Demonstrating SolarPILOT's Python API Through Heliostat Optimal Aimpoint Strategy Use Case

Preprint

William T. Hamilton,[1] Michael J. Wagner,[2] and Alexander J. Zolan[1]

1 National Renewable Energy Laboratory
2 University of Wisconsin-Madison

Presented at the 15th International Conference on Energy Sustainability (ES2021)
Denver, Colorado
June 16-18, 2021

Conference Paper
NREL/CP-5700-78774
April 2021

Demonstrating SolarPILOT's Python API Through Heliostat Optimal Aimpoint Strategy Use Case

Preprint

William T. Hamilton,[1] Michael J. Wagner,[2] and Alexander J. Zolan[1]

1 National Renewable Energy Laboratory
2 University of Wisconsin-Madison

Suggested Citation
Hamilton, William T., Michael J. Wagner and Alexander J. Zolan. 2021. *Demonstrating SolarPILOT's Python API Through Heliostat Optimal Aimpoint Strategy Use Case: Preprint*. Golden, CO: National Renewable Energy Laboratory. NREL/CP-5700-78774. https://www.nrel.gov/docs/fy21osti/78774.pdf.

Conference Paper
NREL/CP-5700-78774
April 2021

National Renewable Energy Laboratory
15013 Denver West Parkway
Golden, CO 80401
303-275-3000 • www.nrel.gov

DEMONSTRATING SOLARPILOT'S PYTHON API THROUGH HELIOSTAT OPTIMAL AIMPOINT STRATEGY USE CASE

William T. Hamilton[*]
Thermal Energy Systems
National Renewable Energy Laboratory
Golden, Colorado 80401
Email: william.hamilton@nrel.gov

Michael J. Wagner
Department of Mechanical Engineering
University of Wisconsin-Madison
Madison, Wisconsin, 53706
Email: mike.wagner@wisc.edu

Alexander J. Zolan
Thermal Energy Systems
National Renewable Energy Laboratory
Golden, Colorado 80401
Email: alexander.zolan@nrel.gov

ABSTRACT

SolarPILOT is a software package that generates solar field layouts and characterizes the optical performance of concentrating solar power (CSP) tower systems. SolarPILOT was developed by the National Renewable Energy Laboratory (NREL) as a stand-alone desktop application but has also been incorporated into NREL's[1] *System Advisor Model (SAM) in a simplified format. Prior means for user interaction with SolarPILOT have included the application's graphical interface, the SAM routines with limited configurability, and through a built-in scripting language called "LK." This paper presents a new, full-featured, Python-based application programmable interface (API) for SolarPILOT, which we hereafter refer to as CoPylot.*

CoPylot provides access to all SolarPILOT's capabilities to generate and characterize power tower CSP systems seamlessly through Python. Supported capabilities include (i) creating and destroying a model instance with message reporting tools; (ii) accessing and setting any SolarPILOT variable including custom land boundaries for field layouts; (iii) programmatically managing receiver and heliostat objects with varied attributes for systems with multiple receiver or heliostat types; (iv) generating, assigning, and modifying solar field layouts including the ability to set individual heliostat locations, aimpoints, soiling rates, and reflectivity levels; (v) simulating solar field performance; (vi) returning detailed results describing performance of individual heliostats, the aggregate field, and receiver flux distribution; and, (vii) exporting Python-based model instances to multiple file formats.

CoPylot enables Python users to perform detailed CSP tower analysis utilizing either the Hermite expansion technique (analytical) or the SolTrace ray-tracing engine. In addition to CoPylot's functionality, Python users have access to the over 100,000 open-source libraries to develop, analyze, optimize, and visualize power tower CSP research. This enables CSP researchers to perform analysis that was previously not possible through SolarPILOT's existing interfaces. This paper discusses the capabilities of CoPylot and presents a use case wherein we demonstrate optimal solar field aiming strategies.

NOMENCLATURE

API	Application Programmable Interface
CSP	Concentrating Solar Power
DLL	Dynamic-Link Library
GUI	Graphical User Interface
HALOS	Heliostat and Layout Optimization Software
LK	Language Kit
MILP	Mixed-Integer Linear Program

[*]Address all correspondence to this author.

[1]The Alliance for Sustainable Energy, LLC (Alliance), is the manager and operator of the National Renewable Energy Laboratory (NREL). Employees of the Alliance, under Contract No. DE-AC36-08GO28308 with the U.S. Dept. of Energy, have authored this work. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this work, or allow others to do so, for United States Government purposes.

NREL	National Renewable Energy Laboratory
SAM	System Advisor Model

INTRODUCTION

Power tower concentrating solar power (CSP) systems consist of a *solar field*, or a field containing hundreds or thousands of *heliostats,* or devices that track the sun and contain individual mirrored surfaces that reflect incoming solar irradiation onto a receiver. Analysis of power tower CSP systems requires a detailed representation and characterization of the field's geometric layout and optical performance. One method to generate and evaluate a solar field is to use the National Renewable Energy Laboratory's (NREL's) SolarPILOT software [1]. However, in previous releases, SolarPILOT offers limited user interfaces which restrict the software's usability and flexibility to perform detailed solar field analysis conjointly with other CSP researchers' modeling efforts.

Within the CSP research community, there exists many software programs to analyze optical performance of CSP configurations [2, 3]. However, to the authors knowledge an open-source, Python compatible, computationally efficient software to perform heliostat layout and field optical performances does not exist. sbpRay, developed by Schlaich Bergermann Partner, is a parallelized ray-tracing software that is accessible through a bespoke graphical user interface (GUI) or a Python interface [4]. sbpRay is a commercially developed software and not open-source. In addition, sbp developed a Python wrapper around SolTrace to predict the optical performance of a bifacial photovoltaic system [5, 6]. Tracer and OTSun are open-source Python libraries that perform Monte Carlo ray tracing and have been validated against other optical performance models [7, 8]. However, ray tracing can be computationally expensive as the optical system scale increases, e.g., increasing the number of heliostats.

SolarPILOT is an open source, C++ software tool that generates solar field layouts and characterizes the optical performance of CSP tower systems. SolarPILOT can simulate receiver flux distributions using two methods: (i) a Hermite expansion technique (analytical) and (ii) a ray-tracing technique called SolTrace^{TM} [6]. The Hermite method enables SolarPILOT to accurately simulate large solar fields in a quick and computationally-efficient manner, while SolTrace provides a robust Monte-Carlo-based ray-tracing method that allows cross-comparison of results and analysis of more complex geometries.

SolarPILOT is a well known and frequently used software in the CSP research community. Previously, SolarPILOT users were limited to interacting with the software through either the GUI or Language Kit (LK) application programming interface (API), shown in Fig. 1 on the left-hand side. The GUI provides an interactive visual method for users to update variables and explore results; however, creating multiple cases and performing large parametric analysis can be cumbersome and time consuming. The LK-API overcomes these challenges by providing the ability to create scripts that execute SolarPILOT's computational methods. However, LK is a domain-specific language that requires SolarPILOT's GUI to operate in the background, has a limited documentation and user support network, and has very limited capabilities for data analysis and visualization; therefore, users typically are required to develop a LK script to perform a specific SolarPILOT analysis and export those results to a more flexible programming language, e.g., Python [9]. Additionally, users can utilize SolarPILOT's computational methods is through System Advisor Model (SAM), which enables receiver fluid flow path calculations [10]; however, this interface is even more restrictive than the above two methods. To address this problem, we developed a full-featured, Python-based API for SolarPILOT, which we hereafter refer to as CoPylot.

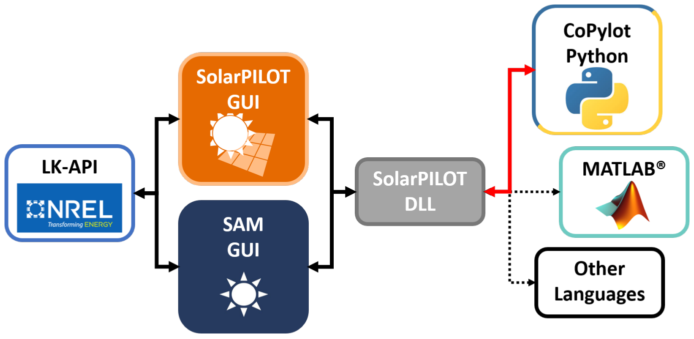


FIGURE 1. A DIAGRAM PRESENTING USER INTERFACES FOR SOLARPILOT'S DYNAMIC-LINK LIBRARY (DLL).

In this paper, we present the architecture and capabilities of CoPylot. We provide a brief description of how users can access CoPylot. Then, we present a working example using CoPylot to generate a solar field, simulate performance, and access SolarPILOT results. Next, we present an aimpoint optimization use case that utilizes CoPylot to provide solar field layout and characterization of heliostat flux images onto the receiver. Lastly, we conclude with a summary and extensions of our work.

COPYLOT DESCRIPTION

CoPylot is a full-featured, Python-based API for SolarPILOT that directly interacts with SolarPILOT's dynamic-link library (DLL), shown in Fig. 1 on the right-hand side. CoPylot enables users access to SolarPILOT's computation engines seamlessly through the Python scripting interface. This new capability of SolarPILOT provides a versatile tool to CSP tower researchers to generate heliostat layouts and characterize field performance through Python or even embed SolarPILOT into other research tools. For example, CoPylot could be integrated into a model investigating optimal receiver design or operations which is outside of the current scope of SolarPILOT's capabilities. Additionally, CoPylot users have access to the over 100,000 open-source Python libraries to develop, analysis, optimize, and visualize CSP tower research.

To develop CoPylot, we create new SolarPILOT C++ source code to export functions from its DLL. CoPylot accesses the DLL exported functions utilizing *ctypes*, a the foreign function library for Python. CoPylot manipulates Python user inputs to compatible C data types for interfacing with the C++ SolarPILOT DLL and converts DLL return information to Python specific data structures; thereby, increasing CoPylot's usability. In addition to being accessed by Python through the CoPylot interface, SolarPILOT's DLL exported functions may be accessed by other scripting languages, e.g., MATLAB®, by creating an API within the specific language to handle data type conversions, shown in Fig. 1. While we did not create these links within this work, the development of these APIs would be straightforward with the existing C++ source code, and could be developed in future work.

CoPylot enables users to access all of SolarPILOT's functionality through a Python scripting interface. A brief summary of CoPylot's functionality includes:

1. Creating and destroying model instances which includes callback functionally to propagate messages from SolarPILOT back to Python users
2. Accessing and setting SolarPILOT's variables including importing custom land boundaries for field layout
3. Managing receiver and heliostat objects with varied attributes for systems with multiple receiver or heliostat types
4. Generating, assigning, and modifying solar field layouts including the ability to set individual heliostat locations, aimpoints, soiling rates, and reflectivity level
5. Simulating solar field performance
6. Returning detailed results describing performance of individual heliostat performance, the aggregated field, and receiver flux distribution
7. Exporting the python created SolarPILOT instance to either a *.csv* file with all variable values or a SolarPILOT *.spt* file enabling the instance to be loaded by SolarPILOT's GUI

Getting Started with CoPylot

To start using CoPylot, download *copylot.py* and *solarpilot.dll* available through the SolarPILOT Github repository. Currently, CoPylot is accessible through the SolarPILOT *copilot* branch within `./deploy/api` directory; however, this branch will be merged into *develop* in future versions of SolarPILOT. When this merge occurs, we will provide users detailed instructions in SolarPILOT's `README` describing the process for accessing CoPylot. Additionally, there is a CoPylot test script within `./deploy/api` that provides example code using the CoPylot Python class. CoPylot assumes the DLL file exists in the same folder as *copylot.py*; if not the case, users will need to update CoPylot's path to the DLL file within the CoPylot class.

Model Building with CoPylot

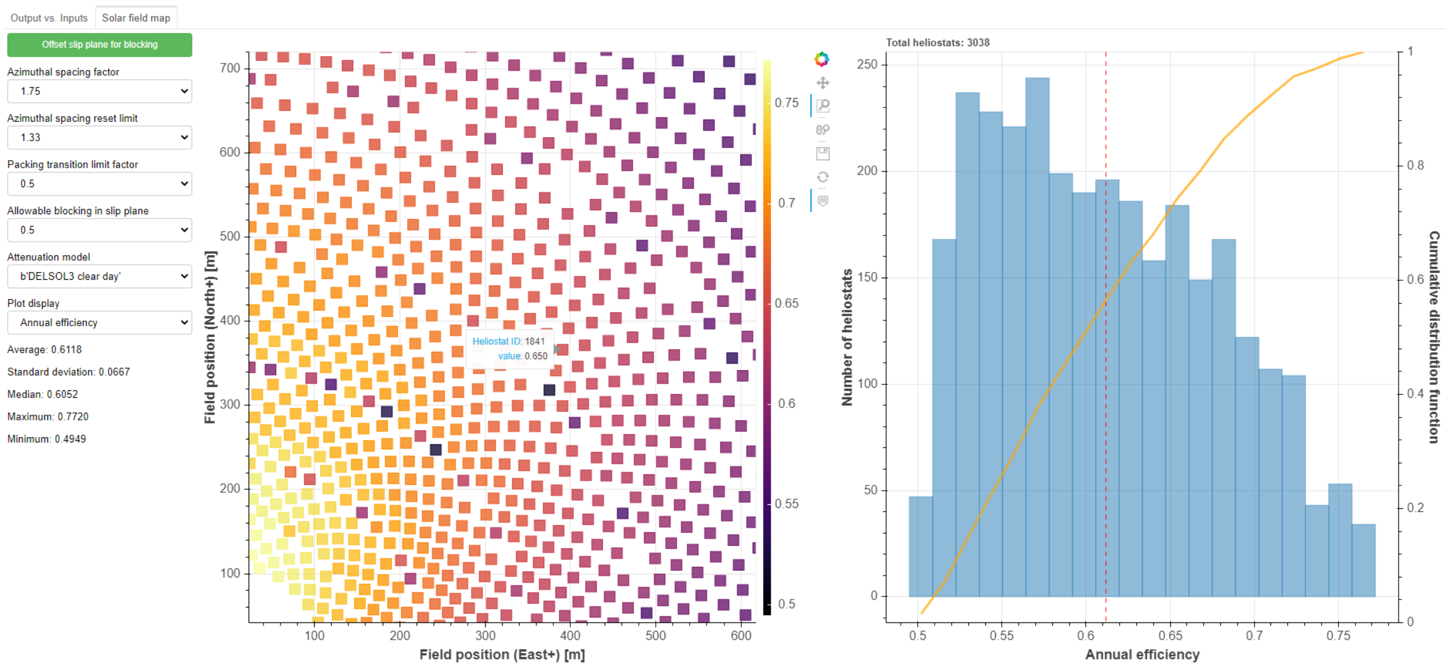
The following is Python code presenting a working example using CoPylot to generate a solar field, simulate performance, and access solarPILOT results.

```
from copylot import CoPylot
cp = CoPylot()  # create a CoPylot class instance
r = cp.data_create()  # create a SolarPILOT instance
cp.api_callback_create(r)   # create callback
cp.data_set_string(r,
  ''ambient.0.weather_file'',
  ${PATH TO WEATHER FILE})  # set path to weather file
print(cp.generate_layout(r))  # generate layout
field = cp.get_layout_info(r)  # get layout
print(cp.simulate(r))  # simulate field performance
flux = cp.get_fluxmap(r)  # get receiver flux
lay_res = cp.detail_results(r)  # get layout results
summary = cp.summary_results(r)  # get system summary
cp.data_free(r)  # free SolarPILOT instance
```

In this example, the solar field is generated and simulated using SolarPILOT default variable values. The first step when working with CoPylot is to import the class from *copylot.py* and create a class instance. The CoPylot class contains all of the methods for interacting with SolarPILOT's DLL. When `data_create()` is called, CoPylot creates an `api_helper` data structure within the DLL to store a SolarPILOT instance's variables, solar field, and results. This method returns a pointer that is utilized by other CoPylot methods to access the specific instance of SolarPILOT. As a result of this methodology, the user can create and manipulate multiple SolarPILOT instances, simultaneously, using their unique memory pointers. This enables easy implementation of parallel computation in the Python interface, e.g., using the *multiprocessing* package.

By default, CoPylot callback functionality is disabled to suppress console messages when CoPylot is embedded into other research modeling tools. However, users may enable the callback by using `api_callback_create()`. This method provides a link between CoPylot and the DLL which allows the latter to return messages to the Python console. This callback can be very useful when working with CoPylot for the first time as it provides users with detailed error messages for common mistakes, e.g., trying to set a variable with the wrong name or data type. To disable the callback, users can call the `api_disable_callback()` method.

When CoPylot creates a SolarPILOT instance, it sets all the variables to their default values except for the weather file path. We designed CoPylot to exist independent of SolarPILOT's GUI and its installed directory which includes `climate_files` containing a collection of location-specific weather files. As a result, the user is required to set `"ambient.0.weather_file"` to a



Azimuthal spacing factor: 1.75
Azimuthal spacing reset limit: 1.33
Packing transition limit factor: 0.5
Allowable blocking in slip plane: 0.5
Attenuation model: b'DELSOL3 clear day'
Plot display: Annual efficiency

Average: 0.6118
Standard deviation: 0.0667
Median: 0.6052
Maximum: 0.7720
Minimum: 0.4949

FIGURE 2. SCREENSHOT OF A INTERACTIVE BOKEH HELIOSTAT FIELD PLOTTING TOOL.

weather file path using the appropriate variable setter method, `data_set_string()`, before field generation. This weather file must conform with the formats described in the SolarPILOT's documentation. A complete list of SolarPILOT variable names can be found through the LK scripting tool accessed through SolarPILOT's GUI (*File* → *New Script* → *Help*). We plan to improve the variable naming documentation in future work.

Once the user has updated variable values as desired, they can run `generate_layout()` to generate a solar field layout. This method returns a Boolean to specify if the process was successful. This method is equivalent to pressing the "Generate New Layout" on the Field Layout page in the SolarPILOT's GUI. `get_layout_info()` returns the solar field layout as a *Pandas* DataFrame which contains each heliostat's x-, y-, and z- coordinates as well as a unique ID number, heliostat template ID, and layout metric. This method has additional keyword functionality which can (i) change the return format to either dictionary (`restype = "dict"`) or a matrix and header lists (`restype = "mat"`), and/or (ii) provide the corner coordinates for each heliostat reflective surface (`get_corners = True`).

CoPylot's `simulate()` simulates performance using the stored solar field, specified sun position, and simulation parameters (equivalent to pressing the "simulate performance" button on the Performance Simulation page in the SolarPILOT's GUI). Similar to `generate_layout()`, `simulate()` returns a Boolean to specify if the method was successful. After simulation, users can access the receiver flux distribution using `get_fluxmap()` which returns a matrix (i.e., list of lists).

CoPylot's `detail_results()` provides the detailed simulations results for each heliostat in a *Pandas* DataFrame (by default). This DataFrame contains all of the information typically found on the Layout Results page in the SolarPILOT GUI. This method contains all of the keyword arguments described for `get_layout_info()`, as well as a way to select specific heliostats using a list of heliostat ID numbers (`selhel=[]`). With the output of `detail_results()`, users can analyze and visualize the solar field performance metrics using any of the available open-source Python libraries. For example, we created an interactive Bokeh heliostat field plotting tool that allows users to: i) change the field performance metric being displayed, ii) zoom in to and highlight specific heliostats within the field, and iii) view overall heliostat field statistics and distribution of performance for the specific metric of interest, shown in Fig. 2 [11]. Currently, this Bokeh application has not been released to the public; however, a version of this application may be released during future development.

CoPylot's `summary_results()` returns a dictionary of system summary results from each simulation which is equivalent to table presented on System Summary page of the SolarPILOT's GUI. This table can be printed to console using the keyword argument `save_dict=False`. Lastly, when the desired computation is completed, it is important to free the SolarPILOT instance allocated memory using `data_free()`. This will prevent a potential memory leak during multi-threading processes.

The purpose of presenting this working example of CoPylot is to provide a basic understanding of using CoPylot to create SolarPILOT instances, generate solar fields, and simulate field performance. This example is by no means comprehensive and does not present all of CoPylot's functionality. For further documentation, CoPylot's methods use docstrings to provide users with details about each method's purpose, parameters, and returns.

AIMPOINT OPTIMIZATION USE CASE

We demonstrate the usefulness of the CoPylot library to the CSP community by describing a use case in which the SolarPILOT Python API is used to support an aimpoint optimization tool that is implemented in Python. The tool, Heliostat and Layout Optimization Software (HALOS) [12], generates layouts using SolarPILOT and obtains optimized aimpoint strategies by solving a mixed-integer linear program. This implementation differs from other aimpoint strategy optimization methods using integer programming methods [13, 14] by separating a solar field into sections whose flux images and aimpoint strategies are optimized in parallel, then aggregated into a final solution.

Figure 3 describes the procedure HALOS uses to generate and then solve instances of its optimization model, which is implemented in Pyomo [15, 16]. If a solar field is provided, HALOS can call SolarPILOT to obtain the individual heliostat flux images by simulating with only one heliostat enabled in the field via `modify_heliostats()`, `simulate()`, and `get_fluxmap()` methods; these individual images serve as input to the HALOS aimpoint strategy optimization model. Alternatively, if no solar field is provided but the required information to generate a solar field is provided to HALOS, it uses CoPylot to generate a solar field layout via `generate_layout()`, then produce the flux images.

The default method of flux characterization in HALOS is Gaussian when SolarPILOT is not used, and so we compare the computing times when using SolarPILOT to calculate the flux images versus using HALOS directly, in which the former uses the Hermite method to develop a single image after aggregating different sources of optical error. By utilizing the flux image processing library in SolarPILOT via the CoPylot library, HALOS has direct access to high-fidelity flux characterization methods without re-implementation of those methods. Additionally, using CoPylot to generate model instances was about 2-4× faster than the analogous flux model in HALOS for a collection of four separate cases, as shown in Table 1; these cases were generated using a Dell Laptop with an Intel Core i7-8650 CPU and 16GB RAM. The results demonstrate the computational benefit associated with using SolarPILOT's more complex but faster C++ implementation versus performing inverse-Gaussian calculations in Python.

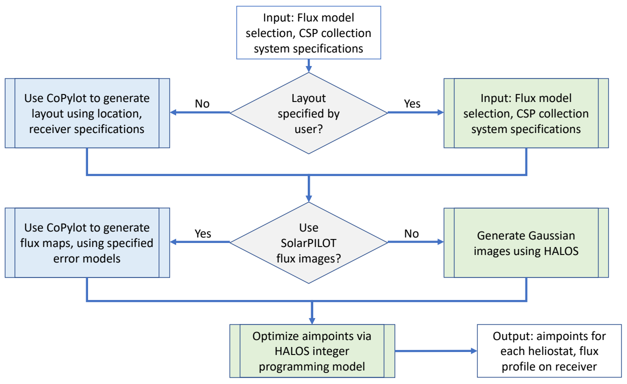

Input: Flux model selection, CSP collection system specifications
Use CoPylot to generate layout using location, receiver specifications
No
Layout specified by user?
Yes
Input: Flux model selection, CSP collection system specifications
Use CoPylot to generate flux maps, using specified error models
Yes
Use SolarPILOT flux images?
No
Generate Gaussian images using HALOS
Optimize aimpoints via HALOS integer programming model
Output: aimpoints for each heliostat, flux profile on receiver

FIGURE 3. FLOWCHART DESCRIBING HALOS GENERATIONS AND SOLUTION OF PROBLEMS, INCLUDING INTERACTIONS WITH COPYLOT. BLUE BOXES DENOTE TASKS THAT ARE PERFORMED USING SOLARPILOT VIA THE COPYLOT LIBRARY, WHILE GREEN BOXES DENOTE TASKS PERFORMED DIRECTLY BY HALOS.

TABLE 1. COMPARISON OF COMPUTING TIME USING HALOS WITH AND WITHOUT COPYLOT TO GENERATE FLUX MAPS TO SERVE AS INPUT TO ITS AIMPOINT OPTIMIZATION MODEL, FOR A COLLECTION OF FOUR CASES WITH VARYING GEOMETRIES WITH DAGGETT, CA AS THE SITE LOCATION. SOLARPILOT GENERATED THE FIELD IN ALL INSTANCES.

		Computing Time (s)		
Geometry	**# Heliostats**	**HALOS w/ CoPylot**	**HALOS wo/ CoPylot**	**Improvement factor**
Flat	614	37	142	3.84
Flat	3025	185	442	2.39
Cylindrical	681	41	112	2.73
Cylindrical	3442	205	485	2.37

CONCLUSIONS

CoPylot is an open-source, computationally efficient Python API for SolarPILOT which enables users to generate and simulate power tower solar field optical performance. In this paper, we presented the architecture and capabilities of CoPylot, provided users the location to access CoPylot, presented a basic CoPylot working example, and described an aimpoint optimization use case that utilizes CoPylot.

CoPylot provides CSP researchers access to SolarPILOT's computational methods within the Python framework. We believe CoPylot will increase SolarPILOT usability and enables researchers to quickly perform CSP power tower modeling with minimum overhead using SolarPILOT. In addition, CoPylot users have access to the over 100,000 open-source Python libraries to develop, analyze, optimize, and visualize CSP tower research.

With the release of CoPylot, we hope CSP researchers find the API useful and encourage them to provide feedback about their user experience. In future work, we will provide CoPylot users access to addition functionality developed within SolarPILOT. In addition, researchers can develop other language API's using the exported DLL functions developed in this work, thereby increasing SolarPILOT accessibility.

ACKNOWLEDGMENT

The authors would like to acknowledge graduate intern Kashif Liaqat for beta testing CoPylot. This work was funded by the United States Department of Energy – Energy Efficiency and Renewable Energy under award numbers DE-EE00035930.

REFERENCES

[1] Wagner, M. J., and Wendelin, T., 2018. "SolarPILOT: A power tower solar field layout and characterization tool". *Solar Energy,* ***171***, pp. 185–196.

[2] Ho, C. K., 2008. Software and codes for analysis of concentrating solar power technologies. Tech. Rep. SAND2008-8053, Sandia National Laboratories.

[3] Li, L., Coventry, J., Bader, R., Pye, J., and Lipiński, W., 2016. "Optics of solar central receiver systems: A review". *Optics express,* ***24***(14), pp. A985–A1007.

[4] Gebreiter, D., Weinrebe, G., Wöhrbach, M., Arbes, F., Gross, F., and Landman, W., 2019. "sbpRAY – a fast and versatile tool for the simulation of large scale CSP plants". In AIP Conference Proceedings, Vol. 2126, AIP Publishing LLC, p. 170004.

[5] Gross, F., Luengo, M., Hennings, L., Landman, W., and Balz, M., 2020. "The impact of tracker structure on bifacial PV performance".

[6] Wendelin, T., 2003. "SolTRACE: A new optical modeling tool for concentrating solar optics". In International Solar Energy Conference, Vol. 36762, pp. 253–260.

[7] Wang, Y., Asselineau, C.-A., Coventry, J., and Pye, J., 2016. "Optical performance of bladed receivers for CSP systems". In ASME 2016 10th International Conference on Energy Sustainability, Vol. 50220, American Society of Mechanical Engineers, p. V001T04A026.

[8] Cardona, G., and Pujol-Nadal, R., 2020. "OTSun, a python package for the optical analysis of solar-thermal collectors and photovoltaic cells with arbitrary geometry". *Plos one,* ***15***(10), p. e0240735.

[9] Python language reference (version 3.9.1).

[10] Blair, N. J., DiOrio, N. A., Freeman, J. M., Gilman, P., Janzou, S., Neises, T. W., and Wagner, M. J., 2018. System advisor model (SAM) general description (version 2017.9.5). Tech. Rep. NREL/TP–6A20–70414, National Renewable Energy Laboratory.

[11] Bokeh Development Team, 2018. *Bokeh: Python library for interactive visualization*.

[12] Zolan, A., Hamilton, W., Liaqat, K., and Wagner, M., 2021. *Heliostat Layout and Aimpoint Optimization Software (HALOS)*.

[13] Ashley, T., Carrizosa, E., and Fernández-Cara, E., 2017. "Optimisation of aiming strategies in solar power tower plants". *Energy,* ***137***, pp. 285–291.

[14] Kuhnke, S., Richter, P., Kepp, F., Cumpston, J., Koster, A. M., and Büsing, C., 2020. "Robust optimal aiming strategies in central receiver systems". *Renewable Energy,* ***152***, pp. 198–207.

[15] Hart, W. E., Watson, J.-P., and Woodruff, D. L., 2011. "Pyomo: modeling and solving mathematical programs in python". *Mathematical Programming Computation,* ***3***(3), pp. 219–260.

[16] Hart, W. E., Laird, C. D., Watson, J.-P., Woodruff, D. L., Hackebeil, G. A., Nicholson, B. L., and Siirola, J. D., 2017. *Pyomo–optimization modeling in python*, second ed., Vol. 67. Springer Science & Business Media.