GenEPJ: A Flexible Python-based EnergyPlus templating library

Scott Bucking, Milad Rostami
Department of Civil and Environmental Engineering, Carleton University, Ottawa, Canada

Abstract

This paper presents an open-source Python library, called *genEPJ* for automated measure implementation using IDF, epJSON, eppy, OpenStudio and Modelkit templates. *genEPJ* is preconfigured with over 100 functions to modify EnergyPlus input files. The approach is structured for the creation of reference, proposed and optimal designs for studying a particular building configuration or for parametric/optimization studies. *genEPJ* enables practioners to accurately and rapidly deploy best modelling practices and upgrades using a variety of templating approaches to any EnergyPlus model. The ultimate goal of this library is to allow practioners to focus on the quality assurance of their models and not on the modelling process while enabling new and novel workflows. A newly added feature allows for the study of resilience by restricting run periods around outage events allowing the simulation of a building's response to a temporary loss of utilities (using backup generators). A multi-objective optimization approach is supported which includes energy, cost, emissions and resiliency indicators. The library is designed to be easily run on external servers or be integrated into existing tools in the Rhino, Revit and OpenStudio ecosystems. A present limitation of this tool is that it lacks a user-friendly front-end for implementing measures. Future development is aimed at adding Rhino Honeybee measures to the pre-existing five template languages for additional flexibility.

Introduction

This paper introduces a Python library called *genEPJ* (generate EnergyPlus JSON) which modifies EnergyPlus inputs files (using IDF and epJSON formats). *genEPJ* automates the rapid deployment of specialized energy models and solves a reproducibility issue commonly found when creating EnergyPlus models to meet ever tightening code requirements. This library is backwards compatible to EnergyPlus version 8 and Python version 2.7 and is distributed with five templating methodologies that can coordinate over 1000 templates.

genEPJ combines multiple templating approaches: (i) IDF manipulation using string templates, (ii) direct JSON manipulations (no templates required), (iii) *eppy* functions, (iv) *OpenStudio* measures/workflows, and (v) *Modelkit* templates (compatible with EnergyPlus versions 8 and 9). The advantage of this approach is that users can benefit from decades of energy modelling expertise by adopting and extending existing templates rather than recreating their own. The following paragraghs briefly review current practices and methods in using third party EnergyPlus templates. The presented templating approaches offer features not presently found in EnergyPlus' base EPmacro capabilities which are limited to direct substitutions of block IDF data.

genEPJ's default templating tool uses IDF and JSON formats. Templates are provided for hundreds of energy saving measures including (but not limited to): renewable energy configurations, HVACTemplates, envelope upgrades and EMS control overrides. *genEPJ* supports several new modelling features such as resiliency simulations which helps BPS practitioners study how their designs respond to outage events (Bucking et al., 2022). The structure of *genEPJ* templates are described in the methods section of this paper. We provide a brief overview of the three third-party templating options below.

eppy is a Python library for making EnergyPlus manipulations using input data dictionaries (`Energy+.IDD`) (`https://pythonhosted.org/eppy/`). *genEPJ* supports the `eppy.modeleditor.IDF` class directly by importing the `eppy` library. This library is used by several other templating libraries such as *geomeppy* (`https://geomeppy.readthedocs.io`) and *besos* (`https://besos.readthedocs.io/`).

OpenStudio applies Ruby scripts and XML/XSLT templates to manipulate EnergyPlus IDF/JSON/OSM file formats (`https://openstudio.net/`). This tool is presently maintained by NREL and supports over 260 measures (via the Building Component Library). *OpenStudio* allows for the implementation of both measures and workflows. Measures are targeted changes to an EnergyPlus input file which can iterate over zones or modify/add components. Workflows are packages of measures which are sequentially applied to a model. Integration into *genEPJ* is achieved using a wrapper

function which calls the '`openstudio`' commandline tool.

Modelkit uses Ruby templates to manipulate EnergyPlus input files (`https://bigladdersoftware.com/projects/modelkit`). Originally this tool was developed to support several US Army Corps of Engineers low-energy military bases. *Modelkit* provides detailed mechanical templates using EnergyPlus HVAC objects (which native *genEPJ* lacks). Integration into *genEPJ* is achieved using a wrapper function which calls the '`modelkit`' commandline tool.

Incorporating several templating approaches into *genEPJ* has several advantages: (i) it allows for user preferences in modification and deployment formats, (ii) avoids vender lock-in to particular modelling environments, (iii) avoids duplication of efforts in template development, (iv) leverages pre-existing expertise in implementing EnergyPlus upgrade measures, (v) allows users to select and combine the best features from each template approach, and (vi) enables the extension of existing templates. Figure 1 summarizes advantages and disadvantages of the integrated 3rd party libraries. By selecting the best approaches from each offering, users can maximize their modelling proficiency and productivity.

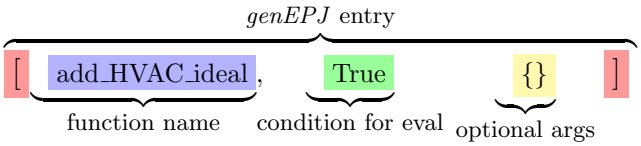
Methods

The following paragraphs describe *genEPJ*'s design principles which can be summarized as: (i) reproducible, (ii) structured, (iii) deployable, (iv) functional, (v) succinct, (vi) auditable, (vii) interpretable, (viii) verbose, (ix) standalone, and (x) flexible.

This library enables reproducible workflows. Starting from an identical base file name, all changes to the file are automated. This implies that every sequence of changes are logged and tracked throughout the entire modelling process. Manual interventions from an end-user are highly discouraged in *genEPJ* as they cannot be tracked or reproduced.

genEPJ is internally structured to create preparatory, reference, proposed and optimized designs. BPS practitioners commonly using energy models to estimate the incremental improvements over a baseline design. This baseline design is often called a reference or code minimal building is the starting point of an energy study. Relative improvements are a requirement to determine pathway analyses and life-cycle cost indicators. The preparatory file implements modelling best practices that are later used by the reference, proposed and optimized designs. The optimization file reads in, and applies, parametric values from a JSON file ('`opti_inputs.json`'). Both the reference and proposed files are user-defined based on codes or as-built/finalized designs. The optimized design is typically algorithmically generated. *genEPJ* uses shortened suffix names for these fundamental tasks: prep, ref, prop and opti.

genEPJ has no dependencies (beyond EnergyPlus and 3rd-party templates) meaning that a project is easily deployable to external servers via copying the project file directory. Since most users will want to modify and extend the provided templates, we recommend that *genEPJ* be deployed to servers with the library located in the deployed directory. This simplifies the installation process, and management of templates. Although, *genEPJ* can be installed using tools such as *pip* and *conda*, direct installation adds significant flexibility. If an end-user wishes to have multiple projects use the same template, then they can choose to have a symbolic link to *genEPJ* within each project directory. It is expected that *genEPJ* will commonly be executed via command line once deployed to a server.



(1)

This library prefers a functional programming approach to text file manipulation. As shown in the above Figure, the user manipulates EnergyPlus input files by applying functions sequentially. As multiple modifications to energy models are commonly required, each sequential function uses the previous function's output and passes a Python List containing the specified model format. This means that the final modified model is simply a series of linear function calls.

Text file manipulations are always isolated to a single line. This makes modifications as succinct and readable as possible for end-users. Functions are defined and imported from the *genEPJ* library. New functions can be added directly to *genEPJ* or added as a 3rd-party file which is imported into the project file.

This approach allows for each manipulation to be added, removed or reordered. Note, that *genEPJ* only executes a function if a condition has been met. In most cases, end-users will set the condition to 'True' (meaning that it is always executed), however, having this capability enables several complicated substitutions such as upgrades applied to a specific HVAC system configuration. Arguments to the function are passed via the 'args' variable. The key-variable pairs should be compatible with the names and types of variables defined in the function (see online documentation).

Each function is auditable meaning that it can be executed in isolation and the changes can be written to a file using the form '`MYFN_file`', where '`MYFN`' is the name of the *genEPJ* function. This is used for debugging

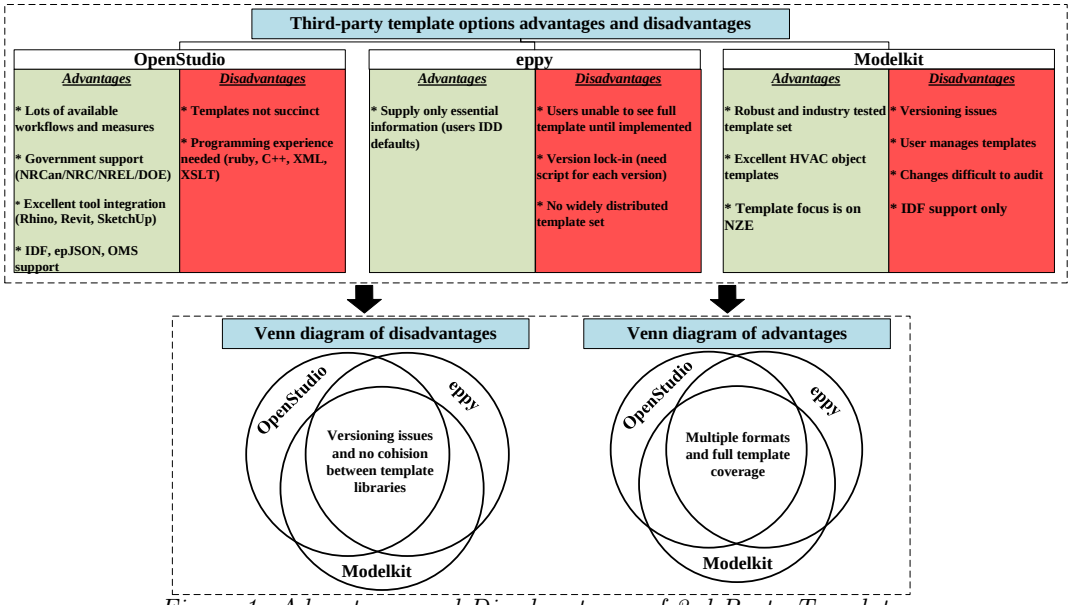


Figure 1: Advantages and Disadvantages of 3rd Party Templates

and quality assurance purposes. An end-user can then compare line-by-line changes using a visual difference tool (a feature common to text file editors). An audit process is necessary since a single *genEPJ* project may manage over a 100,000 lines of text file manipulations. Occasionally, bugs occur in the modelling process. A process is required to isolate bugs and resolve them. *genEPJ* is pre-configured to test all offered functions using the '`genEPJ/tests/run_genEPJ_functions.sh MYFILE.idf`' standalone script (this feature is available only on Linux/Mac OS/Window WSL2). Note that a pass indicates that the function runs as expected and may not indicate that the resulting output runs in EnergyPlus.

By default, *genEPJ* templates look identical to EnergyPlus IDFs, except with a string variable for the missing value. These templates are interpretable for any user familar with EnergyPlus input description files. Listing 1 shows an example template for a '`Schedule:Compact`' object (EnergyPlus comments '`!-`' are removed for brevity).

Listing 1: Exemplar genEPJ *IDF template*

```
def ScheduleCompact_AllDay():
    defaults={
      'temp_low':  '15.0',# Units: C
      'temp_high': '30.0',# Units: C
      'date_start':'6/1', # Format: mm/dd
      'date_stop': '10/1',# Format: mm/dd
    }

    txt_thermsch='''
  Schedule:Compact,
    ${name},
    Temperature,
    Through: ${date_start},
    For: AllDays,
    Until: 24:00,
    Through: ${date_stop},
    For: AllDays,
    Until: 24:00,${temp_high},
    Through: 12/31,
    For: AllDays,
    Until: 24:00,${temp_low};
'''
    return Template(txt_thermsch), defaults
```

This templating approach is based on Python String Templates, defined by PEP 292 (see `https://www.python.org/dev/peps/pep-0292/`), which is a standard library included with any base Python distribution. Text substitutions are conducted inside the function call, not by the string template, allowing for users to override default values (as passed by a function's arguments). If a value is utilized in the template, but no values are supplied, *genEPJ* will raise an error to the end-user. This approach separates concerns around variable substitution into templates and logic around iteration within *genEPJ* functions. An advantage of this approach is that templates can be versioned providing backwards compatibility as new features are added to EnergyPlus. *genEPJ* maintains backwards and forwards compatibility of EnergyPlus versions while still maintaining the readability of templates.

Templates are created and manipulated within *genEPJ* functions. Listing 2 shows an exemplar function for an Ideal HVAC system. As described previously, the syntax for using this function in a *genEPJ* function list would be '`[add_HVAC_ideal, True, {}]`'. The function iterates over zone names and adds the necessary HVACTemplate objects. The identical function in *genEPJ* has additional features such as allowing for the user to specify omitted zones and override thermostat objects.

Listing 2: Exemplar genEPJ *function*

```
from genEPJ import templater as idf_templater
from genEPJ import filter_IDF_objs_raw,
    get_obj_name, templ
def add_HVAC_ideal(objs_all, args={} ):
    # NOTE- objs is a *list* of IDF objects
    new_objs=list(objs_all) # shallow copy
    objs_zon=filter_IDF_objs_raw(objs_all, '
        Zone')
    zone_nms = [get_obj_name(myobj) for myobj
        in objs_zon]
    # genEPJ Template library: 'templ'
    # '*_temp': String Template
    # '*_defs': default dictionary
    ideal_temp,ideal_defs=templ.
        HVACtemplate_Ideal_zone()
    # Foreach Zone:
    for nm in zone_nms:
        # User dictionary overrides
        zone_defs={
              "zone_name": nm,
              # a previously added thermostat
              "thermo_name": thermo_nm,
            }
        # The templater overrides default
            template dict using provided '
            zone_defs' dict
        new_objs.insert(-1, idf_templater(
            zone_defs, ideal_temp, ideal_defs
            ) )
    return new_objs
```

By setting a '*verbose*' flag in *genEPJ* every single template substitution and default variable override is explicity outputted. If run from command line, the end-user can better understand how variables are passed to *genEPJ* and subsequently to template objects.

genEPJ is preconfigured with several standalone scripts which can be called from the commandline located in the '`genEPJ/standalone`' directory. The '`apply_function.py`' script applies a function (provided as a first commandline argument) to a file (specified as a second argument). Any genEPJ function can be called external to the library using this function. The script '`add_comments.py`' adds IDF style comments back to user files post-JSON translation. The script '`check_model.py`' confirms if the necessary IDF objects are present and configured properly for *genEPJ* to function. End-users can choose from over 200 templates distributed in *genEPJ* or extend to over 1000 templates using 3rd-party tools.

Finally, an end-user can choose from several templating approaches, as described in the introduction allowing for significant flexibility to create or reuse existing templates. Presently, *genEPJ* offers experimental capabilities to extract templates from a pre-existing JSON file. This feature can iterate over Zones and objects to apply the extracted templates using JSON formatting. *genEPJ* flexibly supports energy, cost, emissions and resiliency indicators both in parametric and optimization studies.

JSON substitutions can be intermixed with IDF templates as shown in Listing 3. The translation process is made explicit using the function '`swap_IDFJSON`' which interchanges IDF and epJSON formats. eppy support is provided using a function called '`swap_EPPYIDF`' and must also be called explicitly to avoid confusion between which format and templating method is being used.

Listing 3: JSON usage in genEPJ *function list*

```
mygenEPJ_fnlist = [
  [ MYIDFfn1, True, {}],
  [ swap_IDFJSON, True, {}],  # Now in epJSON
      format
  [ mod_JSON, True, {"WindowMaterial:
      SimpleGlazingSystem>Doubleglazed window"
      : {"u_factor": 3.8} } ],
  [ swap_IDFJSON, True, {}],  # Now in IDF
      format
  [ MYIDFfn2, True, {}],
  [ add_comments, True, {}], # Add comments
      back to IDF (which were removed by JSON)
]
```

Note, the symbol '>' represent nested data in a JSON file. Data from objects can quickly be accessed using '`"EnergyPlus Object>Object Name>Data": "Value"`'. *genEPJ* can support nesting of up to five levels. Wildcard substitutions using the '*' symbol are planned but not presently implemented in *genEPJ*. This feature would allow for multiple simultaneous substitutions such as '`"EnergyPlus Object> * >Data": "Value"`' which would allow for substitutions to all matching EnergyPlus objects.

The main advantage of using JSON is that objects can be added and modified without the need for a template or input description dictionary (IDD). Furthermore, each line item is now expressed using an exact '*key: variable*' pair whereas IDF objects required regular expressions to match a position, variable or comment description (which is not a reliable approach to make substitutions). This greatly reduces the upkeep for maintaining templates but comes with the limitation of user readability of input files. Swapping between both formats gives users the best outcomes of both worlds.

Two function are supplied in *genEPJ* to support JSON manipulations. The function '`mod_JSON`' modifies existing JSON objects. The function '`add_JSON`' adds entirely new JSON blocks. When adding entire JSON blocks, information that is not supplied will be set to object defaults.

There is one main disadvantage in translating back and forth between IDF and JSON formats. As of EnergyPlus version 9.6, IDF style comment blocks (i.e. '`!-`') are not translated and thus removed. This hinders the readability of translated IDFs. *genEPJ* addresses this deficiency in the translation process by providing a function called '`add_comments`' which aligns IDF objects

and re-adds comments/line descriptions (using an appropriate IDD file based on the Version object specified in IDF/epJSON). This tool can also be called from the commandline using '`genEPJ/add_comments.py MYFILE.idf`'.

Exemplar *genEPJ* project

This section describes an exemplar *genEPJ* project from conception to completion. We assume that geometry has already be created using a CAD tool such as Sketch-up, Revit, Rhino, ArchiCAD or OpenBIM. Although it is possible to create geometry using Python libraries such as *geomeppy*, energy modellers are commonly provided with a building information model which can be translated directly to EnergyPlus formats (Hong, 2020). Before advancing, this starting model should be executable within EnergyPlus.

genEPJ requires a preparatory file (aka. '*prep* file') as a starting point. This step performs model checks and implements modelling best practices that are common to later reference, proposed and optimize function lists, see Listing 4. Exemplar preparatory steps may include (but are not limited to): removing unused IDF objects (commonly added by third-party tools), rounding significant figures, adding site ground boundary conditions and thermal/internal mass objects. A later step, called '`check_model`', will confirm if sufficient information is provided in this file for most *genEPJ* functions to operate.

Listing 4: Exemplar genEPJ *prep example*

```
from genEPJ import *

mybldg = Building()
mybldg.location_set("Ontario")
prep_fnlist = [
    # Modelling best practices
    [rm_all_unused_objs, True, {}],
    [rm_all_HVAC_simple,True, {}],
    [add_internalmass, True, {}], # Thermal Mass 'Furniture'
    [add_site_BC, True, {}], # Ground shallow/deep temperatures
    [set_primary_energy_factors, True, {'location': mybldg.location } ],
    # ... Add other best practices
    # check that model is compatible with genEPJ
    [ check_model, True, {"interactive": False } ],
]

# genEPJ Input and function lists... (omitted for brevity)

# Create prep file BEFORE ref/prop/opti files
mybldg.tasks= {
  "prep": prep_fnlist,
}
# force_recreate: always recreate IDF/JSON
# run_eplus: Ensure E+ runs on file
mybldg.generate(force_recreate=True, run_eplus=True)
```

The output from '`check_model`' is shown in Listing 5. If the model passes these checks, most functions should run in *genEPJ*. We recommend executing this script both before and after the 'prep' file has been created. The interactive mode of this function requires user feedback to determine if errors exist. The '`check_model`' function should be the last line item added to the preparatory function list. This ensures that earlier modifications and modelling best practices are represented in model checks. Alternatively, '`check_model`' can be run as a standalone script using '`genEPJ/standalone/check_model.py MYFILE_prep.idf`'.

Listing 5: Output from genEPJ*function '`check_model`'*

```
Checks for geometry
..... 'BuildingSurface:Detailed' check model passed
Checks for common substitutions
..... 'Lights' check model passed
..... 'ElectricEquipment' check model passed
..... 'ZoneVentilation:DesignFlowrate' check model passed
Checks for Sizing/Location
..... 'SizingPeriod:DesignDay' check model passed
..... 'Site:Location' check model passed
~... is 'Ottawa Intl...' your desired Location for DesignDay?
~.... 'SizingPeriod:DesignDay Location' check model SKIPPED
All checks passed (with Warning)
```

Once the 'prep' file has been created, simulated and checked, we test to see which *genEPJ* functions are available to us. We run the test script in the '`sim`' directory using '`genEPJ/tests/run_genEPJ_functions.sh MYFILE_prep.idf`' and note that 95% of *genEPJ* functions run without additional configuration. Note that an operational function does not necessarily imply that the resulting output will execute in EnergyPlus. Outputs are saved in the '`tests`' directory. Note, that some functions are not intended to work without the appropriate defaults. At the moment, *genEPJ* assumes essential variables are provided and is programmed to fail if they are not. See the *genEPJ* online documentation for each function for more details.

Variables for reference, proposed and optimized designs are specified using JSON, see Listing 6. Reference ('`ref-value`') and proposed values ('`prop-value`') are typically user defined. Optimization/parametric studies read in the JSON key '`value`' which is set by a higher level metaheuristic or sampling tool. Note, there is a pre-configured optimization tool included with *genEPJ* called *Platypus* (`https://platypus.readthedocs.io`). It requires the specification of variable 'type' for mixed variables (such as integer and real types). This approach allows for both continuous and step definitions (for binary representations) of design

variables. Although a single variable is shown in Listing 6, it is expected at least one variable is required per *genEPJ* function definition.

Listing 6: genEPJ *JSON input file 'opti_inputs.json'*

```
{
    "wall_ins": {
        "units": "m^2K/W",
        "value": 5.0,
        "start": 2.0,
        "stop": 9.0,
        "ref-value": 3.87,
        "prop-value": 7.04,
        "type": "Real",
        "description": "RSI of wall"
    }
}
```

We can now use previously defined design variables within our *genEPJ* function lists. As previously mentioned, we need to define the function, condition and arguments for each entry. The full structure of a *genEPJ* directives file (denoted as '`*_genEPJ.py`') is shown in Listing 7. Note, we show only one variable substitution on lines 26 and 31. Additional entries are required for every design variable and function substitution. Due to paper length restrictions, we describe other templating approaches in the github repository (eppy, OpenStudio, and Modelkit).

Listing 7: Exemplar genEPJ *full example*

```
1  from genEPJ import *
2
3  # Building Config
4  mybldg = Building()
5  # Type determines default thermostat setpoints
6  mybldg.type_set( "multi-residential" )
7  # Location determines primary energy factors/
       GHG emissions
8  mybldg.location_set("Ontario")
9  mybldg.weather="CAN_ON_Ottawa.716280_CWEC.epw"
10 mybldg.basename="myFile.idf"
11 # Determines which genEPJ templates to use
12 mybldg.config.eplus_version_set("9.5.0")
13 # Task list file extensions
14 mybldg.task_suffix= [ "prep", "ref", "prop", "
       opti" ]
15 mybldg.set_filenames()
16 prep_file=mybldg.task_filenames['prep']
17 prep_fnlist = [] # Omitted for brevity
18
19 _inp=find_JSON() # Get JSON, defaults to '
       opti_inputs.json'
20
21 def build_fnlist(json_value_key='value'):
22     wall_ins=_inp['wall_ins'][json_value_key]
23     # ... Remaining key/value pairs here OR
           directly input into fnlist
24
25     # Build Opti:
26     _fnlist=[
27           [ mod_ins, True, { 'loc': 'ExtWall',
                 'resis': wall_ins*5.678 } ],
28
29           # ... Remaining fnlist here
30     ]
31     return _fnlist
32
33 # Reference Building
34 ref_fnlist=build_fnlist('ref-value')
35
36 # Proposed Building
37 prop_fnlist=build_fnlist('prop-value')
38
39 # Parametric/Optimized Building
40 opti_fnlist=build_fnlist('value')
41
42 mybldg.tasks= {
43   "ref":  ref_fnlist,
44   "prop": prop_fnlist,
45   "opti": opti_fnlist,
46 }
47 mybldg.generate(force_recreate=True, run_eplus
       =False)
```

This approach reuses a function list (see '`build_fnlist`') supplying only the appropriate JSON key to differentiate between ref, prop and opti designs. Alternatively, the end-user can choose to define values directly in the '`*_genEPJ.py`' directive file. The directive file can now be executed and a IDF/JSON file will be created. We recommend setting '`run_eplus=False`' and executing EnergyPlus external to *genEPJ* on deployment.

The next step is required only for life-cycle costing (LCC) studies. LCC requires that a reference building be simulated and a reference cash-flow diagram be available for later simulations. *genEPJ* looks for a file named '`ref_cf.py`' to be located in the *genEPJ* project directory ('`sim/ref_cf.py`'). This file is created by enabling the 'ref' building in the task list (see Listing 7). The reference building cash-flow diagram is created using '`cash_flow.py -i FILENAME_prep_ref.IDF`'. The user can copy over the '`*_optiNPV.csv`' file to create the required '`ref_cf.py`' file. The end-user should confirm the reference building cash-flow diagram has been created properly as a quality assurance measure. This is done by rebuilding the cash-flow diagram on the reference building using the newly created '`sim/ref_cf.py`' file and confirming that the net-present value is zero. Several iterations of running the cost analysis and copying over results made be required before this occurs.

After the reference cash flow has been created and the parametric directive file has been executed, the LCC performance of a proposed or parametric design can be determined. Similar to the prevous example, '`cash_flow.py -i FILENAME_prep_opti.IDF`' will create the output file '`*_optiNPV.csv`'. All outputs regarding initial and operational costs are included in this file. An end-user can consider only operation costs by setting the flag '`use_material_cost=False`' in the the '`cash_flow.py`' file.

Coordination of a *genEPJ* study can be streamlined using build tools. Which build tool you choose depends

on your operation system. Window's users may want to consider batch files whereas Linux/MacOS users may prefer Makefiles. We present a simple build system in Listing 8 using a Makefile.

Listing 8: Exemplar genEPJ *build Makefile*

```
#!/usr/bin/make -f

genEPJ=`ls *_genEPJ.py`

all:
 echo "Running genEPJ"
 cp *idf data_temp/
 # Run genEPJ directive file
 -python3 ${genEPJ}
 # Copy data locally (data_temp is symlinked)
 cp data_temp/*opti.idf .
 -${ENERGYPLUS_DIR}/runenergyplus *prep_opti.idf CAN_ON_Ottawa.716280_CWEC.epw
 # Run cost analysis
 python3 cash_flow.py -i *opti.idf --byPeak  # Mech Costs by Peak load
 # Clean up Output data
 -rm -f data_temp/Output
```

For large parametric and optimization studies, we recommend the use of a database to store results. Often, metaheuristics and parametric tools write simulation outputs to a temporary directory. As a final step, key performance indicators should be collected, cleaned and inserted into a database entry. Presently, *genEPJ* uses Linux commands via a bash script (called '`write2db.sh`') to extract data and enter it into the database. Alternatives to this approach can be integrated into the build tool or an external library.

Optimization

It is assumed that many end-users are using scripting approaches to enable parametric or optimization studies. *genEPJ* is pre-configured to use a popular metaheuristics library called '*Platypus*'. To ensure this process is reusable, *genEPJ* expects for projects to be structured in certain manner.

Listing 9 shows the file structure required.

Listing 9: genEPJ *Optimization File Structure*

```
.
+-- db
|   +-- data.db3
+-- genEPJ
+-- opti_inputs.json->sim/opti_inputs.json
+-- MYOPTIMIZER_optiEPJ.py
+-- sim
    +-- MYDIRECTIVE_genEPJ.py
    +-- opti_inputs.json
    +-- cash_flow.py
    +-- makefile
    +-- ref_cf.py
    +-- startsim.sh
    +-- MYFILE.idf
    +-- MYFILE_prep.idf
    +-- genEPJ
    +-- scripts
```

The *genEPJ* project is placed in a subdirectory named '`sim`' which must contain files shown in Listing 7. An identical JSON input file is used for both the optimization and *genEPJ* file creation processes. The folder '`genEPJ`' is the *genEPJ* library. The file '`db/data.db3`' is a pre-configured database for storing simulation results. The '`startsim.sh`' script copies the necessary source files into a temporary directory and begins the build process using '`make`'. The folder '`scripts`' contains several tools such as '`write2db.sh`' which writes EnergyPlus results into a pre-configured database.

Listing 10 presents a skeleton optimization file representing the '`MYOPTIMIZER_optiEPJ.py`' file shown in Listing 9. As configured, the a NSGA-II algorithm is used with three objective functions (energy use intensity, internal rate of return (LCC), and resiliency). The algorithm is run for 2,500 fitness evaluations using all available cores (can be overridden using '`ProcessPoolEvaluator(num_cpus)`'). Results are summarized using the non-dominated Pareto front. A pathway analysis determines the relative signficance of each design parameter (relative to the reference design) using the specified fitness function.

Listing 10: Exemplar genEPJ *optimization template*

```
from genEPJ import opti_EPJ as opti
from platypus import Problem, NSGAII, ProcessPoolEvaluator, nondominated, unique

sql_objfits=['eui', 'resil', 'irr']
def myobjfn(x):
    return opti.objective_function(x, sql_objfits)

# Same JSON input file as genEPJ
json_inputs='opti_inputs.json'

# Mandatory Platypus setup
json_params=opti.get_input_file(json_inputs)
nvars= len( json_params.keys() )
# '0' implies zero constraints
problem = Problem(nvars, len(sql_objfits), 0)
problem.types[:] = opti.build_parameters(json_inputs)
problem.function = myobjfn

#Multiprocessing:
with ProcessPoolEvaluator as ev:
    algo = NSGAII(problem, evaluator=ev)
    algo.run(2500) # 10k fitness calls

#Results
nondom_solns=unique(nondominated(algo.result))
for solution in nondom_solns:
    print( opti.solution2dict(solution), solution.objectives)

# Pathway analysis
ref=opti.get_ref_bldg(problem,json_params)
prop=nondom_solns[0] # Best energy performance
pathway=opti.search_pathway_sequential(ref, prop)
```

The optimization framework can also be used for model calibration. *genEPJ* allows for the specification of measured load profiles, both hourly and monthly, to aid in model calibration (Derakhti et al., 2021). Instead of using conventional optimization functions, the end-user needs to defined their preferred calibration criteria (defaulted to the coefficient of variation root mean square error (CVRMSE)).

Machine Learning using Hourly Load Profiles

This section describes a training approach for a neural network using hourly timeseries data and several weather files using models created in *genEPJ*. This model is exported as a functional mock-up unit to be used in Modelica for district loop simulation.

Listing 11 presents a skeleton machine learning file.

Listing 11: Exemplar genEPJ *machine learning template*

```
from genEPJ import opti_EPJ as opti
from genEPJ import machinelearning as ML

sql_objfits=['eui']
def myobjfn(x):
    return opti.objective_function(x,
        sql_objfits)

# Same JSON input file as genEPJ
json_inputs='opti_inputs.json'
json_params=opti.get_input_file(json_inputs)
nvars= len( json_params.keys() )
# '0' implies zero constraints
problem = Problem(nvars, len(sql_objfits), 0)
problem.types[:] = opti.build_parameters(
    json_inputs)
problem.function = myobjfn

keep_training=True
while keep_training:
    # Population size is 1000
    population=opti.n_rand_indiv(problem,
        1000)

    # Simulate all buildings and keep results
    results=opti.map_eval(population, delete=
        False)

    # Split population into training/testing
        sets
    training_set,testing_set=ML.
        split_population(population)

    # Train Deep Neural Net using GPU
    network_structure=ML.convoDNN(layers,
        num_epochs=120)
    net=ML.train_network(training_set,
        network_structure, device='cuda')

    # Test network
    loss=ML.test_network(net, testing_set)
    if loss<0.01 :
        keep_training=False
# Save network
ML.save_net(net)
```

Conclusion

This paper presents a new library for making EnergyPlus template substitutions using the Python programming language. The contributions of this work are: (i) the development of an open-source software which specializes in deploying multiple template approaches to expedite energy model development, and (ii) novel modelling methodologies, such as the implementation of resilience studies, which can be reused by practioners. Several templating approaches and over 200 preconfigured templates are provided. The capabilities can be expanded to over a 1000 templates using three 3rd-party templating libraries. *genEPJ* takes a multi-objective approach and supports optimization and machine learning workflows. The present limitation of this tool is that it lacks a user-friendly front-end for implementing measures. Future work will add the Rhino Honeybee measures to *genEPJ*'s capabilities.

For additional details, several examples are provided in the github repository for this library: `https://gitlab.com/c5613/genEPJ`.

Acknowledgements

The features of *genEPJ* would not be possible without a thriving open-source community around building energy simulation. In particular, we acknowledge the *Modelkit* project developed by Big Ladder Software, *eppy* developed by Santosh Philip, and OpenStudio by developed by DOE and presently maintained NREL. The implementation of OpenStudio measures and workflows was based on open-source software published by NRCan's Building Technology Assessment Platform (BTAP). We acknowledge both this sourcecode and Canadian-based NECB measures added functionality to this library.

References

Bucking, S., M. Rostami, J. Reinhart, and M. St-Jacques (2022). On modelling of resiliency events using building performance simulation: a multi-objective approach. *Journal of Building Performance Simulation*. Accepted to JBPS and in pre-print.

Derakhti, M., W. O'Brien, and S. Bucking (2021). Impact of measured data frequency on commercial building energy model calibration for retrofit analysis. *Science and Technology for the Built Environment 0*(0), 1–17.

Hong, S. (2020). Geometric Accuracy of BIM-BEM Transformation Workflows : Bridging the State-of-the-Art and Practice. MASc. Thesis in Civil Engineering, Carleton University.