Optibess algorithm

Release 0.1.0

Elmor Renewable Energies dev team

Mar 11, 2024

CONTENTS:

Optibess Algorithm is a python 3.10+ library for simulating and optimizing a photovoltaic system with power storage. It uses data from *pvgis* and algorithms from the *pvlib* and *Nevergrad* python libraries, and is the backend part of the *Optibess* site.

CHAPTER

ONE

QUICK START

Optibess Algorithm can be installed with:

```
pip install optibess_algorithm
```

You can run an optimization on an example system with:

```
import logging
import time
from optibess_algorithm.power_system_optimizer import NevergradOptimizer

# make info logging show
logging.getLogger().setLevel(logging.INFO)
# start optimization
start_time = time.time()
optimizer = NevergradOptimizer(budget=100)
opt_output, res = optimizer.run()
# print results
print(optimizer.get_candid(opt_output), res)
print(f"Optimization took {time.time() - start_time} seconds")
```

1.1 Optibess Algorithm - optimizing PV system combined with storage

Optibess Algorithm is a python 3.10+ library for simulating and optimizing a photovoltaic system with power storage. It uses data from *pvgis* and algorithms from the *pvlib* and *Nevergrad* python libraries, and is the backend part of the *Optibess* site.

1.1.1 Quick start

Optibess Algorithm can be installed with:

```
pip install optibess_algorithm
```

You can run an optimization on an example system with:

```
import logging
import time
```

(continues on next page)

(continued from previous page)

```
from optibess_algorithm.power_system_optimizer import NevergradOptimizer

# make info logging show
logging.getLogger().setLevel(logging.INFO)
# start optimization
start_time = time.time()
optimizer = NevergradOptimizer(budget=100)
opt_output, res = optimizer.run()
# print results
print(optimizer.get_candid(opt_output), res)
print(f"Optimization took {time.time() - start_time} seconds")
```

1.2 Installation

Optibess Algorithm is a Python 3.10+ library and can be installed with:

```
pip install Optibess_algorithm
```

You can also install the `main` branch instead of the latest release with:

```
pip install git+https://github.com/pvstorageoptimization/Optibess_algorithm@master
```

Alternatively, you can clone the repository and run `pip install -e .` from inside the repository folder.

1.3 Usage

1.3.1 Simulation

You can simulate the enrgy flow of a photovoltaic system with energy storage using an OutputCalculator object. You can recieve financial data and stats on a system after the simulation using a FinancialCalcualtor object.

Energy flow simulation

To simulate a photovoltaic system with energy storage you need to create 2 objects: Producer and PowerStorage

You can create a producer using 3 options:

- using a file with hourly energy output of a year (see :code:'test.csv' file for an example)
- using pvgis api to recieve hourly data of a year
- using pvlib to calculate the hourly energy output of a year

Create a producer using file with:

```
from optibess_algorithm.producers import PvProducer

prod = PvProducer(pv_output_file="test.csv", pv_peak_power=13000)
```

- `pv_output_file` is the name of the file
- `pv_peak_power` is the rated power of the producer.

you can also specify the timezone of the system with parameter `timezone` (defualt is Jerusalem timezone).

Create a producer using pvgis with:

```
from optibess_algorithm.producers import PvProducer, Tech

prod = PvProducer(latitude=30.02, longitude=34.84, tilt=16, azimuth=0, tech=Tech.TRACKER,
↪ pv_peak_power=10000, losses=9)
```

- `latitude` and `longitude` specify the location of the system
- `tilt` is the angle of the modules from the ground
- `azimuth` is the direction the module are facing (0 is south)
- `tech` is the type of modules used in the system (`FIXED`, `TRACKER`, or `EAST_WEST`)
- `pv_peak_power` is the rated power of the producer.
- `losses` is the overall PV system losses

Create a producer using pvlib with:

```
from optibess_algorithm.pv_output_calculator import MODULE_DEFAULT, INVERTER_DEFAULT
from optibess_algorithm.producers import PvProducer, Tech

prod = PvProducer(latitude=30.02, longitude=34.84, tilt=16, azimuth=0, tech=Tech.EAST_
↪WEST, modules_per_string=10,
                  strings_per_inverter=2, number_of_inverters=1000, module=MODULE_
↪DEFAULT, inverter=INVERTER_DEFAULT,
                  use_bifacial=True, albedo=0.2)
```

- `latitude` and `longitude` specify the location of the system
- `tilt` is the angle of the modules from the ground
- `azimuth` is the direction the module are facing (0 is south)
- `tech` is the type of modules used in the system (`FIXED`, `TRACKER`, or `EAST_WEST`)
- `modules_per_string` is the number of module in each electronic string
- `strings_per_inverter` is the number of string connected to each inverter
- `number_of_inverters` is the number of inverters in the PV system
- `module` is a pandas series with parameters for the module
- `inverter` is a pandas series with parameters for the inverter
- `use_bifacial` is a boolean idicating if the system uses bifacial calculation
- `albedo` is the fraction of sunlight diffusely reflected by the ground

Create a power storage with:

```
from optibess_algorithm.power_storage import LithiumPowerStorage

power_storage = LithiumPowerStorage(num_of_years=25, connection_size=5000, block_
↪size=500, battery_hours=2,
                                    use_default_aug=True)
```

- `num_of_year` is the number of years the storage system will be used

- `grid_size` is the size of the connection to the grid (in kW)
- `block_size` is the size of each block in the storage system
- `battery_hours` is the number of hours the storage system should supply each day (used to determine the size of the system)
- `use_default_aug` is a boolean idicating using a default augmentation configuration

The size of the system is determined by an augmentation table with the month each augmentation is installed and the number of blocks in each augmentation. When battery_hours is specified and use_default_aug is true we use 3 augmentations. The first augmentation is in the 0 month (system initial construction) with size that suffice for supplying battery_hours times grid_size (with extra for losses), and adding about 20% after 8 and 16 years. If use_default_aug is false only uses te first augmentation.

Additional parameters:

- `deg_table`, `dod_table` and `rte_table` are 3 listed of values between 0 and 1, specifying the degradation, depth of discharge and round trip efficiency for each year
- `pcs_loss`, `mvbat_loss` and `trans_loss` are different losses in the system
- `idle_self_consumption` and `active_self_consumption` are the percentage of the nominal storage system (nameplate) the battery consumes in each hour for its operation (active for hours wher charging/discharge and idle for the rest of the hours)
- `aug_table` is an option to specify the augmentation table directly

Using these objects, create and run an OutputCalculator with:

```
from optibess_algorithm.output_calculator import OutputCalculator
from optibess_algorithm.producers import PvProducer
from optibess_algorithm.power_storage import LithiumPowerStorage

import numpy as np

power_storage = LithiumPowerStorage(num_of_years=25, connection_size=5000, block_
↪size=500, battery_hours=2,
                                    use_default_aug=True)
prod = PvProducer("test.csv", pv_peak_power=13000)
output = OutputCalculator(num_of_years=25, grid_size=5000, producer=prod, power_
↪storage=power_storage,
                          producer_factor=1, save_all_results=True)
# run simulation
output.run()

# change print options to show full rows of the matrix
np.set_printoptions(linewidth=1000)
print(output.monthly_averages())
```

`monthly_averages` return a matrix with the average of the given stat (default total energy output) in each hour of the day for each month, in the given year range (default first year only). You can also use `plot_stat` function with similar parameters to plot a stat.

Parameters:

- `num_of_year` is the number of years the system will be used
- `grid_size` is the size of the connection to the grid (in kW)
- `producer` is a producer object

- `power_storage` is a power_storage object
- `coupling` is the type of coupling (AC/DC) used in the system (below are diagram of the 2 options)
- `mvpv_loss`, `trans_loss`, `mvbat_loss`, `pcs_loss` and `dc_dc_loss` are the different system losses
- `bess_discharge_hour` is the hour the system start to discharge the storage
- `fill_battery_from_grid` is a boolean idicating if the battery is filled from the grid when the producer power is not sufficient to fill the battery
- `save_all_results` is a boolean idicating saving all the data for all the simulation years (and also save additional stats needed for some of the financial calculations)
- `producer_factor` is a number between 0 and 1 which the producer output is multiplied by

Financial calculations

After creating an output calculator you can pass it to a financial calculator that has methods for calculating several financial stats and financial data:

```
from optibess_algorithm.output_calculator import OutputCalculator
from optibess_algorithm.constants import *
from optibess_algorithm.producers import PvProducer
from optibess_algorithm.power_storage import LithiumPowerStorage
from optibess_algorithm.financial_calculator import FinancialCalculator

import time

storage = LithiumPowerStorage(25, 5000, aug_table=((0, 83), (96, 16), (192, 16)))
producer = PvProducer("test.csv", pv_peak_power=15000)

output = OutputCalculator(25, 5000, producer, storage, save_all_results=True, fill_
↪battery_from_grid=False,
                          bess_discharge_start_hour=17, producer_factor=1)
fc = FinancialCalculator(output_calculator=output, land_size=100, capex_per_land_
↪unit=215000, capex_per_kwp=370,
                         opex_per_kwp=5, battery_capex_per_kwh=170, battery_opex_per_
↪kwh=5,
                         battery_connection_capex_per_kw=50, battery_connection_opex_per_
↪kw=0.5, fixed_capex=150000,
                         fixed_opex=10000, interest_rate=0.04, cpi=0.02, battery_cost_
↪deg=0.07, base_tariff=0.14,
                         winter_low_factor=1.1, winter_high_factor=4, transition_low_
↪factor=1,
                         transition_high_factor=1.2, summer_low_factor=1.2, summer_high_
↪factor=6,
                         buy_from_grid_factor=1)
start_time = time.time()
output.run()
print("irr: ", fc.get_irr())
print("npv: ", fc.get_npv(5))
print("lcoe: ", fc.get_lcoe())
print("lcos: ", fc.get_lcos())
print("lcoe no grid power:", fc.get_lcoe_no_power_costs())
print(f"calculation took: {(time.time() - start_time)} seconds")
```

The financial stats caluclated:

- `irr` is the internal rate of return of the system
- `npv` is the net present value of the system
- `lcoe` is the levelized cost of energy of the system (energy produced and purchased)
- `lcos` is the levelized cost of storage of the system
- `lcoe no grid power` is lcoe of the power from PV only

parameters:

- `output_calcualtor` is an OutputCalculator object
- `land_size` is the size of land used for the system
- `capex/opex` are the cost of the system separated into 5 categories: by land size, by PV size, by battery size, by the size of the connection to the battery and misc.
- `usd_to_ils` is a convertion rate from us dollars to israeli new shekel
- `interest_rate` is the market interest rate
- `cpi` is the consumer price index
- `battery_deg_cost` is the annual reduction of battery cost (in percentage)
- `base_tariff` is the base tariff used to construct the tariff table
- `low/high_winter/transition/summer_factor` are factors by which the the base tariff is multiplied to create the tariff table
- `buy_from_grid_factor` is a factor by which to multiply a tariff to get the prices of buy power
- `tariff_table` is an option to specify the tariff table directly

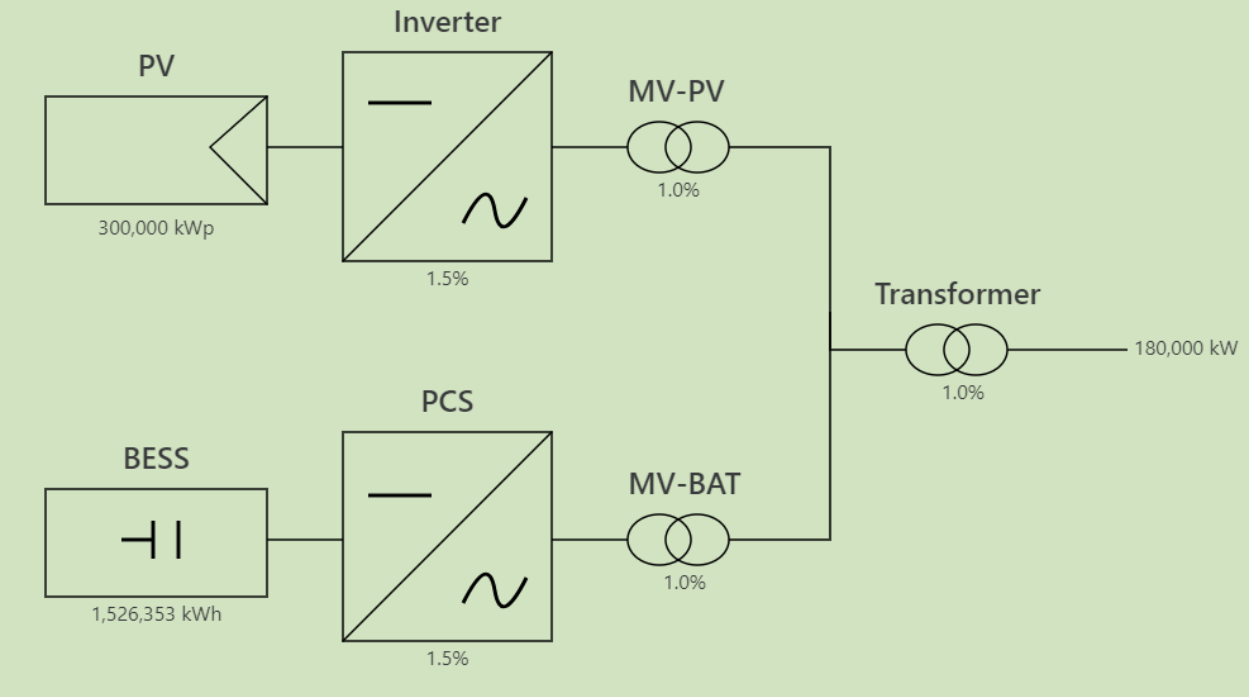
The tariff table is constructed according to the following table:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Jan	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	1	1
Feb	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	1	1
Mar	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	3	3
Apr	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	3	3
May	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	3	3
Jun	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	5
Jul	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	5
Aug	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	5
Sep	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	5
Oct	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	3	3
Nov	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	3	3
Dec	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	1	1

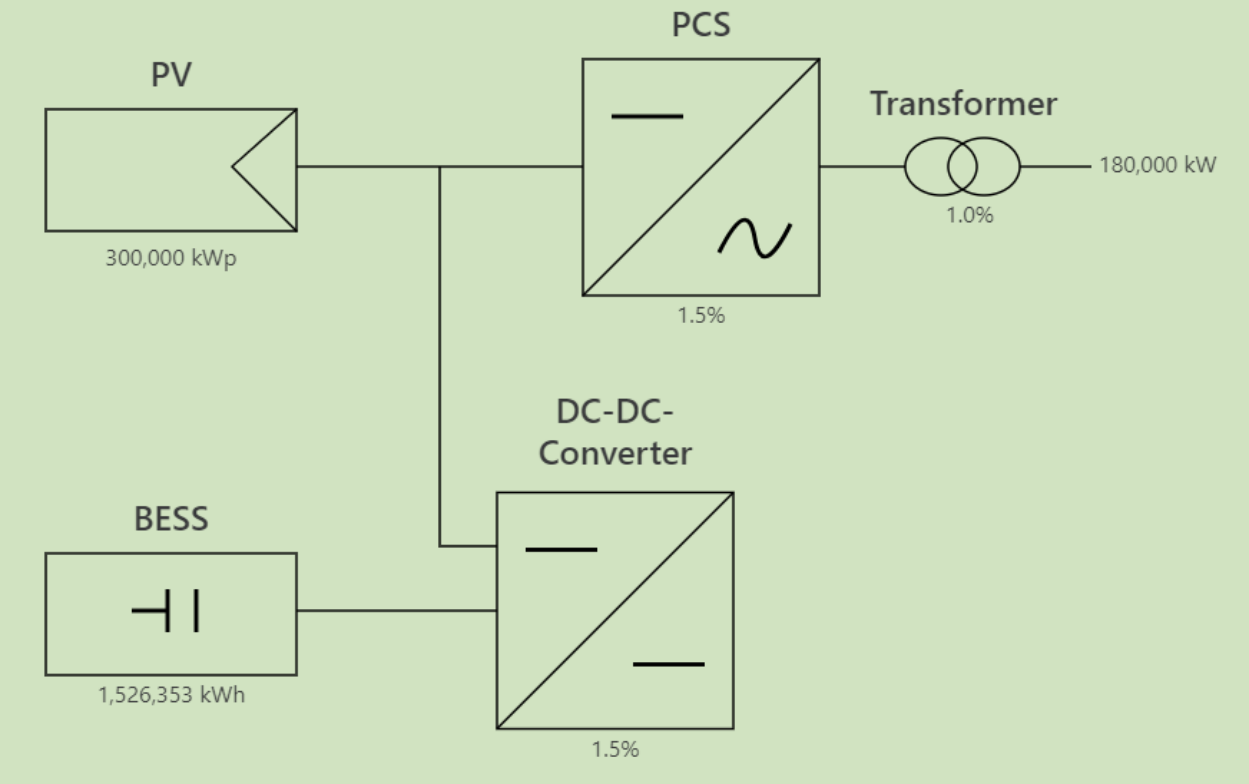
Note: The current version is only suited for working with tariffs with similar structure to the table above

Diagrams of the system

AC coupling:



DC coupling:



1.3.2 Optimization

You can run an optimization on an example system with:

```
import logging
import time
from optibess_algorithm.power_system_optimizer import NevergradOptimizer

# make info logging show
logging.getLogger().setLevel(logging.INFO)
# start optimization
start_time = time.time()
optimizer = NevergradOptimizer(budget=100)
opt_output, res = optimizer.run()
# print results
print(optimizer.get_candid(opt_output), res)
print(f"Optimization took {time.time() - start_time} seconds")
```

The outputs of the optimizer run method are the parameters of the optimal result (augmentation table and PV size factor) and the optimal result (the irr of the system with these parameters).

Run an optimization on a custom system with:

```
from optibess_algorithm.constants import MAX_BATTERY_HOURS
from optibess_algorithm.financial_calculator import FinancialCalculator
from optibess_algorithm.output_calculator import OutputCalculator
from optibess_algorithm.power_storage import LithiumPowerStorage
from optibess_algorithm.producers import PvProducer
from optibess_algorithm.power_system_optimizer import NevergradOptimizer

import logging
import time
# make info logging show
logging.getLogger().setLevel(logging.INFO)
# setup power system
storage = LithiumPowerStorage(25, 5000, use_default_aug=True)
producer = PvProducer("test.csv", pv_peak_power=15000)
output = OutputCalculator(25, 5000, producer, storage, save_all_results=False)
finance = FinancialCalculator(output, 100)

# start optimization
start_time = time.time()
optimizer = NevergradOptimizer(financial_calculator=finance, use_memory=True, max_aug_
↪num=6, initial_aug_num=3,
                               budget=2000)
opt_output, res = optimizer.run()
# print results
print(optimizer.get_candid(opt_output), res)
print(f"Optimization took {time.time() - start_time} seconds")
```

- `financial_calculator` is a FinancialCalcualtor object
- `use_memory` is a boolean idicating if the optimizer will use a memory dict to get result of repeating queries quickly
- `max_aug_num` is the maximum number of augmentations the optimizer will try in a solution

- `initial_aug_num` is the number of augmentation in the initial guess
- `budget` is the number of simulation to use for optimization

Additional parameters for Nevergrad optimizer:

- `max_no_change_steps` is the maximum number of optimization step with no change before stopping (if none, does not use early stopping)
- `min_change_size` is the minimum change between steps to consider as a change for early stopping
- `verbosity` is print information from the optimization algorithm (0: None, 1: fitness values, 2: fitness values and recommendation)

1.4 Optibess_algorithm

1.4.1 optibess_algorithm package

optibess_algorithm.constants module

optibess_algorithm.financial_calculator module

optibess_algorithm.output_calculator module

optibess_algorithm.power_storage module

optibess_algorithm.power_system_optimizer module

optibess_algorithm.producers module

optibess_algorithm.pv_output_calculator module

optibess_algorithm.utils module

Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search