SoftwareX 32 (2025) 102324

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

ACDC-OpFlow: A unified, cross-language framework for AC/DC optimal power flow solutions

Haixiao Li [a,*], Azadeh Kermansaravi [a], Robert Dimitrovski [a,b], Aleksandra Lekić [a]

[a] Faculty of Electrical Engineering, Mathematics, and Computer Science, TU Delft, Mekelweg 4, 2618 CD, Delft, Netherlands
[b] TenneT TSO GmbH, Bernecker Str. 70, 95448 Bayreuth, Germany

ARTICLE INFO

Keywords:
Cross-language framework
AC/DC optimal power flow
Multi-terminal DC

ABSTRACT

Hybrid AC/voltage source converter-based multi-terminal DC (VSC-MTDC) power grids play a crucial role in enabling long-distance power transmission and flexible interconnection between AC grids. To fully leverage the functional advantages of such systems, it is essential that they operate in or close to optimal power flow (OPF) conditions. To address this, *ACDC-OpFlow* is developed as an open-source and cross-language framework for solving AC/DC OPF problems. Its core innovation lies in a unified modeling structure that supports MATLAB, Python, Julia, and C++, with Gurobi used as a consistent solver backend. This framework is beginner-friendly and allows users to work in their preferred programming languages. Both text-based and graph-topology results are provided to help users understand the system-wide power flow distribution and operational status. This work presents the design concept of *ACDC-OpFlow*, showcases representative example results, and discusses the performance differences observed in multiple programming language implementations.

Code metadata

Current code version	v0.1.2
Permanent link to code/repository	https://github.com/ElsevierSoftwareX/SOFTX-D-25-00355
Code Ocean compute capsule	N/A
Legal code license	MIT
Code versioning system used	Git
Software languages, tools, and services	MATLAB, Python, Julia, C++
Compilation requirements, operating environments & dependencies	MATLAB requires: YALMIP toolbox. Python requires: numpy, pandas, scipy, pyomo, networkx, matplotlib, gurobipy. Julia requires: CSV, DataFrames, JuMP, Gurobi, Graphs, GraphPlot, GLMakie, ColorSchems, Colors. C++ requires: Eigen, Matplot++, Gurobi C++ API.
Link to developer documentation/manual	https://github.com/CRESYM/ACDC_OPF/blob/main/Manual_v0.1.2.pdf
Support email for questions	haixiaoli.ee@gmail.com

1. Motivation and significance

AC/voltage source converter-based multi-terminal DC (VSC-MTDC) grids play a crucial role in modern power systems by enhancing long-distance power transmission, renewable energy sources (RESs) accommodation, and interconnection of regional power grids. VSC-MTDC allows independent active and reactive power control, enabling seamless connection of AC grids in different regions. The scheduling of a power system consisting of multiple interconnected AC grids and a VSC-MTDC grid is significant [1,2]. As depicted in Fig. 1, we need a technique to help enhance the rationality and optimality of scheduling, which involves solving the problems known as the AC/DC optimal power flow (OPF).

OPF solution can be broadly classified into data-driven and model-based methods. Data-driven OPF techniques have become a research hot spot in recent times [3–5]. They typically use large and diverse datasets to train neural networks that generate optimal decisions. In contrast, model-based OPF is a more conventional solution path, where OPF decisions are obtained by solving mathematical programming

* Corresponding author.
E-mail address: haixiaoli.ee@gmail.com (H. Li).

https://doi.org/10.1016/j.softx.2025.102324
Received 28 May 2025; Received in revised form 3 August 2025; Accepted 18 August 2025
Available online 15 September 2025

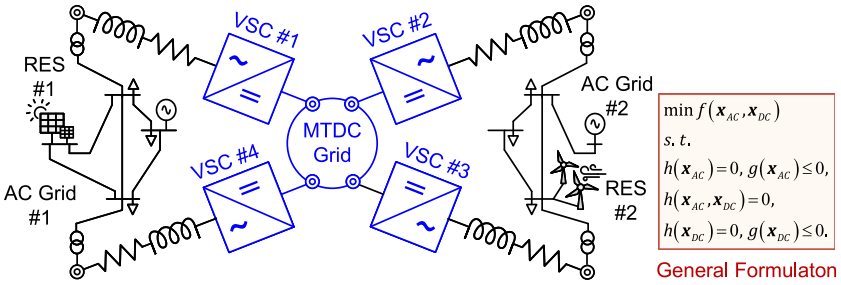

Fig. 1. Overview of the AC/DC OPF problem used to schedule AC/VSC-MTDC grids.

models that explicitly describe the power system operation. Undoubtedly, data-driven OPF methods hold great promise for future applications. However, this does not mean that model-based OPF approaches are obsolete. They offer distinct advantages that data-driven methods often lack. For example, model-based OPF provides strong theoretical guarantees and strict adherence to operational constraints. More importantly, the power flow data required to train data-driven OPF models fundamentally relies on model-based OPF solutions.

As such, model-based OPF research remains of significant interest. Currently, several open-source tools have been developed to support model-based AC/DC OPF analysis. *PowerModelsACDC.jl* [6] developed on top of *PowerModels.jl* [7] is one of the most widely used. It provides flexible options for AC/DC OPF formulation. Subsequently, several extensions based on *PowerModelsACDC.jl*, such as *PowerModelsMCDC.jl* [8] and *PowerModelsACDCsecurityconstrained.jl* [9], have been developed to address specific AC/DC OPF challenges, including unbalanced operation, $N-1$ security constraints, and more. In addition to Julia-based tools, a Python-based tool named *pyflow_acdc.py* [10] has recently been released. It features constructing the enhanced AC/DC OPF formulation that supports the heterogeneous MTDC grid.

Despite the success of the aforementioned tools, our work does not aim to develop a more comprehensive or feature-rich toolkit. Instead, we target a practical yet often overlooked issue: researchers and educators, particularly those new to AC/DC OPF, often have different preferred programming languages. To this end, we design *ACDC-OpFlow*, a beginner-friendly unified framework that supports MATLAB, C++, Julia, and Python, which are four of the most commonly used programming languages in the research and engineering practice of power system analysis. All implementations share the same core coding logic: data preparation, parameter processing, OPF model construction, and OPF result display. For each programming language, the AC/DC OPF model is formulated as a convex second-order cone programming (SOCP) problem by appropriate relaxation of non-convex AC and DC power flow equations [11,12]. Compared to traditional nonlinear programming (NLP) OPF models, SOCP-based OPF models are more scalable and extensible, especially when integrating additional operational constraints such as topology reconfiguration [13,14] or unit commitment decisions [15]. These extensions can often be represented as mixed-integer SOCP (MISOCP), which can still be solved efficiently using off-the-shell mathematical programming solvers. In contrast, adding integer variables to an already NLP model results in mixed-integer NLP (MINLP), which is computationally intractable. To ensure consistent solutions in all programming languages, *ACDC-OpFlow* employs Gurobi,[1] a powerful solver, as the unified solver back-end in all language implementations. To respect the conventions of each supported language, language-specific dependencies are carefully selected and used for OPF model construction and result display. This structure-unified and dependency-aware design enables users to start with the programming language they are most comfortable with, and later transition smoothly to other versions as needed. *ACDC-OpFlow* makes it easier to analyze differences in modeling syntax, solver behavior, and computational performance. *ACDC-OpFlow* is now available on GitHub as an open-source repository [16].

To introduce *ACDC-OpFlow*, this paper is organized as follows: In Section 2, a description of *ACDC-OpFlow* is detailed. The aspects, such as the architecture and functionalities of this framework, as well as the defining approaches of parameters and variables in the AC/DC OPF problem are presented. Later, in Section 3, the results of solving the AC/DC OPF problems are presented. It includes both printed textual outputs and visualized plotting results and efficiency of each programming language. The impact of *ACDC-OpFlow* is discussed in 4. Finally, the conclusions and future work are drawn in Section 5.

2. Software description

2.1. Software architecture

As presented in Fig. 2, under the framework of *ACDC-OpFlow*, every programming language follows a unified workflow: Firstly, we prepare input data related to AC and MTDC grids, which are all saved in structured comma-separated value (CSV) files. Secondly, extracting key parameter information from CSV data files and handling it to form the specific parameter vectors (matrices). Thirdly, based on predefined parameter vectors (matrices), the AC/DC OPF model with SOCP formulation is constructed and subsequently solved via the Gurobi solver, which supports the solution of SOCP problems by using its built-in interior-point method. Lastly, the OPF results are organized and output in text format, accompanied by a visual presentation to facilitate observation of the results. For each programming language, the core functions involved are shown in Fig. 3. In the following, a detailed description of each step in the workflow is provided.

2.1.1. Data preparation

The data required for AC/DC OPF are listed and shown in Fig. 4, and all data are stored in the CSV file due to its compatibility with various programming language environments.

The fundamental data related to the MTDC grid are: Base power of the MTDC grid saved in `baseMW_dc.csv`; Polarity of the MTDC grid saved in `pol_dc.csv`; Bus orders and allowable nodal voltage bounds saved in `branch_dc.csv`; From/To nodes and resistances of DC branches saved in `branch_dc.csv`; Bus orders of the connected AC and DC terminals of converters, equivalent AC-side impedances, converter control modes, saved in `conv_dc.csv`.

The fundamental data related to the interconnected AC grid are: Unified base power of interconnected AC grids saved in `baseMVA_ac.csv`; Bus orders, nodal shunt admittances, nodal voltage bounds, saved in `bus_ac.csv`; From/To nodes and impedances of AC branches saved in `branch_ac.csv`; Bus order of generators connected to and generator power output bounds saved in `gen_ac.csv`; Coefficients of generation costs saved in `gencost_ac.csv`; Bus order of RESs connected to, power capacity limits, and coefficients of generation costs saved in `res_ac.csv`.

[1] https://www.gurobi.com/

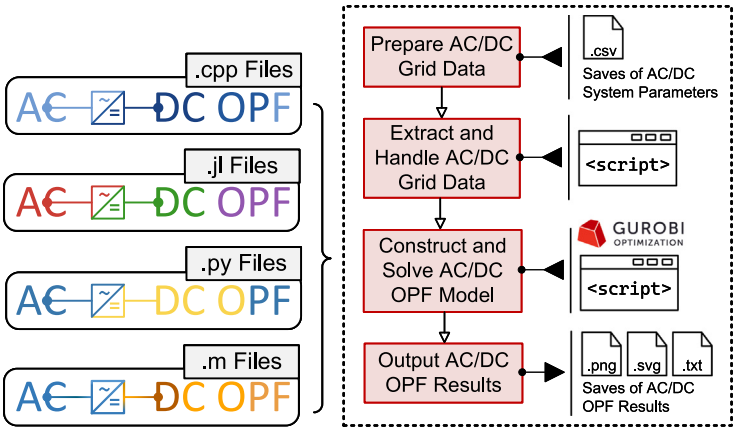

Fig. 2. Unified work flow of *ACDC-OpFlow* across programming languages.

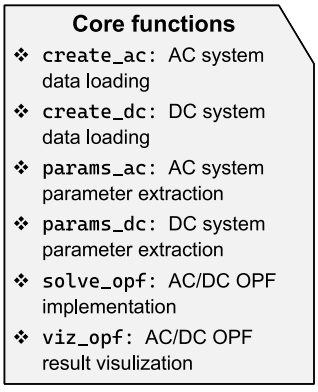

Fig. 3. Core functions across programming languages.

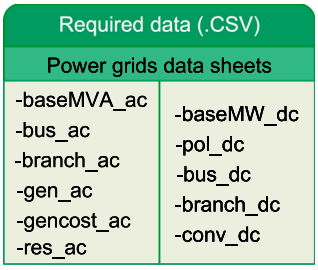

Fig. 4. Data sheets of AC and MTDC grids.

Users can manually fill in custom values in the CSV files mentioned above.[2] In general, entering parameter values related to the MTDC grid is a moderately simple task, as the number of MTDC terminals is quite limited, typically ranging from three to five [18]. In the case of a five-terminal MTDC network, the number of associated DC buses is 5, with the maximum number of DC branches being $\binom{5}{2} = 10$. In addition, the file associated with RES is also very easy to create, as the number of grid-connected renewable energy sources is typically limited in practice, the required parameters are few, and the format is well-suited for manual entry. However, when dealing with interconnected AC grids, manual entry of parameter values can be considerably labour-intensive. Consequently, we suggest that users make use of the existing single-area AC test cases in MATPOWER[3] to build interconnected AC grids in multiple areas. In *ACDC-OpFlow*, mpc_merged.m and save_csv.m can be utilized to facilitate this process, and an example step is shown below:

```
merged_ac = mpc_merged('case9', 'case14'); % mpc_merged('case_a', 'case_b', ...)
save_csv(merged_ac); % If no path is provided, saved in the current path.
```

A message will be displayed in the terminal window once the code has been successfully executed:

```
>> Reordered bus IDs for case9
>> Reordered bus IDs for case14
>> Enter a prefix for the CSV filenames: ac9ac14
>> Merged AC grid data has already been saved to:
>>  D:\acdcopf\Tests\ac9ac14_baseMVA_ac.csv
>>  D:\acdcopf\Tests\ac9ac14_bus_ac.csv
>>  D:\acdcopf\Tests\ac9ac14_gen_ac.csv
>>  D:\acdcopf\Tests\ac9ac14_branch_ac.csv
>>  D:\acdcopf\Tests\ac9ac14_gencost_ac.csv
```

2.1.2. Parameter processing

Language-specific functions `create_ac` and `create_dc` read data from the CSV file associated with the interconnected AC grids and the MTDC grid. Depending on the programming environment, different handling approaches are used to read and organize these CSV files, as listed in Table 1.

Then, language-specific functions `params_ac` and `params_dc` are further utilized to extract the essential parameters required to build the AC/DC OPF model. During this process, different language-specific conventions are adopted to define one-dimensional, two-dimensional, and three-dimensional parameters, which are listed in Table 2.

Having passed through this phase, all necessary parameters for the interconnected AC grids and MTDC grid have been fully defined, forming sufficient support for the subsequent AC/DC OPF computations.

2.1.3. Core OPF solving

The core OPF solver can be found in `solve_opf`, where the AC/DC OPF model is formulated and solved. The power flow equations for both AC and DC grids are inherently strong non-linear and non-convex, whereas NLP often suffers from convergence to local optima, numerical instability, and high computational complexity. From this

[2] In CSV files, each column corresponds to a parameter item. The specific rules can be found in [17].

[3] https://matpower.org/

Table 1
CSV file data handling approaches across programming languages.

Language	Read .csv files	Data structure for storage
MATLAB	`readmatrix` (built-in)	Uses a `struct` where each named field maps to a numeric matrix or a scalar.
Python	`pandas.read_csv`	Uses a `dict` where each string key maps to a `NumPy` array or a scalar.
Julia	`CSV.File` + `DataFrame`	Uses a `Dict{String, Any}` where each string key maps to a numeric array or a scalar.
C++	`ifstream` + `csv_reader` (custom)	Uses an `unordered_map<std::string, Eigen::MatrixXd>` where each string key maps to an `Eigen` matrix.

Table 2
Parameter definition approaches across programming languages.

Language	1D Parameter (`fbus_dc`, `nbuses_ac`, etc.)	2D Parameter (`bus_dc`, `pd_ac`, etc.)	3D Parameter (`bus_entire_ac`, etc.)
MATLAB	`vector/x(i)`, `cell/x{n}`	`matrix/x(i, j)`, `cell/x{n}(i)`	`cell/x{n}(i, j)`
Python	`np.ndarray/x[i]`, `list/x[n]`	`np.ndarray/x[i, j]`, `list/x[n][i]`	`list/x[n][i, j]`
Julia	`Vector/x[i]`, `Vector/x[n]`	`Matrix/x[i, j]`, `Vector/x[n][i]`	`Vector/x[n][i, j]`
C++	`Eigen::Vector/x(i)`, `std::vector/x[i]`	`Eigen::Matrix/x(i, j)`, `std::vector/x[n](i)`	`std::vector/x[n](i, j)`

✧ The mark means `type/syntax`. Here, `n` refers to the interconnected AC area index, `i` and `j` refers to the system bus index.

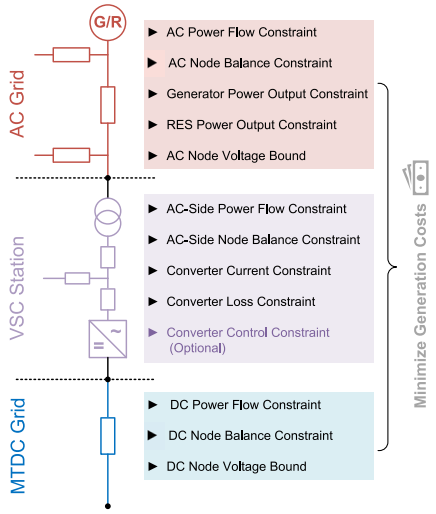

Fig. 5. Operational constraints considered in the constructed AC/DC OPF model.

perspective, *ACDC-OpFlow* emphasizes the importance of convex relaxation and reformulates the AC/DC OPF model as a SOCP model, which allows off-the-shelf convex solver to achieve globally optimal solutions. The detailed SCOP formulation, on which our AC/DC OPF model is based, can be found in [17].

In implementations across MATLAB, Python, Julia, and C++, *ACDC-OpFlow* utilizes Gurobi that supports SOCP solving and offers academic licenses free of charge for research and educational use,[4] as the unified solver backend. To facilitate AC/DC OPF modeling and solver invocation, *ACDC-OpFlow* integrates each language with a widely adopted modeling interface and the corresponding solver bridge. Table 3 summarizes the dependencies used for optimization in each programming environment.

[4] https://www.gurobi.com/academia/academic-program-and-licenses/

To further support optimization modeling, each implementation explicitly defines optimization decision variables and adds optimization constraints, where variables and constraints are constructed in compatible formats with each modeling framework, as shown in Table 4.

Similarly to parameter definition, optimization variable definition also conforms to language-specific conventions, and the summary of representative expressions for defining one-dimensional, two-dimensional, and three-dimensional variables in across programming languages can be found in Appendix.

The operational constraints considered in the constructed AC/DC OPF model are presented in Fig. 5. Among these constraints, the converter control constraint is optional. In `solve_opf`, users can set the attribute `vscControl` as `true` to enable this constraint. It means the pre-defined VSC control settings in `conv_dc.csv` are included in the formed AC/DC OPF model. Conversely, setting `vscControl` as `false` allows the optimization to treat the control settings as decision variables, and the converter control constraint is excluded accordingly.

Upon completion of this phase, the optimal values of key electrical variables can be obtained and ready to be displayed.

2.1.4. Result display

ACDC-OpFlow designs two approaches to display results. The first approach directly prints the optimized results in the terminal window, following a style close to that of MATPOWER. The printed outputs include optimized node and branch information for both interconnected AC grids and the MTDC grid. In addition, users can set the attribute `writeTxt` as `true` in `solve_opf` to write the printed results into a TXT file. The second approach provides a graphical visualization of power flow results. Such visualization function relies on `viz_opf`, which is integrated and called within `solve_opf`. This feature can be enabled by setting the attribute `plotResult` as `true` in `solve_opf`. Required plotting tools for programming languages are summarized in Table 5.

2.2. Software functionalities

ACDC-OpFlow is a unified framework developed for conducting AC/DC OPF analysis in hybrid AC/VSC-MTDC power systems. The framework integrates the entire OPF workflow, from data preparation and parameter extraction to model formulation and result presentation,

Table 3
Optimization modeling dependencies across programming languages.

Language	Dependencies (functionalities)
MATLAB	YALMIP[a] (modeling interface + solver bridge)
Python	Pyomo[b] (modeling interface) + gurobipy (solver bridge)
Julia	JuMP[c] (modeling interface) + Gurobi.jl (solver bridge)
C++	Gurobi C++ API[d] (modeling interface + solver bridge)

[a] https://yalmip.github.io/.
[b] https://www.pyomo.org/.
[c] https://jump.dev/.
[d] https://docs.gurobi.com/projects/optimizer/en/current/reference/cpp.html.

Table 4
Optimization variable and constraint definition across programming languages.

Language	Define variables	Add constraints
MATLAB	`sdpvar(...)`	`Cons = [Cons; expr]`
Python	`Var(..., domain = Reals)`	`model.addconstraints = ConstraintList()` `model.addconstraints.add(expr)`
Julia	`@variable(model, ...)`	`@constraint(model, expr)`
C++	`model.addVar(..., GRB_CONTINUOUS)`	`model.addConstr(expr)` (for linear constraints), `model.addQConstr(expr)` (for quadratic constraints)

Table 5
Plotting tools across programming languages.

Language	Visualization tools
MATLAB	Graph, plot, scatter, etc. (built-in plotting + network layout tools)
Python	matplotlib (plotting tool) + networkx (network layout tool)
Julia	GLMakie, ColorScehmes.jl, Colors.jl (plotting tool) + Graphs.jl, GraphPlot.jl (network layout tool)
C++	Matplot++ (plotting + network layout tools)

supporting implementations in MATLAB, Python, Julia, and C++. A key feature of *ACDC-OpFlow* is its consistent use of Gurobi as a unified AC/DC OPF solver across all environments. By aligning with language-specific conventions for data structures and mathematical modeling, the framework ensures high familiarity for researchers and developers. This design philosophy make the framework extensible in both research and education related to AC/DC OPF studies.

3. Illustrative examples

3.1. Running AC/dc OPF

MATLAB, Python, Julia, and C++ follows a unified running procedure. As mentioned earlier, before running the AC/DC OPF, we need to prepare the required CSV files to save the parameter data related to interconnected AC grids and the MTDC grid. A custom set of CSV files related to the MTDC grid has been created, uniformly named under the prefix `mtdc3slack_a`. Also, a set of interconnected AC grid data has been created, uniformly named under the prefix `ac9ac14`. In each programming language, users can execute AC/DC OPF through the code snippet as below:

```
solve_opf('mtdc3slack_a', 'ac9ac14',
      'vscControl', true,
      'writeTxt', false,
      'plotResult', true)
```

Then, users will see the optimized AC/DC OPF results printed on the terminal window such that (see top of next page):

Subsequently, the results of the AC/DC OPF will be visualized, as shown in Fig. 6. The graphical representation allows users to intuitively learn the entire system-wide power flow distribution. Furthermore, for each programming language, the generated plots support interactive zooming, allowing users to check specific regions of interest in detail.

3.2. Comparing computation performances

To evaluate the performance of different programming languages in solving AC/DC OPF problems, we developed equivalent benchmark scripts in MATLAB, Python, Julia, and C++, each implementation was tested under the identical hardware condition.[5] We provide four test cases, ranging in scale from relative small to relative large. Each case involves two main parts: the MTDC grid and the interconnected AC grids. In the test cases, the MTDC grid is shared and is named `mtdc3slack_a` which represents a three-terminal MTDC grid in which one VSC station maintains a constant DC voltage as the slack node. The detailed parameter settings is originally from "case5_stagg_MTDCslack" in MATACDC.[6] The interconnected AC grids in the test cases are `ac9ac14`, `ac14ac57`, `ac57ac118`, and `ac118ac300`, originally from the dataset in MATPOWER. For example, `ac9ac14` indicates an interconnected AC network composed of a 9-bus AC system and a 14-bus AC system, whose detailed parameter setting are identical to "case9" and "case14" of MATPOWER. Each script was executed five times per case following three warmup runs to mitigate just-in-time (JIT) compilation and caching effects. Execution time and memory usage were recorded using language-specific timing and memory monitoring approaches, which are summarized in Table 6.

Post-processing recorded the median runtime, runtime variation (min–max), peak memory usage, and relative memory use normalized against the smallest test case as the baseline for each language. The benchmarking results are summarized in Table 7. For each case, we report the median execution time, run-time range, peak memory usage, and relative memory usage normalized to the smallest test case within each language. The presented results reveal clear trade-offs among the four programming languages in terms of the implementation of AC/DC OPF:

[5] Benchmark scripts was executed on a laptop equipped with a 12th Gen Intel(R) Core(TM) i9 Processor, 32 GB of RAM, Windows environment.
[6] https://www.esat.kuleuven.be/electa/teaching/matacdc

```
================================================================================================
|      AC Grid Bus Data                                                                        |
================================================================================================
 Area   Branch  Voltage          Generation              Load                    RES
 #      #       Mag [pu]    Pg [MW]   Qg [MVAr]   Pd [MW]   Qd [MVAr]   Pres [MW]  Qres [MVAr]
-----   -----   --------   --------   ---------   -------   ---------   ---------  -----------
   1       1     1.046*    87.560    118.622      0.000       0.000        -           -
   1       2     1.094    134.532    299.997      0.000       0.000        -           -
   1       3     1.037     93.126     91.777      0.000       0.000        -           -
   1       4     1.009        -          -        0.000       0.000        -           -
   1       5     1.000        -          -       90.000      30.000     40.000       -8.082
   1       6     1.016        -          -        0.000       0.000        -           -
   1       7     1.002        -          -      100.000      35.000        -           -
   1       8     1.015        -          -        0.000       0.000        -           -
   1       9     0.983        -          -      125.000      50.000        -           -
   2       1     1.053*   168.897      5.726      0.000       0.000     35.000        8.989
   2       2     1.027     32.396     26.193     21.700      12.700        -           -
   2       3     0.990      0.000     25.297     94.200      19.000        -           -
   2       4     0.997        -          -      47.800      -3.900        -           -
   2       5     1.000        -          -       7.600       1.600        -           -
   2       6     1.042      0.000     21.712     11.200       7.500        -           -
   2       7     1.034        -          -        0.000       0.000        -           -
   2       8     1.060      0.000     15.435      0.000       0.000        -           -
   2       9     1.028        -          -       29.500      16.600        -           -
   2      10     1.023        -          -        9.000       5.800        -           -
   2      11     1.029        -          -        3.500       1.800        -           -
   2      12     1.027        -          -        6.100       1.600        -           -
   2      13     1.022        -          -       13.500       5.800        -           -
   2      14     1.007        -          -       14.900       5.000        -           -
-----   -----   --------   --------   ---------   -------   ---------   ---------  -----------
 The total generation cost is $10815.00/MWh(€10013.89/MWh)

 ... ... (omitted)

 ================================================================
 |     MTDC Branch Data                                         |
 ================================================================
 Branch  From   To     From Branch    To Branch      Branch Loss
 #       Bus#   Bus#   Flow Pij [MW]  Flow Pij [MW]  Pij_loss [MW]
 ------  -----  -----   ---------      ---------      --------
     1       1      2      31.986        -31.724         0.262
     2       2      3       7.281         -7.268         0.014
     3       1      3      28.014        -27.732         0.282
 ------  -----  -----   ---------      ---------      --------
 The totoal DC network losses is 0.557 MW .

 Execution time is 5.295s .
```

Table 6
Timing and memory monitoring methods across programming languages.

Language	Timing approach	Memory monitoring approach
MATLAB	`tic / toc`	`et_memory_usage()`
Python	`time.time()`	`psutil.Process().memory_info().rss`
Julia	`@elapsed` (from `BenchmarkTools.jl`)	`wmic`
C++	`std::chrono::high_resolution_clock`	`GetProcessMemoryInfo()`

- **Runtime:** Julia achieved the fastest execution times across all test cases, leveraging JIT compilation and highly optimized numerical routines. C++ also exhibited strong performance. Python's runtime was acceptable for smaller problems but became less stable and slower as problem size increased. MATLAB was consistently the slowest, especially in larger cases.
- **Memory Usage:** C++ had the lowest memory use due to manual memory control and low overhead, making it ideal for constrained hardware environments. MATLAB also maintained relatively low memory use through in-place operations. Python's memory usage was moderate, but higher relative memory was observed in small cases due likely to interpreter or external library overhead. Julia consumes the most memory, which likely reflects its design choice to have more memory used to enable faster execution speed and array "fusion".
- **Scalability:** Julia demonstrated excellent scalability with sub-linear growth in execution time, making it highly suitable for large-scale AC/DC OPF applications. C++ also scaled well, though development complexity increases with problem size. Python showed moderate scalability but suffers from too much variability in runtime. MATLAB did not scale significantly and failed to complete the largest test case.

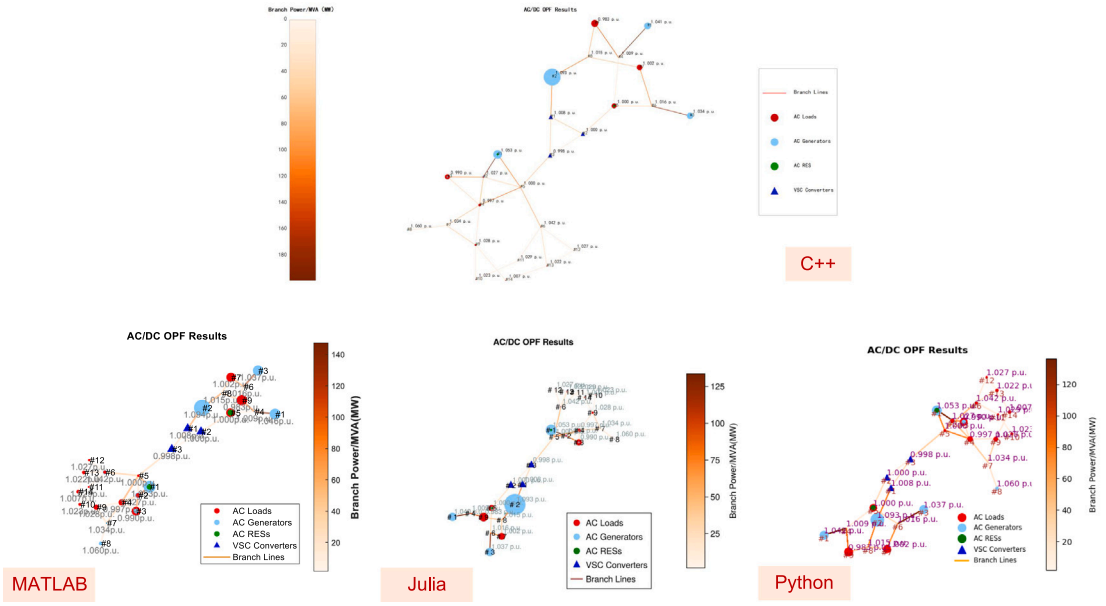

Fig. 6. Visualization of AC/DC OPF results across programming languages. The size of ● is proportional to the AC load level. The size of ● is proportional to the AC generator power level. The size of ● is proportional to the AC RES power level. The color intensity of ▬ indicates the branch power level.

Table 7
Benchmark results summary across programming languages.

Case	Language	Time [s]	Time Range [s]	Memory [MB]	Rel. Mem. [–]
	MATLAB	7.673	7.611–7.732	0.1	1.0x
ac9ac14	C++	0.139	0.133–0.142	0.0	0x
+mtdc3slack_a	Julia	0.055	0.052–0.072	0.1	1.0x
	Python	0.138	0.127–0.182	1.7	17.0x
	MATLAB	157.353	154.715–166.027	5.3	0.14x
ac14ac57	C++	1.780	1.478–2.365	0.5	0.013x
+mtdc3slack_a	Julia	0.233	0.202–0.940	36.8	1.0x
	Python	1.014	0.997–1.054	2.4	0.07x
	MATLAB	177.273	92.259–245.281	14.9	0.12x
ac57ac118	C++	3.464	3.299–4.056	0.8	0.006x
+mtdc3slack_a	Julia	0.799	0.783–0.821	129.4	1.0x
	Python	5.063	4.977–5.240	2.6	0.02x
	MATLAB	–	–	–	–
ac118ac300	C++	38.431	37.126–38.924	0.5	0x
+mtdc3slack_a	Julia	4.446	4.353–5.343	537.7	1.0x
	Python	35.976	28.344–96.248	1.9	0.003x

✧ The case `ac118ac300+mtdc3slack_a` is not implemented successfully for MATLAB.

4. Impact

ACDC-OpFlow is developed with reference to a broad range of well-established techniques in power system modeling and optimization [1,6,11,12,19]. The computational framework incorporates a part of techniques, particularly those related to power flow modeling, convex relaxations, and solver-based optimization, have been previously applied and validated in our previous studies [20,21]. Building upon this technical foundation, *ACDC-OpFlow* establishes a unified, extensible framework for cross-language implementation. In pursuit of the traceability of the solution process, *ACDC-OpFlow* considers only the fundamental system operational constraints. As a result, its overall structure remains simple and is well-suited for early-stage research.

With *ACDC-OpFlow*, users are able to perform AC/DC OPF analysis in the programming language they are most familiar with. The unified framework design allows for seamless transition between programming languages, enabling users to explore other programming languages of interest without significant effort to adapt to unfamiliar syntax. Moreover, the framework encourages users to investigate a variety of research and engineering questions, including but not limited to:

1. How consistent are the modeling formulations, data types, and code structures across different programming languages when implementing AC/DC OPF solving?
2. What are the trade-offs between modeling flexibility, runtime performance, and memory usage in OPF implementations developed in different programming environments?
3. If there is any better approach to organizing dependencies and structuring OPF solvers across programming languages, how might such improvements contribute to more efficient and flexible solution procedure?

At present, *ACDC-OpFlow* remains at an early stage of development. It supports basic OPF functionality aimed at minimizing total generation costs and offers limited options for modeling detailed operational constraints. Future developments will aim to extend the framework by incorporating a broader set of system-level constraints. Community contributions are also welcome to help enrich the functionalities of *ACDC-OpFlow*.

5. Conclusion and future work

This work presents *ACDC-OpFlow*, a cross-language framework for solving AC/DC OPF problems, involved in hybrid AC/VSC-MTDC power systems. *ACDC-OpFlow* distinguishes itself through the key feature: a consistent programming modeling structure across MATLAB, Python, Julia, and C++, which makes it particularly well-suited for direct comparison of modeling syntax, solver behavior, and computational performance across different programming environments. The offered text output and graph-based visualizations, allowing users to clearly observe how generators, RESs, and VSCs collaborate to economically meet load demands.

At present, *ACDC-OpFlow* is in its early stage. Its functionality remains relatively limited and the following two aspects, as the primary plans, are expected to be further improved in future versions:

1. The current *ACDC-OpFlow* is limited to single-time-slot AC/DC OPF calculations and is expected to be extended to support multi-period analysis.
2. More flexible system operation features, such as MTDC reconfiguration, will also be considered in future versions of *ACDC-OpFlow* to better address practical scenarios.

CRediT authorship contribution statement

Haixiao Li: Writing – original draft, Validation, Software, Methodology. **Azadeh Kermansaravi:** Writing – original draft, Validation, Formal analysis, Conceptualization. **Robert Dimitrovski:** Supervision, Project administration, Funding acquisition. **Aleksandra Lekić:** Writing – review & editing, Supervision, Project administration, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Haixiao Li reports financial support was provided by CRESYM Collaborative Research for Energy SYstem Modelling. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the CRESYM project Harmony (https://cresym.eu/harmony/).

Appendix. Representative expressions for defining variables across programming languages

```
%% Variable Definition in Matlab via YALMIP

% 1D Variable (e.g., vn2_dc)
 vn2_dc = sdpvar(nbuses_dc, 1);
% 2D Variable (e.g., pij_dc)
 pij_dc = sdpvar(nconvs_dc, nconvs_dc, 'full');
% 2D Variable (e.g., vn2_ac)
 vn2_ac = cell(ngrids, 1);
 for ng = 1:ngrids
    vn2_ac{ng} = sdpvar(nbuses_ac{ng}, 1);
end
% 3D Variable (e.g., pij_ac)
 pij_ac = cell(ngrids, 1);
 for ng = 1:ngrids
    pij_ac{ng} = sdpvar(nbuses_ac{ng}, nbuses_ac{ng},
  'full');
end
```

```
# Variable Definition in Python via Pyomo

# 1D variable (e.g., vn2_dc)
 model.vn2_dc = Var(range(nbuses_dc), domain=Reals)
# 2D variable (e.g., pij_dc)
 model.pij_dc = Var(range(nbuses_dc), range(nbuses_dc),
   domain=Reals)
# 2D variable (e.g., vn2_ac)
 model.vn2_ac = Var(
    [(ng, i) for ng in range(ngrids) for i in
     range(nbuses[ng])],
    domain=Reals)
# 3D variable (e.g., pij_ac)
 model.pij_ac = Var(
    [(ng, i, j) for ng in range(ngrids) for i in
     range(nbuses[ng]) for j in range(nbuses[ng])],
    domain=Reals)
```

```
# Variable Definition in Julia via JuMP

# 1D Variable (e.g., vn2_dc)
 vn2_dc = @variable(model, [1:nbuses_dc])
# 2D Variable (e.g., pij_dc)
 @variable(model, pij_dc[1:nbuses_dc, 1:nbuses_dc])
# 2D Variable (e.g., vn2_ac)
 vn2_ac = Vector{Vector{JuMP.VariableRef}}(undef,
  ngrids)
 for ng in 1:ngrids
    vn2_ac[ng] = @variable(model, [1:nbuses_ac[ng]])
 end
# 3D Variable (e.g., pij_ac)
 pij_ac = Vector{Matrix{JuMP.VariableRef}}(undef,
  ngrids)
 for ng in 1:ngrids
    pij_ac[ng] = @variable(model, [1:nbuses_ac[ng],
   1:nbuses_ac[ng]])
 end
```

```
// Variable Definition in C++ via Gurobi C++ API

// 1D Variable (e.g., vn2_dc)
 Eigen::Matrix<GRBVar, Eigen::Dynamic, 1>
  vn2_dc(nbuses_dc);
 for (int i = 0; i < nbuses_dc; ++i) {
    vn2_dc(i) = model.addVar(-GRB_INFINITY,
   GRB_INFINITY, 0.0, GRB_CONTINUOUS);
 }
// 2D Variable (e.g., pij_dc)
 Eigen::Matrix<GRBVar, Eigen::Dynamic, Eigen::Dynamic>
  pij_dc(nbuses_dc, nbuses_dc);
 for (int i = 0; i < nbuses_dc; ++i) {
    for (int j = 0; j < nbuses_dc; ++j) {
      pij_dc(i, j) = model.addVar(-GRB_INFINITY,
   GRB_INFINITY, 0.0, GRB_CONTINUOUS);
    }
 }
// 2D Variable (e.g., vn2_ac)
 std::vector<Eigen::Matrix<GRBVar, Eigen::Dynamic, 1>>
  vn2_ac(ngrids);
 vn2_ac[ng].resize(nbuses_ac[ng]);
 for (int i = 0; i < nbuses_ac[ng]; ++i) {
    vn2_ac[ng](i) = model.addVar(-GRB_INFINITY,
   GRB_INFINITY, 0.0, GRB_CONTINUOUS);
 }
// 3D Variable (e.g., pij_ac)
 std::vector<Eigen::Matrix<GRBVar, Eigen::Dynamic,
   Eigen::Dynamic>> pij_ac(ngrids);
```

```
pij_ac[ng] = Eigen::Matrix<GRBVar, Eigen::Dynamic,
↪  Eigen::Dynamic>(nbuses_ac[ng], nbuses_ac[ng]);
for (int i = 0; i < nbuses_ac[ng]; ++i) {
  for (int j = 0; j < nbuses_ac[ng]; ++j) {
    pij_ac[ng](i, j) = model.addVar(-GRB_INFINITY,
↪  GRB_INFINITY, 0.0, GRB_CONTINUOUS);
  }
}
```

References

[1] Beerten J, Cole S, Belmans R. Generalized steady-state VSC MTDC model for sequential AC/DC power flow algorithms. IEEE Trans Power Syst 2012;27(2):821–9. http://dx.doi.org/10.1109/TPWRS.2011.2177867.

[2] Van Hertem D, Gomis-Bellmunt O, Liang J. HVDC grids: for offshore and supergrid of the future. John Wiley & Sons; 2016, http://dx.doi.org/10.1002/9781119115243.

[3] Feng B, Zhao J, Huang G, Hu Y, Xu H, Guo C, et al. Safe deep reinforcement learning for real-time AC optimal power flow: A near-optimal solution. CSEE J Power Energy Syst 2024. http://dx.doi.org/10.17775/CSEEJPES.2023.02070.

[4] Sayed AR, Wang C, Anis HI, Bi T. Feasibility constrained online calculation for real-time optimal power flow: A convex constrained deep reinforcement learning approach. IEEE Trans Power Syst 2022;38(6):5215–27. http://dx.doi.org/10.1109/TPWRS.2022.3220799.

[5] Li C, Kies A, Zhou K, Schlott M, El Sayed O, Bilousova M, et al. Optimal power flow in a highly renewable power system based on attention neural networks. Appl Energy 2024;359:122779. http://dx.doi.org/10.1016/j.apenergy.2024.122779.

[6] Ergun H, Dave J, Van Hertem D, Geth F. Optimal power flow for AC–DC grids: Formulation, convex relaxation, linear approximation, and implementation. IEEE Trans Power Syst 2019;34(4):2980–90. http://dx.doi.org/10.1109/TPWRS.2019.2897835.

[7] Coffrin C, Bent R, Sundar K, Ng Y, Lubin M. PowerModels. JL: An open-source framework for exploring power flow formulations. In: 2018 power systems computation conference. 2018, p. 1–8. http://dx.doi.org/10.23919/PSCC.2018.8442948.

[8] Jat CK, Dave J, Van Hertem D, Ergun H. Hybrid AC/DC OPF model for unbalanced operation of bipolar HVDC grids. IEEE Trans Power Syst 2024;39(3):4987–97. http://dx.doi.org/10.1109/TPWRS.2023.3329345.

[9] ud din GM, Heidari R, Ergun H, Geth F. AC–DC security-constrained optimal power flow for the Australian national electricity market. Electr Power Syst Res 2024;234:110784. http://dx.doi.org/10.1016/j.epsr.2024.110784.

[10] Valerio BC, Lacerda VA, Cheah-Mane M, Gebraad P, Gomis-Bellmunt O. An optimal power flow tool for AC/DC systems, applied to the analysis of the north sea grid for offshore wind integration. IEEE Trans Power Syst 2025;1–14. http://dx.doi.org/10.1109/TPWRS.2025.3533889.

[11] Gan L, Low SH. Optimal power flow in direct current networks. IEEE Trans Power Syst 2014;29(6):2892–904. http://dx.doi.org/10.1109/TPWRS.2014.2313514.

[12] Kocuk B, Dey SS, Sun XA. Strong SOCP relaxations for the optimal power flow problem. Oper Res 2016;64(6):1177–96. http://dx.doi.org/10.1287/opre.2016.1489.

[13] Altun T, Madani R, Davoudi A. Topology-cognizant optimal power flow in multi-terminal DC grids. IEEE Trans Power Syst 2021;36(5):4588–98. http://dx.doi.org/10.1109/TPWRS.2021.3067025.

[14] Bastianel G, Vanin M, Van Hertem D, Ergun H. Optimal transmission switching and busbar splitting in hybrid AC/DC grids. 2024, arXiv preprint arXiv:2412.00270.

[15] Jiang Z, Liu Y, Kang Z, Han T, Zhou J. Security-constrained unit commitment for hybrid VSC-MTDC/AC power systems with high penetration of wind generation. IEEE Access 2022;10:14029–37. http://dx.doi.org/10.1109/ACCESS.2022.3148316.

[16] Acdc_opflow GitHub repository. 2024, https://github.com/CRESYM/ACDC_OPF.

[17] Acdc_opflow GitHub repository manual (PDF). 2024, https://github.com/CRESYM/ACDC_OPF/blob/main/Manual_v0.1.2.pdf.

[18] Ansari JA, Liu C, Khan SA. MMC based MTDC grids: A detailed review on issues and challenges for operation, control and protection schemes. IEEE Access 2020;8:168154–65. http://dx.doi.org/10.1109/ACCESS.2020.3023544.

[19] Altun T, Madani R, Davoudi A. Topology-cognizant optimal power flow in multi-terminal DC grids. IEEE Trans Power Syst 2021;36(5):4588–98. http://dx.doi.org/10.1109/TPWRS.2021.3067025.

[20] Li H, Lekić A. Distributed robust optimization method for AC/MTDC hybrid power systems with DC network cognizant. In: 2024 international conference on smart energy systems and technologies. 2024, p. 1–6. http://dx.doi.org/10.1109/SEST61601.2024.10694436.

[21] Li H, Ergun H, Van Hertem D, Lekic A. Scenario-oriented multi-cut generalized benders decomposition-based distributed OPF for AC/DC hybrid grids. In: 2024 IEEE PES innovative smart grid technologies europe. 2024, p. 1–5. http://dx.doi.org/10.1109/ISGTEUROPE62998.2024.10863082.