Enhancing EnergyPlus Simulation Capabilities through Extended API Integration

Akash Kota Raju
New York University, New York
ak9381@nyu.edu

Abstract. This paper discusses the development and integration of advanced Application Programming Interfaces (APIs) into EnergyPlus, a widely used building energy simulation software. The APIs, including functional, runtime, and data exchange categories, enable enhanced inter- action with EnergyPlus simulations through external scripting and real-time data exchange. These enhancements broaden EnergyPlus' utility by facilitating complex control strategies, dynamic data integration, and interoperability with external tools and frameworks, thus advancing its capabilities in building energy performance analysis and simulation.

Keywords: API integration, energyplus, dynamic data integration, simulation

1 Introduction

Building energy simulation has evolved significantly over the past decades, yet many core simulation engines remain rigid and difficult to integrate with modern, dynamic tools. EnergyPlus, a widely used simulation engine, has traditionally operated as a standalone system with limited support for real-time control or integration with external applications. Recent efforts to expose its internal functionalities via an Application Programming Interface (API) and Python scripting have opened new avenues for enhanced flexibility and interoperability.

This paper presents an extended API integration for EnergyPlus that trans- forms it into an accessible, interactive simulation library. The major contributions of this work are as follows:

Dynamic Integration: I introduce a comprehensive API that enables real- time data exchange and dynamic control of EnergyPlus simulations, facilitating seamless integration with graphical interfaces and third-party tools.

Enhanced Usability: In response to initial usability challenges, the API now incorporates robust error handling, clear debugging tools, and higher- level abstractions to lower the barrier for untrained users.

Multi-threading Considerations: I identify potential multi-threading is- sues with runtime callbacks and outline future strategies to ensure reliability and stability in concurrent execution environments.

P. Ranjan et al. (eds.), *Proceedings of the International Conference on Recent Advances in Artificial Intelligence for Sustainable Development (RAISD 2025)*, Advances in Intelligent Systems Research 196,
https://doi.org/10.2991/978-94-6463-787-8_53

Empirical Validation Roadmap: Although our design shows promise, I acknowledge that the claims regarding integration and performance must be validated. I propose a roadmap for future work that includes real-world case studies, performance benchmarks, and systematic user feedback to substantiate these benefits empirically. These contributions collectively aim to elevate EnergyPlus from a traditional, monolithic simulation engine to a modern, flexible tool capable of addressing emerging challenges in building energy simulation. The remainder of the pa- per is organized as follows: Section 2 details the API design and integration methodology, Section 3 discusses the experimental evaluation and future work, and Section 4 concludes with a discussion of the impact and future directions of this research.

2 Background

The history of simulation of building physics, including peak load prediction and energy simulation, has been around for decades, [1] with countless variations in accuracy and flexibility. In the 1990s, the United States Department of Energy created a state-of-the-art flagship simulation engine, which was ultimately named EnergyPlus and released around the turn of the millennium [2]. Since that time, EnergyPlus has evolved in many ways. Some of the more prominent changes are listed in Table 1.

Table 1. Major related events in EnergyPlus development

Year	Event
1999 or 2000	Initial release of EnergyPlus
2007	User defined controls (EMS)
2011	External interface for co-simulation
2014	Conversion from Fortran to C++

Recent releases of the tool have achieved around 40,000 direct downloads, plus many thousand more uses as it is packaged with software development kits like OpenStudio and with other commercial graphical interfaces. Each of the events listed in Table 1 were critical in improving the usability and accuracy of the tool, thus playing a role in establishing the current state of the program.

2.1 User Defined Controls

For nearly EnergyPlus' entire life, it has existed as an opaque widget that accepts input data in terms of an input file, runs a series of calculations, integrates over time, and produces a set of output files. A user could only alter the calculation methodology to evaluate custom

controls or capture new physics by modifying the Fortran source code and rebuilding the tool itself. In 2007, a new capability was added to EnergyPlus in the form of a user-defined runtime language [3]. This system called the Energy Management System (EMS) in EnergyPlus, allowed users to read simulation data and actuate simulation controls according to their custom logic at dynamic calling points during a simulation. This was ultimately extended to allow users to write custom control logic and define custom component models. This was a major advancement that allowed users to fill in the gaps of the built-in EnergyPlus controls and component models and perform simulations that involved niche, obscure, or cutting-edge technologies and controls that are not pre-compiled with the EnergyPlus release. Developers have exercised these [4].

At the time of implementation, EnergyPlus was still in the adolescent phase of development and was viewed primarily as a research-oriented tool. The addition of a user-defined language supported the research interest in simulating novel control strategies and technologies and ensured that companies could simulate their as-built technologies without having to contribute control logic back into the original Fortran codebase. This spurred interest in the tool from new industry partners and was critical for moving EnergyPlus toward achieving widespread adoption. As capabilities evolved, and with the ability to actuate thousands of controls and settings in a running simulation, the system became more of a generic plugin system. However, with all that power, the EMS code was still restricted in a few primary ways:

- The “EnergyPlus Runtime Language” is a custom language with no external support beyond the EnergyPlus ecosystem.
- It includes limited data types and control flow capabilities.
- User-defined functionality is embedded directly into the IDF input file syn- tax, complicating code management.
- Debugging simulations with user-defined content involves sifting through large stack evaluation files, often in gigabytes, with no simple way to limit outputs to specific times or variables.

Even with these limitations, the Energy Management System was an impres- sive feature that helped EnergyPlus expand further into research and develop- ment and commercial opportunities.

2.2 External Interface

The external interface feature of EnergyPlus was the first API to allow other tools to communicate with EnergyPlus. The specifics of this system are available in literature [5], but it can be summarized as a socket-based approach, allowing tools to communicate with a running simulation (VERIFY THIS). This sys- tem was used by the Building Controls Virtual Test Bed [6] and was the first attempt at exposing the core of EnergyPlus to external tools. External inter- faces were utilized in research applications; however, wiring up new capabilities required expertise in multiple domains: computer science and software develop- ment to set up these tools and programs, as well as mechanical and building science expertise to make engineering judgments during model development and results evaluation. This limited the number of users who could make use of this development.

2.3 Conversion from Fortran to C++

At the birth of EnergyPlus, modern Fortran standards provided the most state- of-the-art programming capabilities for a numerical simulation tool [7]. Capa- bilities such as dynamically allocating user-defined derived types allowed the program to be modular. The program also took advantage of the memory man- agement and optimization capabilities of the Fortran language and tools. Fortran was the best language for creating the simulation tool and persisted for well over a decade. At that time, however, the C++ language gained worldwide popularity while Fortran's popularity declined, and the interval between official standards lengthened, thus leaving modern enhancements behind. To date, the ability to do otherwise commonplace modern coding practices in Fortran justifies a journal article [8]. While the C++ language lacks some Fortran features, such as built- in multi-dimensional array support, the massive developer base and community surrounding C++ allow access to many modern libraries and scientific packages. The conversion from Fortran to C++ was thus performed for two main reasons:

- improving the access to developers, including gaining commercial support, was more achievable with the codebase in C++ and
- moving to C++ lowers the bar to provide many new capabilities, such as allowing code reuse through a C/C++ API. Although Fortran does technically have objects and classes and can expose them through a programming interface, it is widely accepted that the class structure in C++ is set up to be much more amenable to this situation than Fortran, and the C++ community has substantially more resources for these patterns

2.4 EnergyPlus API History

Software programs and libraries utilize variables and functions in order to perform their required duties. While most functionality from standalone programs is kept internal, programs can evolve into more accessible libraries. As functions, structures, and variables are exposed through an interface, they can be reused by clients. Desktop applications, mobile apps, and web applications commonly expose an Application Programming Interface (API). The design of APIs has evolved, but there are obvious benefits, including:

- individual libraries and programs maintain formal boundaries, with isolated responsibilities and
- Individual programs and applications can be tied together to create larger workflows, reusing each program's features.

External tools have been developed that build on EnergyPlus and provide various capabilities for interacting with the simulation. [9,10] These external ap- plications' primary applications have not been related to interacting with a run- ning simulation because that capability was not exposed via EnergyPlus. These applications were instead related to developing input values for a simulation or a group of simulations and then managing individual simulation processes, such as running many simulations concurrently.

EnergyPlus releases did not officially expose simulation functionality through an API layer while the codebase was in Fortran. Once the code was translated to C++, the build and packaging process was modified to include an intermediate step of creating a shared library for EnergyPlus. This shared library included minimal functionality for executing a simulation and allowed a calling program to register functions to be called back periodically during a simulation. The EnergyPlus program that users execute is a consumer of this shared library, creating a thin application layer on top of the simulation function call. While there were examples of graphical interfaces that would call EnergyPlus as a library, nearly every tool simply executed EnergyPlus as a standalone program. This status would remain leading up to this current work, when the current work, a new formal API for EnergyPlus, would enable novel workflows.

3 The EnergyPlus API

Developing an API layer on a scientific computing library is not unprecedented [10-12]. In the current work, the capabilities of EnergyPlus are extended through the implementation of two distinct interface approaches. The first interface allows users to call into EnergyPlus from an outside calling script or program, read sensors and write actuator data, make use of exposed calculation func- tionality, and run simulations. This capability is referred to as the “EnergyPlus API”. This API allows access to EnergyPlus’s internal pieces, both at runtime and on-demand, when a simulation is not running. The second interface adds a Python interpreter to the EnergyPlus engine’s internal capabilities, allowing users to execute user-defined Python code from typical EnergyPlus workflows. This second case is similar to the traditional EMS code, except instead of ex- pressing the custom functionality in the input file using the EnergyPlus runtime language, the user-defined functionality lives outside the EnergyPlus boundary in a Python script. This second workflow is called Python EMS. In both cases, there is a key benefit: boundary conditions and input data do not need to be fully declared and known ahead of the simulation. In traditional EMS and non-API workflows, the entire set of input data, along with the logic, had to be declared before the simulation could run. With the new API and Python EMS capabilities, the boundary conditions can be dynamically evaluated from any outside source. In both cases, the Python code can call out to other tools and libraries to achieve the goals and perform calculations dynamically. The EnergyPlus API allows access to the simulation, so while the Python EMS code is executing, it can leverage the API to communicate back to the running simulation that started executing the code. The remainder of this section describes the scope, design, capabilities, and limitations of the EnergyPlus API itself, with further information on the Python EMS workflow in the following section.

3.1 Scope and Purpose

Building simulation programs, like other tools, should evolve to maximize useful- ness in the hands of the users. This evolution can be justified for various reasons,

including supporting changes in state-of-the-art technology, allowing novel con- trol algorithms, and providing improved interoperability and user experience. In modern computer applications, it is increasingly common to allow access to the internal functions and classes through an API layer and to allow special operations to occur through the use of plugins. The purpose of this current work is to expose EnergyPlus internal functionality. The functionality is exposed through a single API divided into categories purely for organization, which are summarized in Table 2.

Table 2. Summary of EnergyPlus API Categories

Category	Purpose
State	Access, an instance of EnergyPlus, used in most API operations
Functionality	Access calculations and functionality without requiring a running simulation
Runtime	Hook into a running simulation at specific calling points, as well as initiating the simulation itself
Data Exchange	Access internal data for reading (sensors) and writing (actuators)

The state API category provides access to an "instance" of EnergyPlus. This instance is needed for nearly all API functionality, as EnergyPlus may need to instantiate data behind the scenes even during simple API calls. This API category provides functions to create a new state, clear a state, and destroy a state. From a user perspective, creating a state through the API is similar to creating an EnergyPlus instance, which is what a client could expect to do with other libraries.

The functional API category provides access to static calculation methods and structures in EnergyPlus. These represent aspects of EnergyPlus that can be accessed without requiring the client to initiate a simulation. Internally, En- ergyPlus uses functions and classes to organize and calculate the physics and bookkeeping operations in a simulation. The operations include geometric calcu- lations, thermodynamic property evaluation, solar calculations, and component models such as coils and boilers, among many others. Some of these operations require an actively running simulation to calculate transient properties and set up time series data. Some, however, can be performed by providing a reason- able amount of initialization and boundary condition data. The functional API category is intended to capture these static operations and expose them to En- ergyPlus clients. For fluid property and psychrometric calculations, a client may utilize the API to calculate enthalpy, moisture properties, specific heats, and

more, using relevant input values for the functions. For geometric zone calculations, which are not yet completed, a user will need to provide vertex values to layout the geometry of a surface or zone. Then, calculations can be per- formed on the surface or zone. The API currently includes refrigerant property calculations, fluid property calculations, and psychrometric calculations. Future enhancements to the functional API will include the aforementioned geometric calculations, as well as the ability to "instantiate" a component model such as a pump, boiler, or coil, and evaluate outlet conditions given required inlet, boundary, and potentially initial or historical conditions.

In contrast to the static functionality provided by the functional API, the runtime API provides the means to attach to a running simulation. Consider the traditional way to run EnergyPlus, where a simulation is executed with all the required input data predefined and no way to interact with the simulation boundary data. With the runtime API, a client creates functions and registers those functions with an EnergyPlus instance, and initiates a simulation. The functions are registered for specific calling points so that the client functions are called back at particular points in the simulation. The available calling points are the same as with the built-in EMS system, representing a wide range of frequencies, from once per run environment to the depths of the iteration structure. After registering the callback function, the client starts the simulation, which calls back to the client function as it runs. The client function can do anything you can expect to do with a programming language, including:

- Access files and directories on the file system
- Access databases, both local and over a network
- Access web servers that can provide dynamic data
- Evaluate machine learning algorithms

Regardless of what the client function does externally, the primary use case for a user-defined function is to interact with the running simulation instance, the details of which are described in the following data exchange API discussion.

The primary goal of the API effort is to provide the means to read and write data inside a running EnergyPlus simulation. The state API category provides functionality for creating the instance of EnergyPlus, the functional API exposes some internal functions to the client, and the runtime API allows for connecting an external function to be called back by EnergyPlus at specific points. The final category of API, the data exchange API, provides the final required piece: the means to exchange data with the running simulation. The data exchange process includes reading the current value of any possible output variable from the spe- cific running simulation and overriding control parameters using actuators. To exchange this data, clients call into EnergyPlus from a callback function previ- ously registered with EnergyPlus. The client calls functions that request handles to variables, meters, and actuators by name. These handles are then used in all subsequent function calls into the running simulation when reading or writing values. The key aspect of this system is that the client can then combine an incredible amount of data to make decisions.

When the callback function returns to EnergyPlus, the simulation engine will continue running using any updated actuator values. This is the same pattern as the current EMS system, but it allows clients to go well beyond the boundary of the input-file-based language limitations.

3.2 Use Cases

The different categories of API have been described: state, functional, runtime, and data exchange. The two interface "modes" have been described:
The mode where EnergyPlus is executed by a client as a standalone tool that is then directed to execute user-defined Python code using an embedded interpreter library
mode where EnergyPlus is called as a function from external code, either C or Python.
The first mode, Python EMS, is expected to be familiar to users of traditional EnergyPlus workflows. In the second mode, EnergyPlus is expected to become a single cog in a broader machine or workflow. The client creates a calling script that links with the EnergyPlus library to make calls into the simulation engine using any or all of the API categories. The calls do not need to start a simulation; they might only access the functional API to perform calculations. In many cases, however, clients will go beyond that and access the runtime API to register a user-defined function to be called back at specific points. Then, they will utilize the data exchange API to read and write simulation state data. Figure 3.2 provides a visualization of the two different modes.

Fig. 1. EnergyPlus Workflows

The EnergyPlus as a library approach opens new potential calling structures, although they generally follow the same pattern:
A client registers functions as callbacks using the runtime API and kicks off a simulation run with the runtime API. When those registered func tions are called back, they can then call back into the library to read and write data using the data exchange API and perform calculations with the functional API.
Regardless of whether EnergyPlus is called using the traditional approach or calling the tool a library, the possibilities opened up by this API effort are extensive. The ability to perform data exchange between a running simulation and an outside world just by writing a few lines of code rapidly unlocks the ability to do many things, including:

- connecting with multiple tools at once, potentially keeping tools in lockstep to perform simulation,
- interfacing with hardware or virtualized hardware,
- interfacing with specialized control languages and frameworks,
- providing live data visualization while a simulation is running.

3.3 C API Design

The details of actually making internal function calls available to clients outside of the compilation unit are not fully described here, as calling into application programming interfaces is standard practice (CITE). Creating a useful API requires careful planning of the functions and classes. Documenting and conveying the design to clients is equally important as the purpose is to make an interface accessible for users. In this section, the core C API is described.
EnergyPlus is written in the C++ programming language [11]. Exposing a C++ API is possible, but exposing a C API results in a more accessible API for more programming languages, primarily Python. With the C API layer in place, a Python layer can be easily built on top of it to access the API. With the C and Python APIs in place, clients can write C++, C, or Python code that can access EnergyPlus functionality for performing runtime control, data exchange, and functional calculation calls. In the case of C++ or C, the client code must be compiled and dynamically linked to the EnergyPlus library. In the case of Python, the EnergyPlus Python API layer actually handles the linking process when constructed. (There could be a diagram from this paragraph)
Some design guidelines are followed when laying out the API. The actual function names described here should not be considered official documentation, as the API is subject to change in public versions of EnergyPlus. Users should access EnergyPlus documentation, examples, and header files to get official API documentation. However, the important guidelines followed are described here. In standard practice, the structure of EnergyPlus code is comprised of vari- ables, classes, and functions. When classes and methods on those classes are to be exposed through the C API,, each class is exposed with the following process:

- A C function is created, which captures the construction of the class instance, returning a pointer to that instantiated class. The function should accept all arguments necessary to create an example of the underlying C++ class.
- A C function is created, which captures the destruction of the class instance, accepting only a pointer to the instance to be destructed.
- A C function for each relevant member variable on the class, with one method for writing and one method for reading where appropriate.
- A C function for each relevant method on the class, with arguments that can be translated to the arguments in the C++ function.

An example of this is the glycol routines. Consider the glycol class in Ener- gyPlus and how it is exposed via API functions.

```
// C++ Glycol Class
struct GlycolAPI {
FloatType specific Heat( Real64 temperature ); FloatType density ( Real64 temperature ); FloatType conductivity ( Real64 temperature ); FloatType viscosity(Real64 temperature);
};
// C API
typedef void * Glycol;
Glycol glycolNew(const char* glycolName); void glycolDelete(Glycol);
FloatType glycolSpecific Heat ( Glycol , Real64 temperature ); FloatType glycolDensity (Glycol , Real64 temperature ); FloatType glycolConductivity ( Glycol , Real64 temperature ); FloatType glycolViscosity (Glycol , Real64 temperature);
```

In this interface, from the core C++ code to the C API, appropriate C types are used in place of standard C++ library types, such as character pointers, in place of standard strings. In addition, to ensure floating point consistency, the internal floating point type used when compiling EnergyPlus is exposed on the C header so that the API, and code built against the API, can use the same floating point type declaration. Static functions are exposed similarly, with thin wrapper functions that only communicate in C data types.

The runtime and data exchange APIs are developed similarly, creating thin wrapper functions in C and mapping data in and out of the interface. There is one notable caveat about data exchange variables: sensor data must be re- quested prior to initiating a simulation. During an EnergyPlus simulation, many thousands of output variables can be instantiated and potentially reported. To avoid a huge memory penalty at runtime, only the requested variables are actually instantiated and available. This is straightforward in the traditional use case because the input file is available for interpretation at the beginning of the simulation, and all required output variables are already known. In the library

use case, a sensor variable handle may not be retrieved until the simulation is well underway. To avoid having to track the full potential set of variables, the client should request the variable by name prior to making a call to execute the simulation. A minimal example is provided here, which demonstrates the usage of all three API categories, including the use of requesting an output variable prior to executing the simulation.

```
#include <EnergyPlus/api/func.h>
// Glycol props
#include <EnergyPlus/api/datatransfer.h>
// exchange
#include <EnergyPlus/api/state.h>
// EnergyPlus state
#include <EnergyPlus/api/runtime.h>
// E+ and callbacks

void afterZone Time Step Handler ( Energy Plus State s) { int oa Dew Point Actuator =
getActuatorHandle(
s, "Environment", "Weather␣Data",
"Outdoor␣Dew ␣Point"
);
setActuatorValue ( oa Dew Point Actuator , -25); int oa Temp Sensor = getVariable
Handle(
s, "SITE␣OUTDOOR␣AIR␣DRYBULB␣TEMPERATURE", "ENVIRONMENT"
);
float oaTemp = getVariableValue(oaTempSensor); Glycol glycol = NULL;
glycol = glycolNew(s, "Water");
float specificHeat = glycolSpecificHeat (s, glycol , 23.3);
}
int main ( int argc , const char * argv []) { Energy Plus s = state New
(); callback End Of Zone Time Step AfterZone Reporting ( s, afterZone Time Step
Handler
);
requestVariable(
s, "SITE␣OUTDOOR␣AIR␣DRYBULB␣TEMPERATURE", "ENVIRONMENT"
);
energyplus(s, argc, argv); return 0;
}
```

3.4 Python API Design

The addition of a C API brings EnergyPlus out of the dark from being a closed- off simulation engine to becoming an interactive library. This C API alone opens the door for many new EnergyPlus applications. However, the accessibility and usability of this enhancement can be expanded into a massive set of new possibilities by making the API accessible through the Python programming language and ecosystem [12]. Adding a Python API enables clients to generate script-based workflows utilizing EnergyPlus as well as a wealth of packages and libraries available in Python repositories online.

The EnergyPlus Python API layer is written in pure Python and communicates with EnergyPlus through the now-established C API layer. These capabilities are built into the Python standard library, and as such, the lower-level details of accessing the C API are concealed from users. For users consuming the Python API layer, the experience is akin to using any other pure Python package.

Python is an object-oriented language, and Python users are accustomed to utilizing object-oriented libraries. As such, the EnergyPlus Python API layer is also object-oriented. Each API "type" described above in the C API section (functional, data exchange, runtime) has an accompanying Python class. Within each class, members and methods provide access to different elements of the API. There is a single, highest-level class that provides access to all API types and subclasses. This class ("EnergyPlusAPI") is defined within a Python package called "pyenergyplus", which is packaged with EnergyPlus and is effectively a scope for all Python API calls. The API layer is fully documented, but a pseu-docode representation can be described here. The highest level class contains member variables that capture the three API types. These, in turn, also include methods and member variables, which are equivalent to the C API methods. For example, the functional API contains a method called glycol, which requires a valid EnergyPlus state instance plus the name of the glycol to construct as function arguments.

Once again, the actual names of the methods and classes provided here are subject to change and should not be considered documentation of the official API; they should only be a description of how the API was developed. With this in mind, consider the following example of using the Python API layer to access EnergyPlus:

```
from pyenergyplus.api import EnergyPlusAPI def time_step_handler(s):
outdoor_temp_sensor =
api.exchange.get_variable_handle(
s, "SITE_OUTDOOR_AIR_DRYBULB_TEMPERATURE", "ENVIRONMENT"
)
dew_point_actuator = api.exchange.get_actuator_handle(
s, "Environment", "Weather_Data", "Outdoor_Dew_Point"
```

```
)
api.exchange.set_actuator_value(s, dew_point_actuator, -25
)
oa_temp = api.exchange.get_variable_value(s, outdoor_temp_sensor
)
api = EnergyPlusAPI()
s = api.state_manager.new_state() api.runtime.callback_end_zone_
timestep_after_zone_reporting(
s, time_step_handler
)
api.exchange.request_variable(
s, "SITE␣OUTDOOR␣AIR␣DRYBULB␣TEMPERATURE", "ENVIRONMENT"
)
api.runtime.run_energyplus(s, sys.argv[1:]) glycol = api.functional.
glycol(s, "water") cp = glycol.specific_heat(s, 23.5)
```

With less than 30 lines of (verbose) Python code, a client can access the runtime portion of the API to register a callback function, request and access variable data, set an actuator value, and also access the functional API to cal- culate the specific heat of a glycol. This is a minimal—and naive—example of using the API, but it demonstrates its accessibility.

4 Python Plugins

The previous sections of this document have focused on the development of APIs to be called from outside tools into EnergyPlus, with a focus on controlling, accessing, and modifying a simulation. A simultaneous and related effort was made to allow EnergyPlus to call out to execute Python scripts at runtime. This methodology is termed Python Plugins for EnergyPlus. To users familiar with EnergyPlus and the traditional EMS system, this is essentially an alternative EMS system that uses external Python scripts instead of the IDF-based user- defined Erl language scripts.

In the Python Plugin system, EnergyPlus is driving the process; it is not being called as a library from a separate script or tool. However, these user-defined Python plugins can be specified to be called from the same calling points as those made available for library calls. The difference is simply that EnergyPlus is calling out to the Python plugin rather than the Python code calling into EnergyPlus. As such, EnergyPlus is the driving program and must interpret user-defined Python code. EnergyPlus is linked to the Python interpreter library to handle this functionality. Internally, EnergyPlus calls out to the user-defined plugin at the desired calling points, and the plugin can then do anything a user can do in Python. Most use cases of the Python Plugin system will then leverage

the previous API efforts and make requests back into the running EnergyPlus API to get sensor data and write actuator settings. This is similar to how current EMS-based functionality can read sensors and write actuator data.

In order to create a standardized method for calling out from any program to a plugin, it is commonplace to create a plugin base class, which clients must override in creating their plugin-derived classes. This provides a way to define the requirements of the plugin class clearly and allows for evolving plugin ca- pabilities over time by creating alternative base classes. As with the C API access methods described previously, the following listing and examples are not considered official but rather representative class names and member names. A representative snippet of the plugin base class is shown here:

```
from pyenergyplus.api import EnergyPlusAPI class EnergyPlusPlugin(object):
def on_begin_new_environment(self, state):

# called at the begin_new_environment point def
on_begin_timestep_before_predictor (self, state):
# called at the begin_timestep_
before_predictor point
def on_inside_hvac_system_iteration_loop (self, state):
# called at the inside_hvac_system_
iteration point
... etc ...
```

A user will create their class that derives from the EnergyPlus Plug-in base class and overrides one, multiple, or all calling point methods. The user-defined class is instantiated once at the beginning of the simulation, and the instance is held in memory throughout the lifetime of the simulation. This is an important feature because the user can carry data on the class instance and make it available to multiple calling point functions. This can be used to persist tracking information, historical data, or any other reason. Throughout the simulation, the user-defined class functions will be called at the appropriate times. Inside the overridden function, the plugin can then call on the EnergyPlus API to per- form calculations using the functional API and exchange data, as well as reading sensors and writing actuators using the data exchange API. Once complete, the function must return an integer. If the function returns a zero, EnergyPlus will continue running based on that successful signal. If a nonzero value is returned, it is a signal to EnergyPlus that the simulation should abort based on an error condition. The plugin function can issue warnings and errors through API methods if desired prior to returning the nonzero value so that the error file is fully populated with information.

The full process to utilize the Python Plugin system consists of the following pieces:

– Create a derived plugin as described above, which overrides at least one of the calling point methods

- Utilize the Python API to make calls in and out of the running EnergyPlus simulation as needed
- Declare the plugin in the simulation input file, specifying the Python module name the class name and adding any search paths required to find the Python script

Graphics:

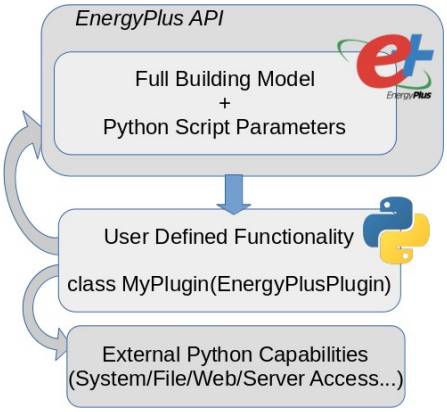

Fig. 2. Plugin Workflow Details

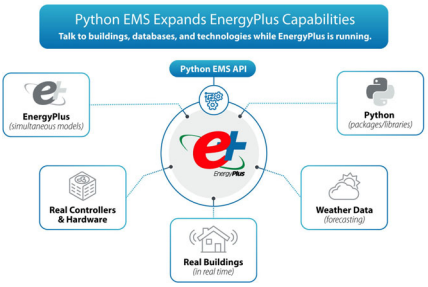

Fig. 3. Python EMS Capabilities

5 Conclusions

This paper marks an advancement in transforming EnergyPlus from a rigid simulation engine into an accessible, interactive library. By exposing its internal functionalities through a comprehensive API and integrating Python scripting, EnergyPlus now offers a flexible platform that enables seamless integration with graphical interfaces and other external tools. This evolution not only simplifies the development of custom control systems and interfaces but also accelerates the evaluation of novel technologies and complex control strategies.

The enhanced API architecture allows developers to leverage established calculation methods and perform dynamic data exchange during simulations, thereby broadening the scope of building performance analysis. Bundling a dedicated Python interpreter and standard library with EnergyPlus addresses traditional dependency concerns by ensuring that users do not need to manage separate Python installations. While challenges remain—particularly regarding the streamlined installation of third-party packages with native binaries—this work lays a robust foundation for future refinements.

Looking ahead, there is considerable potential for further development. Future work will focus on extending the functional API to incorporate additional calculation modules, refining the runtime callback mechanisms, and enhancing interoperability with emerging simulation tools and hardware systems. These advancements are expected to overcome current limitations and empower the EnergyPlus community to tackle increasingly complex and dynamic simulation tasks.

Overall, the contributions presented in this work significantly lower the barriers to innovation, setting the stage for a new era in building energy simulation where flexibility, interoperability, and rapid technological adaptation are at the forefront.

Disclosure of Interest. The authors declare that there are no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

1. Mao, C., Baltazar, J.-C., Haberl, J. S.: Literature review of building peak cooling load methods in the United States. Science and Technology for the Built Environment **24**(3), 228–237 (2018)
2. Crawley, D., Lawrie, L., Winkelmann, F., Buhl, W., Huang, Y., Pedersen, C., Strand, R., Liesen, R., Fisher, D., Witte, M., Glazer, J.: EnergyPlus: Creating a new-generation building energy simulation program. Energy and Buildings 33, 319–331 (2001)
3. Ellis, P., Torcellini, P., Crawley, D.: Simulation of energy management systems in EnergyPlus. Proc. of Conference, pp. 1346–1353 (2007)
4. Dutton, S., Zhang, H., Zhai, Y., Arens, E., Smires, Y. B., Brunswick, S., Konis, K., Haves, P.: Application of a stochastic window use model in EnergyPlus. Proceedings of SimBuild **5**(1), 63–70 (2012) [Online]. Available: http://ibpsa-usa.org/index.php/ibpusa/article/view/416

5. Nouidui, T., Wetter, M., Zuo, W.: Functional mock-up unit for co-simulation import in EnergyPlus. Journal of Building Performance Simulation 7 (2014)
6. Real-time Building Energy Simulation using EnergyPlus and the Building Controls Virtual Test Bed. Sydney, Australia (2011)
7. Adams, J. C., Smith, B. T., Martin, J. T., Brainerd, W. S., Wagener, J. L.: FORTRAN 95 Handbook. 1st edn. MIT Press, Cambridge, MA, USA (1997)
8. Özturan, C., Morris, K.: Emulating multiple inheritance in Fortran 2003/2008. Scientific Programming 2015, 126069 (2015) [Online]. Available: https://doi.org/10.1155/2015/126069
9. Campos, G., Ramos, G., Stauffer, Y., Dasen, S., Fernandez Bandera, C.: EPlus-Launcher: An API to perform complex EnergyPlus simulations in MATLAB® and C#. Sustainability 12, 672 (2020)
10. Mohanan, A. V., Bonamy, C., Augier, P.: FluidFFT: Common API (C++ and Python) for fast Fourier transform HPC libraries. (2018)
11. Stroustrup, B.: The C++ Programming Language. 2nd edn., reprinted with corrections, Addison-Wesley, Reading, Mass. (1995) [Online]. Available: https://search.library.wisc.edu/catalog/999786409402121
12. Python Software Foundation: Python Language Reference. [Online]. Available: http://python.org·