Check for updates

Demonstrating SOLARPILOT's Python Application Programmable Interface Through Heliostat Optimal Aimpoint Strategy Use Case

William T. Hamilton[1]
Thermal Energy Systems,
National Renewable Energy Laboratory,
15013 Denver W Pkwy,
Golden, CO 80401
e-mail: william.hamilton@nrel.gov

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

Alexander J. Zolan
Thermal Energy Systems,
National Renewable Energy Laboratory,
Golden, CO 80401
e-mail: alexander.zolan@nrel.gov

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 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 article presents a new, full-featured, PYTHON-based application programmable interface (API) for SOLARPILOT, which we hereafter refer to as CoPylot. CoPylot enables PYTHON users to perform detailed CSP tower analysis utilizing either the Hermite expansion technique (analytical) or the SolTrace ray-tracing engine. CoPylot's enables CSP researchers to perform analysis that was previously not possible through SOLARPILOT's existing interfaces. This article discusses the capabilities of CoPylot and presents a use case wherein we populate a model that obtains optimal solar field aiming strategies. [DOI: 10.1115/1.4053973]

Keywords: SOLARPILOT, solar power tower, heliostat layout, optical characterization, heliostat field modeling, optics, simulation, concentrating solar power

1 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. The 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, previous releases of SOLARPILOT offer 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]. Tonatiuh [4] is an open-source performance characterization tool that uses Monte Carlo ray tracing (MCRT) and is implemented in C++. SOLSTICE [5] is a standalone executable release that utilizes an integrated formulation Monte Carlo algorithm to obtain the solar flux on the receiver. Heliosim [6] includes a graphical user interface (GUI) and performs MCRT while leveraging the parallel capabilities of GPUs, utilizing NVIDIA's OptiX Ray Tracing Engine. Similarly, TieSOL [7,8] optimizes both layouts and aiming strategies for CSP tower plants by leveraging GPUs in parallel computing to obtain flux maps. In addition to MCRT, other software packages utilize analytical, convolution-based methods to obtain flux maps efficiently; see, e.g., Refs. [9–13]. Further comparisons between these models are available in Ref. [14]. 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 GUI or a PYTHON interface [15]. SBPRAY is a commercially developed software and not open source. Tracer and OTSun are open-source PYTHON libraries that perform Monte Carlo ray tracing and have been validated against other optical performance models [16,17]. However, ray tracing can be computationally expensive as the optical system scale increases, e.g., increasing the number of heliostats.

Heliostat field layout and receiver sizing is an important step when designing and optimizing a power tower CSP system. Grigoriev et al. described an approach that uses FluxTracer to optimize a cylindrical receiver sizing by computing an annual ray-tracing simulation and then postprocessing receiver sizing impacts on annual energy collected [18]. The authors conclude that their design space has a global optimum; however, the space is shallow with respect to their design variables, i.e., receiver height and diameter. Ortega et al. used SENSOL evaluate a tower CSP plant profitability for a sensitivity analysis, which included tower height, receiver diameter, and receiver height [19]. The work we present in this article can be applied to similar problems within the CSP industry and research communities.

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™ [20]. The Hermite method enables SOLARPILOT to accurately simulate large solar fields in a quick and computationally efficient manner,

[1]Corresponding author.

Contributed by the Solar Energy Division of ASME for publication in the JOURNAL OF SOLAR ENERGY ENGINEERING: INCLUDING WIND ENERGY AND BUILDING ENERGY CONSERVATION. Manuscript received September 10, 2021; final manuscript received February 23, 2022; published online March 18, 2022. Assoc. Editor: Nesrin Ozalp.

This work is in part a work of the U.S. Government. ASME disclaims all interest in the U.S. Government's contributions.

Downloaded from http://verification.asmedigitalcollection.asme.org/solarenergyengineering/article-pdf/144/3/030906/6871816/sol_144_3_030906.pdf by guest on 18 July 2024

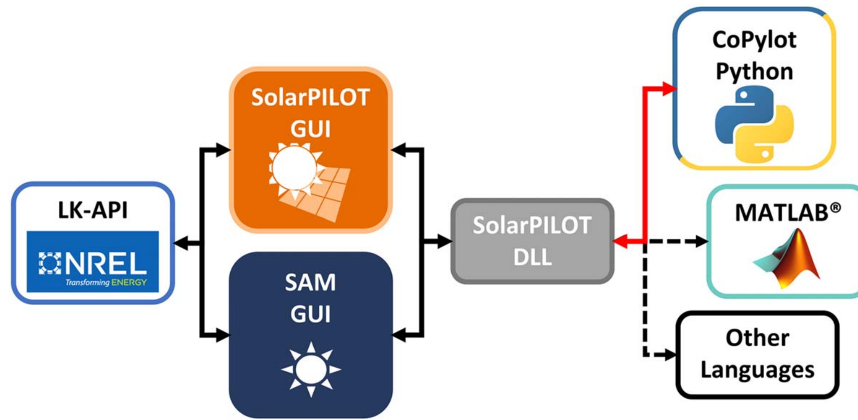

Fig. 1 A diagram presenting user interfaces for SOLARPILOT's DLL

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 software through either the GUI or Language Kit (LK) application programming interface (API), as 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 [21]. In addition, users can utilize SOLARPILOT's computational methods through system advisor model (SAM), which enables receiver fluid flow path calculations [22]; however, this interface is even more restrictive than the aforementioned two methods. To address this problem, we developed a full-featured, PYTHON-based API for SOLARPILOT, which we hereafter refer to as CoPylot.

In this article, 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. Finally, we conclude with a summary and extensions of our work.

2 CoPylot Description

CoPylot is a full-featured, PYTHON-based API for SOLARPILOT that directly interacts with SOLARPILOT's dynamic-link library (DLL), as shown in Fig. 1 on the right-hand side. The flow of information when using any function in CoPylot is as follows:

(1) A PYTHON-based function within CoPylot is called.
(2) The analogous C++ function within the SOLARPILOT DLL is executed.
(3) The C++ outputs from the DLL are returned in PYTHON.

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. In addition, 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 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, as 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 that include call-back 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, aim-points, 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

The Appendix presents a complete list of CoPylot's supported function declarations with brief descriptions. These functions provide users access to SOLARPILOT's modeling capability through PYTHON and enable a flexible interface that can be

Downloaded from http://verification.asmedigitalcollection.asme.org/solarenergyengineering/article-pdf/144/3/030906/6871816/sol_144_3_030906.pdf by guest on 18 July 2024

applied to the user's specific application of interest. Please refer to the CoPylot PYTHON class source code for more detail on function arguments and return values. All interfaces with CoPylot should start with `data_create()` and end with `data_free()`, which creates and frees an instance of SOLARPILOT in memory, respectively.

2.1 Getting Started With CoPylot.

To start using CoPylot, download SOLARPILOT available for Windows or Linux through the NREL website. For the Windows configuration, follow the installer to complete installation of SOLARPILOT. After installation is complete, navigate to the SOLARPILOT 1.4.0 directory (default location: `C:\Program Files\SolarPILOT\1.4.0`), where you will see the `api` folder that contains everything needed to use CoPylot. The minimum set of files required to use CoPylot are *copylot.py* and *solarpilot.dll*. Users can copy and paste these files into their specific projects that want to interact with CoPylot. CoPylot assumes that 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 (contained in *copylot.py*). In addition, there is a CoPylot test script (*test_script.py*) and a minimum working example (*min_example.py*) that provides example code using the CoPylot PYTHON class. For the Linux version, everything stated earlier holds true except that SOLARPILOT's download will be a tarball gzip file (i.e., `.tar.gz` extension) and CoPylot will access SOLARPILOT using a `.so` dynamic library rather than the Windows `.dll`.

2.2 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.

```
1   from copylot import CoPylot
2   cp = CoPylot()   # create a CoPylot class instance
3   r = cp.data_create()   # create a SolarPILOT instance
4   cp.api_callback_create(r)   # create callback
5   cp.data_set_string(r,
6   "ambient.0.weather_file",
7   ${PATH TO WEATHER FILE})   # set path to weather file
8   print(cp.generate_layout(r))   # generate layout
9   field = cp.get_layout_info(r)   # get layout
10  print(cp.simulate(r))   # simulate field performance
11  flux = cp.get_fluxmap(r)   # get receiver flux
12  lay_res = cp.detail_results(r)   # get layout results
13  summary = cp.summary_results(r)   # get system summary
14  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 that 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 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 the 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 [23], 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) [23]. 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, as shown in Fig. 2 [24]. Currently, this Bokeh application has not been released to the public; however, a version of this application may be released during the 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`. Finally, 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

Downloaded from http://verification.asmedigitalcollection.asme.org/solarenergyengineering/article-pdf/144/3/030906/6871816/sol_144_3_030906.pdf by guest on 18 July 2024

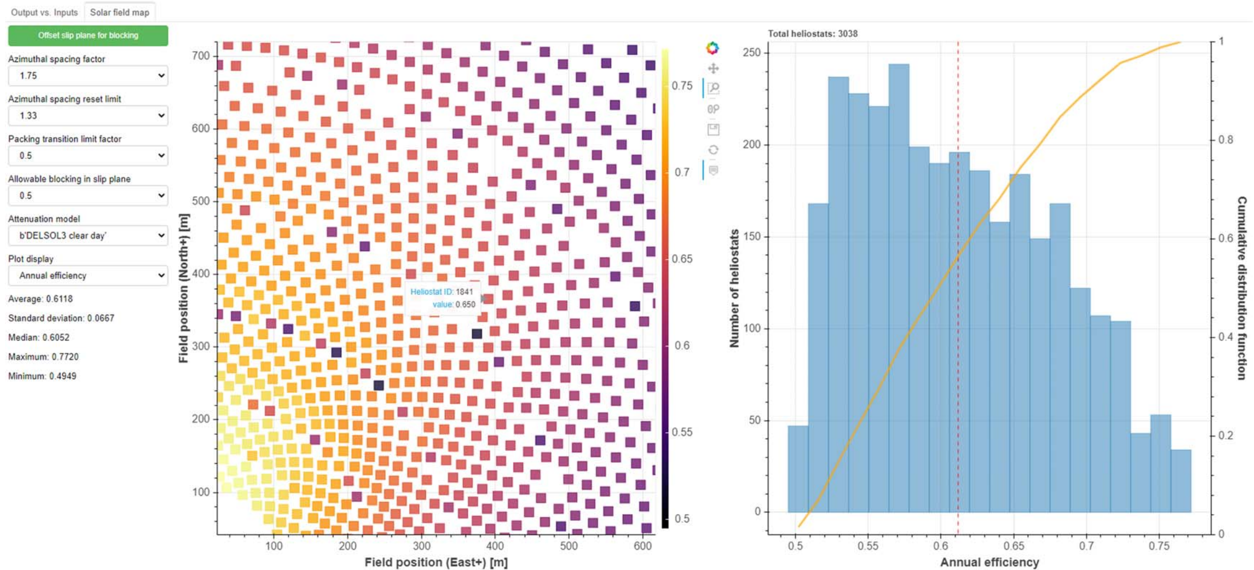

Fig. 2 Screenshot of an interactive Bokeh heliostat field plotting tool

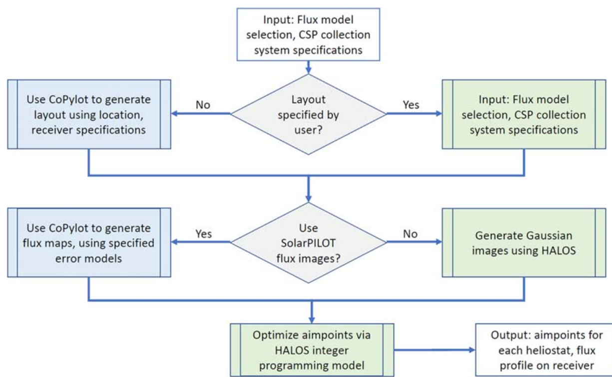

Fig. 3 Flowchart describing HALOS generation and solution of problems, including interactions with CoPylot. The shaded boxes on the left denote tasks that are performed use SOLARPILOT via CoPylot library, while the shaded boxes on the right and bottom denote tasks performed directly by HALOS.

methods use docstrings to provide users with details about each method's purpose, parameters, and returns.

3 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) [25], 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 [26,27] by separating a solar field into sections whose flux images and aimpoint strategies are optimized in parallel and then aggregated into a final solution.

3.1 HALOS Description. HALOS[2] is an open-source aimpoint optimization software tool that is developed in PYTHON [25]. Using a field layout and receiver characteristics as input, HALOS generates flux images assuming a centrally located aimpoint on the receiver and then translates the flux images directly across a discretized surface for a collection of aimpoints to obtain candidate aimpoints and flux maps. These then serve as input to a mixed-integer linear program that is implemented in PYOMO, an open-source algebraic modeling language developed at Sandia National Laboratories [28]. The formulation within HALOS enforces receiver flux limits and allows for a discrete collection of aimpoints and measurement points on the receiver, similar to other mixed-integer programming models developed by Ashley et al. [26] and Kuhnke et al. [27]. However, although the other models utilize a single instance to obtain all aimpoints, HALOS subdivides the solar field into sections that may be solved in parallel to reduce computing time for large-scale systems. More information on HALOS and the mathematical formulation are available in Ref. [29].

HALOS connects to SOLARPILOT via the CoPylot library, which allows for a large collection of flux models and plant characteristics to be evaluated using optimized aiming strategies rather than the heuristic employed within SOLARPILOT. In what follows, we describe a collection of case studies that test the computational advantage of using CoPylot to obtain flux images to populate instances of the model in HALOS instead of either using PYTHON or interacting with SOLARPILOT through the filesystem.

3.2 Use Case and Results. Figure 3 describes the procedure HALOS uses to generate and then solve instances of its optimization model, which is implemented in PYOMO [28,30]. 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()` and then to 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

[2]https://github.com/NREL/HALOS

Downloaded from http://verification.asmedigitalcollection.asme.org/solarenergyengineering/article-pdf/144/3/030906/6871816/sol_144_3_030906.pdf by guest on 18 July 2024

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

Flux model	Geometry	Heliostats	Computing time (s) HALOS w/ CoPylot	HALOS wo/ CoPylot	Improvement factor
Gaussian	Flat	614	37	142	3.84
Gaussian	Flat	3025	185	442	2.39
Gaussian	Cylindrical	681	41	112	2.73
Gaussian	Cylindrical	3442	205	485	2.37
Pillbox	Flat	614	42	73	1.74
Pillbox	Flat	3025	215	329	1.53
Pillbox	Cylindrical	681	51	86	1.69
Pillbox	Cylindrical	3442	223	378	1.70

Note: SOLARPILOT generated the field in the all instances.

access to high-fidelity flux characterization methods without re-implementation of those methods. In addition, 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 presented in Table 1. Because HALOS does not calculate non-Gaussian images within its own library, we develop a second collection of case studies for the pillbox flux method by comparing the direct use of CoPylot against using SOLARPILOT to generate individual flux maps, saving each one to the filesystem, and then running HALOS to read in the images from the filesystem to populate the aimpoint optimization model. All cases were generated using a Dell Laptop with an Intel Core i7-8650 CPU and 16GB RAM. The results of the first four cases demonstrate the computational benefit associated with using SOLARPILOT's more complex but faster C++ implementation versus performing inverse-Gaussian calculations in PYTHON, while the results of the second four cases demonstrate the benefit of directly passing the flux images to the optimization model in memory rather than using the filesystem. We note that in the latter four cases, we utilize CoPylot to generate the individual flux maps because doing so through the SOLARPILOT GUI would require a run for each individual heliostat, which would increase the improvement factor by orders of magnitude.

In addition to the aforementioned case studies, we generated a collection of 30 different individual flux images using both the SOLARPILOT GUI and CoPylot. In each case, the flux maps generated were identical; as a result, the accuracy of CoPylot as a flux method is identical to that of SOLARPILOT.

4 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 article, 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 the future work, we will provide CoPylot users access to additional 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

This material is based upon work supported by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Solar Energy Technologies Office Award Number 35930. This paper was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Conflict of Interest

There are no conflicts of interest.

Appendix: CoPylot's Supported Function Declarations

Table 2 presents the list of CoPylot's supported function declarations.

Downloaded from http://verification.asmedigitalcollection.asme.org/solarenergyengineering/article-pdf/144/3/030906/6871816/sol_144_3_030906.pdf by guest on 18 July 2024

Table 2 A complete list of CoPylot's supported function declarations with brief descriptions

CoPylot Function Declaration and Description
data_create(self) → int
Creates an instance of SOLARPILOT in memory
data_free(self, p_data: int) → bool
Frees SOLARPILOT instance (p_data) from memory
version(self, p_data: int) → str
Provides SOLARPILOT version number
api_callback_create(self, p_data: int) → None
Creates a callback function for message transfer
api_disable_callback(self, p_data: int) → None
Disables callback function
data_set_number(self, p_data: int, name: str, value) → bool
Sets a SOLARPILOT numerical variable, used for float, int, bool, and numerical combo options.
data_set_string(self, p_data: int, name: str, svalue: str) → bool
Sets a SOLARPILOT string variable, used for string and combos
data_set_array(self, p_data: int, name: str, parr: list) → bool
Sets a SOLARPILOT array variable, used for double and int vectors
data_set_array_from_csv(self, p_data: int, name: str, fn: str) → bool
Sets a SOLARPILOT vector variable from a csv, used for double and int vectors
data_set_matrix(self, p_data: int, name: str, mat: list) → bool
Sets a SOLARPILOT matrix variable, used for double and int matrix
data_set_matrix_from_csv(self, p_data: int, name: str, fn: str) → bool
Sets a SolarPILOT matrix variable from a csv, used for double and int matrix
data_get_number(self, p_data: int, name: str) → float
Gets a SOLARPILOT numerical variable value
data_get_string(self, p_data: int, name: str) → str
Gets a SOLARPILOT string variable value
data_get_array(self, p_data: int, name: str) → list
Gets a SOLARPILOT array (vector) variable value
data_get_matrix(self,p_data: int,name: str) → list
Gets a SOLARPILOT matrix variable value
reset_vars(self, p_data: int) → bool
Resets SOLARPILOT variable values to defaults
add_receiver(self, p_data: int, rec_name: str) → int
Creates a receiver object
Note: CoPylot starts with a default receiver configuration at receiver object ID = 0, with "Receiver 1" as the receiver's name. If you add a receiver object without dropping this default receiver, generating a layout will result in a multi-receiver problem, which could produce strange results.
drop_receiver(self, p_data: int, rec_name: str) → bool
Deletes a receiver object
add_heliostat_template(self, p_data: int, helio_name: str) → int
Creates a heliostat template object
Note: CoPylot starts with a default heliostat template at ID = 0, with "Template 1" as the Heliostat's name. If you add a heliostat template object without dropping this default template, generating a layout will fail because the default heliostat geometry distribution (''solarfield.0.template_rule") is "Use single template" but the select heliostat geometry ("solarfield.0.temp_which") is not defined.
drop_heliostat_template(self, p_data: int, helio_name: str) → bool
Deletes heliostat template object
update_geometry(self, p_data: int) → bool
Refresh the solar field, receiver, or ambient condition settings based on current parameter settings
generate_layout(self, p_data: int, nthreads: int = 0) → bool
Create a solar field layout
assign_layout(self, p_data: int, helio_data: list, nthreads: int = 0) → bool
Run layout with specified positions, (optional canting and aimpoints)
get_layout_info(self, p_data: int, get_corners: bool = False, restype: str = "dataframe") → pandas.DataFrame
Get information regarding the heliostat field layout
simulate(self, p_data: int, nthreads: int = 1, update_aimpoints: bool = True) → bool
Calculate heliostat field performance and receiver flux distribution
summary_results(self, p_data: int, save_dict: bool = True) → dict
Returns table of summary results from each simulation
detail_results(self, p_data: int, selhel: list = None, restype: str = "dataframe", get_corners: bool = False) → pandas.DataFrame
Get heliostat field layout detail results
get_fluxmap(self, p_data: int, rec_id: int = 0) → list
Retrieve the receiver fluxmap, optionally specifying the receiver ID to retrieve
clear_land(self, p_data: int, clear_type: str = None) → None
Reset the land boundary polygons, clearing any data
add_land(self, p_data: int, add_type: str, poly_points: list, is_append: bool = True) → bool
Add land inclusion or a land exclusion region within a specified polygon
heliostats_by_region(self, p_data: int, coor_sys: str = 'all', **kwargs) → pandas.DataFrame
Returns heliostats that exists within a region
modify_heliostats(self, p_data: int, helio_dict: dict) → bool
Modify attributes of a subset of heliostats in the current layout
save_from_script(self, p_data: int, sp_fname: str) → bool
Save the current case as a SOLARPILOT .spt file
dump_varmap_tofile(self, p_data: int, fname: str) → bool
Dump the variable structure to a text csv file

Downloaded from http://verification.asmedigitalcollection.asme.org/solarenergyengineering/article-pdf/144/3/030906/6871816/sol_144_3_030906.pdf by guest on 18 July 2024

References

[1] Wagner, M. J., and Wendelin, T., 2018, "SolarPILOT: A Power Tower Solar Field Layout and Characterization Tool," Sol. Energy, **171**, pp. 185–196.

[2] Ho, C. K., 2008, "Software and Codes for Analysis of Concentrating Solar Power Technologies," Sandia National Laboratories, Albuquerque, NM, Report No. SAND2008-8053, Technical Report.

[3] Li, L., Coventry, J., Bader, R., Pye, J., and Lipiński, W., 2016, "Optics of Solar Central Receiver Systems: A Review," Opt. Express, **24**(14), pp. A985–A1007.

[4] Blanco, M. J., Amieva, J. M., and Mancillas, A., 2005, "The Tonatiuh Software Development Project: An Open Source Approach to the Simulation of Solar Concentrating Systems," ASME 2005 International Mechanical Engineering Congress and Exposition. Computers and Information in Engineering, Orlando, FL, Nov. 5–11, pp. 157–164.

[5] Delatorre, J., Baud, G., Bézian, J. J., Blanco, S., Caliot, C., Cornet, J. F., Coustet, C., Dauchet, J., El Hafi, M., Eymet, V., and Fournier, R. 2014, "Monte Carlo Advances and Concentrated Solar Applications," Sol. Energy, **103**, pp. 653–681.

[6] Potter, D. F., Kim, J. -S., Khassapov, A., Pascual, R., Hetherton, L., and Zhang, Z., 2018, "Heliosim: An Integrated Model for the Optimisation and Simulation of Central Receiver CSP Facilities," SolarPACES 2017: International Conference on Concentrating Solar Power and Chemical Energy Systems, Vol. 2033, Santiago, Chile, Sept. 26–29, AIP Publishing LLC, Paper No. 210011.

[7] Izygon, M., Armstrong, P., Nilsson, C., and Vu, N., 2011, "TieSOL—A GPU-Based Suite of Software for Central Receiver Solar Power Plants," SolarPACES 2011: International Conference on Concentrating Solar Power and Chemical Energy Systems, Granada, Spain, Sept. 20–23, pp. 1–8.

[8] Izygon, M., McMurtrie, K., and Vu, N., 2018, "Particle Swarm Optimization of the Layout of a Heliostat Field," SolarPACES 2017: International Conference on Concentrating Solar Power and Chemical Energy Systems, Vol. 2033, Santiago, Chile, Sept. 26–29, AIP Publishing LLC, Paper No. 040018.

[9] Lipps, F., and Vant-Hull, L. L., 1978, "A Cellwise Method for the Optimization of Large Central Receiver Systems," Sol. Energy, **20**(6), pp. 505–516.

[10] Dellin, T. A., and Fish, M. J., 1979, "User's Manual for DELSOL: A Computer Code for Calculating the Optical Performance, Field Layout, and Optimal System Design for Solar Central Receiver Plants," Sandia National Lab. (SNL-CA), Livermore, CA, Report No. SAND79-8215, Technical Report.

[11] Vittitoe, C., and Biggs, F., 1981, "A User's Guide to HELIOS: A Computer Program for Modeling the Optical Behavior of Reflecting Solar Concentrators. Part I. Introduction and Code Input," Sandia National Laboratories, Albuquerque, NM, Report No. SAND81-1180, Technical Report.

[12] Schwarzbözl, P., Pitz-Paal, R., and Schmitz, M., 2009, "Visual HFLCAL—A Software Tool for Layout and Optimisation of Heliostat Fields," SolarPACES 2009: International Conference on Concentrating Solar Power and Chemical Energy Systems, T, Mancini, and R, Pitz-Paal, eds., Berlin, Germany, Sept. 15–18, pp. 1–8.

[13] He, C., Zhao, Y., and Feng, J., 2019, "An Improved Flux Density Distribution Model for a Flat Heliostat (iHFLCAL) Compared With HFLCAL," Energy, **189**, p. 116239.

[14] Wang, Y., Potter, D., Asselineau, C.-A., Corsi, C., Wagner, M., Caliot, C., Piaud, B., Blanco, M., Kim, J.-S., and Pye, J., 2020, "Verification of Optical Modelling of Sunshape and Surface Slope Error for Concentrating Solar Power Systems," Sol. Energy, **195**, pp. 461–474.

[15] 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," SolarPACES 2018: International Conference on Concentrating Solar Power and Chemical Energy Systems, Vol. 2126, Casablanca, Morocco, Oct. 2–5, AIP Publishing LLC, Paper No. 170004.

[16] Wang, Y., Asselineau, C.-A., Coventry, J., and Pye, J., 2016, "Optical Performance of Bladed Receivers for CSP Systems," ASME 2016 10th International Conference on Energy Sustainability Collocated With the ASME 2016 Power Conference and the ASME 2016 14th International Conference on Fuel Cell Science, Engineering and Technology, Vol. 1, Charlotte, NC, June 26–30, American Society of Mechanical Engineers, p. V001T04A026, Paper No. ES2016-59693.

[17] 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), pp. 1–15.

[18] Grigoriev, V., Milidonis, K., Constantinou, M., Corsi, C., Pye, J., and Blanco, M., 2021, "Optimal Sizing of Cylindrical Receivers for Surround Heliostat Fields Using FLUXTRACER," ASME J. Sol. Energy Eng., **143**(6), p. 061007.

[19] Ortega, J. I., Burgaleta, J. I., and Téllez, F. M., 2008, "Central Receiver System Solar Power Plant Using Molten Salt as Heat Transfer Fluid," ASME J. Sol. Energy Eng., **130**(2), p. 024501.

[20] Wendelin, T., 2009, "SolTRACE: A New Optical Modeling Tool for Concentrating Solar Optics," ASME 2003 International Solar Energy Conference, Vol. 36762, Kohala Coast, HI, Mar. 15–18, pp. 253–260, Paper No. ISEC2003-44090.

[21] Python Software Foundation, 2021, "Python Language Reference (Version 3.9.1)," https://docs.python.org/3.9/reference/, Accessed March 4, 2022.

[22] 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)," National Renewable Energy Lab. (NREL), Golden, CO, Report No. NREL/TP–6A20–70414, Technical Report.

[23] McKinney, W., 2010, "Data Structures for Statistical Computing in Python," Proceedings of the 9th Python in Science Conference, Vol. 445, Austin, TX, June 10, pp. 51–56, Paper No. 1.

[24] Jolly, K., 2018, *Hands-On Data Visualization With Bokeh: Interactive Web Plotting for Python Using Bokeh*, 1st ed., Packt Publishing Ltd., Birmingham, UK.

[25] Zolan, A., Hamilton, W., Liaqat, K., and Wagner, M., 2021, "HALOS (Heliostat Aimpoint and Layout Optimization Software)," https://github.com/NREL/HALOS, Accessed March 4, 2022.

[26] Ashley, T., Carrizosa, E., and Fernández-Cara, E., 2017, "Optimisation of Aiming Strategies in Solar Power Tower Plants," Energy, **137**, pp. 285–291.

[27] 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.

[28] 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*, 2nd ed., Vol. 67, Springer Science & Business Media, Berlin, Germany.

[29] Zolan, A., Hamilton, W., Wagner, M., and Liaqat, K., 2021, "Solar Field Layout and Aimpoint Strategy Optimization," National Renewable Energy Lab. (NREL), Golden, CO, Florida State University, Tallahassee, FL, Report No. NREL/TP–5700-80596, Technical Report.

[30] Hart, W. E., Watson, J.-P., and Woodruff, D. L., 2011, "Pyomo: Modeling and Solving Mathematical Programs in Python," Math. Program. Comput., **3**(3), pp. 219–260.

Downloaded from http://verification.asmedigitalcollection.asme.org/solarenergyengineering/article-pdf/144/3/030906/6871816/sol_144_3_030906.pdf by guest on 18 July 2024