode*)
Add cost in objective for each node element.

Parameters **node** – node to add

Returns

build()

class hadar.optimizer.lp.optimizer.**StorageBuilder**(*solver: ortools.linear_solver.pywraplp.Solver*)

Bases: object

Build storage constraints

add_node(*name_network: str*, *name_node: str*, *node: hadar.optimizer.lp.domain.LPNode*, *t: int*) → ortools.linear_solver.pywraplp.Constraint

build()

hadar.optimizer.lp.optimizer.**solve_lp**(*study: hadar.optimizer.domain.input.Study*, *out_mapper=None*) → hadar.optimizer.domain.output.Result
Solve adequacy flow problem with a linear optimizer.

Parameters

- **study** – study to compute
- **out_mapper** – use only for test purpose to inject mock. Keep None as default.

Returns Result object with optimal solution

Module contents

hadar.optimizer.remote package

Submodules

hadar.optimizer.remote.optimizer module

exception hadar.optimizer.remote.optimizer.**ServerError**(*mes: str*)
Bases: Exception

hadar.optimizer.remote.optimizer.**check_code**(*code*)

hadar.optimizer.remote.optimizer.**solve_remote**(*study: hadar.optimizer.domain.input.Study*, *url: str*, *token: str = 'none'*) → hadar.optimizer.domain.output.Result

Send study to remote server.

Parameters

- **study** – study to resolve
- **url** – server url
- **token** – authorized token (default server config doesn't use token)

Returns result received from server

Module contents

Submodules

hadar.optimizer.optimizer module

class hadar.optimizer.optimizer.**LPOptimizer**
Bases: hadar.optimizer.optimizer.Optimizer

Basic Optimizer works with linear programming.

solve(*study: hadar.optimizer.domain.input.Study*) → hadar.optimizer.domain.output.Result
Solve adequacy study.

Parameters **study** – study to resolve

Returns study's result

class hadar.optimizer.optimizer.**RemoteOptimizer**(*url: str*, *token: str = ''*)
Bases: hadar.optimizer.optimizer.Optimizer

Use a remote optimizer to compute on cloud.

solve(*study: hadar.optimizer.domain.input.Study*) → hadar.optimizer.domain.output.Result
Solve adequacy study.

Parameters **study** – study to resolve

Returns study's result

hadar.optimizer.utils module

class hadar.optimizer.utils.**DTO**
Bases: object

Implement basic method for DTO objects

class hadar.optimizer.utils.**JSON**
Bases: *hadar.optimizer.utils.DTO*, abc.ABC

Object to be serializer by json

static convert(*value*)

static from_json(*dict*, *factory=None*)

to_json()

Module contents

hadar.viewer package

Submodules

hadar.viewer.abc module

class hadar.viewer.abc.**ABCElementPlotting**
Bases: abc.ABC

Abstract interface to implement to plot graphics

candles(*open: numpy.ndarray*, *close: numpy.ndarray*, *title: str*)
Plot candle stick with open close :param open: candle open data :param close: candle close data :param title: title to plot :return:

gaussian(*rac: numpy.ndarray*, *qt: numpy.ndarray*, *title: str*)
Plot gaussian.

Parameters

- **rac** – Remain Available Capacities matrix (to plot green or red point)
- **qt** – value vector
- **title** – title to plot

Returns

map_exchange(*nodes*, *lines*, *limit*, *title*, *zoom*)
Plot map with exchanges as arrow.

Parameters

- **nodes** – node to set on map
- **lines** – arrow to se on map
- **limit** – colorscale limit to use
- **title** – title to plot
- **zoom** – zoom to set on map

Returns

matrix(*data: numpy.ndarray*, *title*)
Plot matrix (heatmap)

Parameters

- **data** – 2D matrix to plot
- **title** – title to plot

Returns

monotone(*y: numpy.ndarray*, *title: str*)
Plot monotone.

Parameters

- **y** – value vector
- **title** – title to plot

Returns

stack(*areas: List[Tuple[str, numpy.ndarray]], lines: List[Tuple[str, numpy.ndarray]], title: str*)
Plot stack.

Parameters

- **areas** – list of timelines to stack with area
- **lines** – list of timelines to stack with line
- **title** – title to plot

Returns

timeline(*df: pandas.core.frame.DataFrame*, *title: str*)
Plot timeline with all scenarios.

Parameters

- **df** – dataframe with scenario on columns and time on index
- **title** – title to plot

Returns

class hadar.viewer.abc.**ABCPlotting**(*agg: hadar.analyzer.result.ResultAnalyzer*, *unit_symbol: str = ''*, *time_start=None*, *time_end=None*, *node_coord: Dict[str, List[float]] = None*)

Bases: abc.ABC

Abstract method to plot optimizer result.

network(*network: str = 'default'*)
Entry point to use fluent API.

Parameters **network** – select network to anlyze. Default is 'default'

Returns NetworkFluentAPISelector

class hadar.viewer.abc.**ConsumptionFluentAPISelector**(*plotting: hadar.viewer.abc.ABCElementPlotting*, *agg: hadar.analyzer.result.ResultAnalyzer*, *network: str*, *name: str*, *node: str*, *kind: str*)

Bases: *hadar.viewer.abc.FluentAPISelector*

Consumption level of fluent api.

gaussian(*t: int = None*, *scn: int = None*)
Plot gaussian graphics

Parameters

- **t** – focus on t index
- **scn** – focus on scn index if t not given

Returns

monotone(*t: int = None*, *scn: int = None*)
Plot monotone graphics.

Parameters

- **t** – focus on t index
- **scn** – focus on scn index if t not given

Returns

timeline()
Plot timeline graphics. :return:

class hadar.viewer.abc.**DestConverterFluentAPISelector**(*plotting: hadar.viewer.abc.ABCElementPlotting*, *agg: hadar.analyzer.result.ResultAnalyzer*, *network: str*, *node: str*, *name: str*)

Bases: *hadar.viewer.abc.FluentAPISelector*

Source converter level of fluent api

gaussian(*t: int = None*, *scn: int = None*)
Plot gaussian graphics

Parameters

- **t** – focus on t index
- **scn** – focus on scn index if t not given

Returns

monotone(*t: int = None*, *scn: int = None*)
Plot monotone graphics.

Parameters

- **t** – focus on t index
- **scn** – focus on scn index if t not given

Returns

timeline()
Plot timeline graphics. :return:

class hadar.viewer.abc.**FluentAPISelector**(*plotting: hadar.viewer.abc.ABCElementPlotting*, *agg: hadar.analyzer.result.ResultAnalyzer*)

Bases: abc.ABC

static not_both(*t: int*, *scn: int*)

class hadar.viewer.abc.**LinkFluentAPISelector**(*plotting: hadar.viewer.abc.ABCElementPlotting*, *agg: hadar.analyzer.result.ResultAnalyzer*, *network: str*, *src: str*, *dest: str*, *kind: str*)

Bases: *hadar.viewer.abc.FluentAPISelector*

Link level of fluent api

gaussian(*t: int = None*, *scn: int = None*)
Plot gaussian graphics

Parameters

- **t** – focus on t index
- **scn** – focus on scn index if t not given

Returns

monotone(*t: int = None*, *scn: int = None*)
Plot monotone graphics.

Parameters

- **t** – focus on t index
- **scn** – focus on scn index if t not given

Returns

timeline()
Plot timeline graphics. :return:

class hadar.viewer.abc.**NetworkFluentAPISelector**(*plotting: hadar.viewer.abc.ABCElementPlotting*, *agg: hadar.analyzer.result.ResultAnalyzer*, *network: str*)

Bases: *hadar.viewer.abc.FluentAPISelector*

Network level of fluent API

map(*t: int*, *zoom: int*, *scn: int = 0*, *limit: int = None*)
Plot map exchange graphics

Parameters

- **t** – t index to focus
- **zoom** – zoom to set
- **scn** – scn index to focus
- **limit** – color scale limite to use

Returns

node(*node: str*)
Go to node level fo fluent API :param node: node name :return: NodeFluentAPISelector

rac_matrix()
plot RAC matrix graphics

Returns

class hadar.viewer.abc.**NodeFluentAPISelector**(*plotting: hadar.viewer.abc.ABCElementPlotting*, *agg: hadar.analyzer.result.ResultAnalyzer*, *network: str*, *node: str*)

Bases: *hadar.viewer.abc.FluentAPISelector*

Node level of fluent api

consumption(*name: str*, *kind: str = 'given'*) → hadar.viewer.abc.ConsumptionFluentAPISelector
Go to consumption level of fluent API

Parameters

- **name** – select consumption name
- **kind** – kind of data 'asked' or 'given'

Returns

from_converter(*name: str*)
get a converter importation level fluent API :param name: :return:

link(*dest: str*, *kind: str = 'used'*)
got to link level of fluent API

Parameters

- **dest** – select destination node name
- **kind** – kind of data available ('avail') or 'used'

Returns

production(*name: str*, *kind: str = 'used'*) → hadar.viewer.abc.ProductionFluentAPISelector
Go to production level of fluent API

Parameters

- **name** – select production name
- **kind** – kind of data available ('avail') or 'used'

Returns

stack(*scn: int = 0*, *prod_kind: str = 'used'*, *cons_kind: str = 'asked'*)
Plot with production stacked with area and consumptions stacked by dashed lines.

Parameters

- **node** – select node to plot.
- **scn** – scenario index to plot.
- **prod_kind** – select which prod to stack : available ('avail') or 'used'
- **cons_kind** – select which cons to stack : 'asked' or 'given'

Returns plotly figure or jupyter widget to plot

storage(*name: str*) → hadar.viewer.abc.StorageFluentAPISelector
Got o storage level of fluent API

Parameters **name** – select storage name

Returns

to_converter(*name: str*)
get a converter exportation level fluent API :param name: :return:

class hadar.viewer.abc.**ProductionFluentAPISelector**(*plotting: hadar.viewer.abc.ABCElementPlotting*, *agg: hadar.analyzer.result.ResultAnalyzer*, *network: str*, *name: str*, *node: str*, *kind: str*)

Bases: *hadar.viewer.abc.FluentAPISelector*

Production level of fluent api

gaussian(*t: int = None*, *scn: int = None*)
Plot gaussian graphics

Parameters

- **t** – focus on t index
- **scn** – focus on scn index if t not given

Returns

monotone(*t: int = None*, *scn: int = None*)
Plot monotone graphics.

Parameters

- **t** – focus on t index
- **scn** – focus on scn index if t not given

Returns

timeline()
Plot timeline graphics. :return:

class hadar.viewer.abc.**SrcConverterFluentAPISelector**(*plotting: hadar.viewer.abc.ABCElementPlotting*, *agg: hadar.analyzer.result.ResultAnalyzer*, *network: str*, *node: str*, *name: str*)

Bases: *hadar.viewer.abc.FluentAPISelector*

Source converter level of fluent api

gaussian(*t: int = None*, *scn: int = None*)
Plot gaussian graphics

Parameters

- **t** – focus on t index
- **scn** – focus on scn index if t not given

Returns

monotone(*t: int = None*, *scn: int = None*)
Plot monotone graphics.

Parameters

- **t** – focus on t index

- **scn** – focus on scn index if t not given

Returns

timeline()
Plot timeline graphics. :return:

class hadar.viewer.abc.**StorageFluentAPISelector**(*plotting: hadar.viewer.abc.ABCElementPlotting*, *agg: hadar.analyzer.result.ResultAnalyzer*, *network: str*, *node: str*, *name: str*)

Bases: *hadar.viewer.abc.FluentAPISelector*

Storage level of fluent API

candles(*scn: int = 0*)

monotone(*t: int = None*, *scn: int = None*)
Plot monotone graphics.

Parameters

- **t** – focus on t index
- **scn** – focus on scn index if t not given

Returns

hadar.viewer.html module

class hadar.viewer.html.**HTMLPlotting**(*agg: hadar.analyzer.result.ResultAnalyzer*, *unit_symbol: str = ''*, *time_start=None*, *time_end=None*, *node_coord: Dict[str, List[float]] = None*)

Bases: *hadar.viewer.abc.ABCPlotting*

Plotting implementation interactive html graphics. (Use plotly)

Module contents

hadar.workflow package

Submodules

hadar.workflow.pipeline module

class hadar.workflow.pipeline.**RestrictedPlug**(*inputs: List[str] = None*, *outputs: List[str] = None*)

Bases: hadar.workflow.pipeline.Plug

Implementation where stage expect presence of precise columns.

linkable_to(*next*) → bool
Defined if next stage is linkable with current. In this implementation, plug is linkable only if input of next stage are present in output of current stage.

Parameters **next** – other stage to link

Returns True if current output contain mandatory columns for next input else False

class hadar.workflow.pipeline.**FreePlug**
Bases: hadar.workflow.pipeline.Plug

Plug implementation when stage can use any kind of DataFrame, whatever columns present inside.

linkable_to(*other: hadar.workflow.pipeline.Plug*) → bool
Defined if next stage is linkable with current. In this implementation, plug is always linkable

Parameters **other** – other stage to link

Returns True whatever

class hadar.workflow.pipeline.**Stage**(*plug: hadar.workflow.pipeline.Plug*)
Bases: abc.ABC

Abstract method which represent an unit of compute. It can be addition with other to create workflow pipeline.

static build_multi_index(*scenarios: Union[List[int], numpy.ndarray], names: List[str]*)
Create column multi index.

Parameters

- **scenarios** – list of scenarios serial
- **names** – names of data type preset inside each scenario

Returns multi-index like [(scn, type), . . .]

static get_names(*timeline: pandas.core.frame.DataFrame*) → List[str]

static get_scenarios(*timeline: pandas.core.frame.DataFrame*) → numpy.ndarray

static standardize_column(*timeline: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Timeline must have first column for scenario and second for data timeline. Add the Oth scenario index if not present.

Parameters **timeline** – timeline with or without scenario index

Returns timeline with scenario index

class hadar.workflow.pipeline.**FocusStage**(*plug*)
Bases: *hadar.workflow.pipeline.Stage*, abc.ABC

Stage focuses on same behaviour for any scenarios.

class hadar.workflow.pipeline.**Drop**(*names: Union[List[str], str]*)
Bases: *hadar.workflow.pipeline.Stage*

Drop columns by name.

class hadar.workflow.pipeline.**Rename**(***kwargs*)
Bases: *hadar.workflow.pipeline.Stage*

Rename column names.

class hadar.workflow.pipeline.**Fault**(*loss: float, occur_freq: float, downtime_min: int, downtime_max, seed: int = None*)
Bases: *hadar.workflow.pipeline.FocusStage*

Generate a random fault for each scenarios.

class hadar.workflow.pipeline.**RepeatScenario**(*n*)
Bases: *hadar.workflow.pipeline.Stage*

Repeat n-time current scenarios.

class hadar.workflow.pipeline.**ToShuffler**(*result_name: str*)
Bases: *hadar.workflow.pipeline.Rename*

To Connect pipeline to shuffler

class hadar.workflow.pipeline.**Pipeline**(*stages: List[T]*)
Bases: object

Compute many stages sequentially.

assert_computable(*timeline: pandas.core.frame.DataFrame*)
Verify timeline is computable by pipeline.

Parameters **timeline** – timeline to check

Returns True if computable False else

assert_to_shuffler()

class hadar.workflow.pipeline.**Clip**(*lower: float = None*, *upper: float = None*)
Bases: *hadar.workflow.pipeline.Stage*

Cut data according to upper and lower boundaries. Same as np.clip function.

hadar.workflow.shuffler module

class hadar.workflow.shuffler.**Shuffler**(*sampler=<built-in method randint of numpy.random.mtrand.RandomState object>*)
Bases: object

Receive all data sources like raw matrix or pipeline. Schedule pipeline generation and shuffle all timeline to create scenarios.

add_data(*name: str*, *data: numpy.ndarray*)
Add raw data by numpy array. If you generate data by pipeline use add_pipeline. It will parallelize computation and manage swap. :param name: timeline name :param data: numpy array with shape as (scenario, horizon) :return: self

add_pipeline(*name: str*, *data: pandas.core.frame.DataFrame*, *pipeline: hadar.workflow.pipeline.Pipeline*)
Add data by pipeline and input data for pipeline.

Parameters

- **name** – timeline name
- **data** – data to use as pipeline input
- **pipeline** – pipeline to generate data

Returns self

shuffle(*nb_scn*)
Start pipeline generation and shuffle result to create scenario sampling.

Parameters **nb_scn** – number of scenarios to sample

Returns

class hadar.workflow.shuffler.**Timeline**(*data: numpy.ndarray = None*, *sampler=<built-in method randint of numpy.random.mtrand.RandomState object>*)
Bases: object

Manage data used to generate timeline. Perform sampling too.

compute() → numpy.ndarray
Compute method called before sampling. For Timeline method just return data.

Returns return data given in constructor

sample(*nb*) → numpy.ndarray
Perform sampling. Compute data is needed before.

Parameters **nb** – number of sampling

Returns scenario matrix shape like (nb, horizon)

Module contents

5.1.2 Module contents

CHAPTER 6

Legal Terms

6.1 Libraries used

library	licences	copyright
python	PSF	Copyright (c) 2001-2020 Python Software Foundation.
pandas	BSD 3	Copyright (c) 2011-2020, Open source contributors.
numpy	BSD 3	Copyright (c) 2005-2020, NumPy Developers.
ortools	Apache 2	Copyright (c) 2010 Google LLC
plotly	MIT	Copyright (c) 2016-2018 Plotly, Inc
jupyter	BSD 3	Copyright (c) 2017, Project Jupyter Contributors
matplotlib	PSF	Copyright (c) 2002-2009 John D. Hunter
request	Apache 2	Copyright (c) 2019 Kenneth Reitz

Python Module Index

h

Index

D

F

G

H

I

J

L

M

N

O

P

R

S

T